Scalable Data Management and Processing for Genomic Computing

A DISSERTATION PRESENTED
BY
ABDULRAHMAN KAITOUA
TO
THE DEPARTMENT OF ELECTRONICS, INFORMATION AND BIOENGINEERING

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
INFORMATION TECHNOLOGY

POLITECNICO DI MILANO
MILAN, ITALY
NOVEMBER 2016

Thesis advisor: Stefano Ceri Abdulrahman Kaitoua

Scalable Data Management and Processing for Genomic Computing

Abstract

The recent emergence of Next Generation Sequencing (NGS) technologies, in genomics field, produced vast amounts of genomic data. NGS resulted in dropped the cost of sequencing (”reading” in general terms) genomic material very fast. There exist many methods to extract signals from the genomic data, that associate a region of the genome with some interesting information - such as a mutation or a peak of expression. Thus, a new problem is emerging: making sense of these signals, heterogeneous in nature, through new kind of languages that can extract relevant information from various heterogeneous sources, integrate them in a new data management system, and compute interesting results. Biologists say that a huge amount of information is undiscovered within the repositories that have been built in the last decade - therefore, the focus of genomic data management for the next decade is querying and analysing heterogeneous genomic data.

This thesis is about scalable data management and processing for genomic data. We developed a new system which consists of a new query language called GenoMetric Query Language (GMQL), a new data model for heterogeneous data (called Genomic Data Model - GDM), and a new processing engine which embeds scalable genomic algorithms implemented on several data flow engines. The name (GMQL) derives from the ability of the language of dealing with region-based operations which take into account the regions topological location to the reference genome. GDM mediates all existing heterogeneous data formats. In combination, GDM and GMQL introduce a paradigm shift, by providing a high-level, declarative query language which supports data-driven computations.

GMQL is a collective effort which has involved a group of students and professors from Politecnico di Milano. The work reported in this PhD thesis is focused on the design, implementation and validation of the scalable genomic data management and processing system; The architecture of various prototypes of GMQL, GDM, and the scalable genomic processing algorithms.

GMQL is a domain-specific language. In this thesis, we developed several scalable algorithms for genomic processing for serving the needs of GMQL queries. By the use of data flow engines as our target implementations, we capitalize upon existing frameworks which are available today and will be developed in the future by the data management community. In order to increase the parallelism of the genomic algorithms in data flow engines, we proposed new data binning methodologies that are suitable for genomic data and for the nature of the data flow engines. We also used our genomic algorithms for comparing dataflow engines and we developed versions of the scalable algorithms that take advantage of the nature of data flow engines - for example in SciDB, based on multidimensional arrays, we make use of the fast access to slices of the array.

Along with the design of the system and the scalable genomic algorithms, the implementation has gone through phases, and specifically we delivered a first implementation of GMQL, called GMQL V1, based on Hadoop 1 and the target systems Pig; and a second implementation, called GMQL V2, based on Hadoop 2 and the target systems Spark, Flink, and SciDB. The thesis describes the rationale of the two implementations and the process that led from the first to the second prototype.

GMQL engine has a well-designed system architecture with modular organization; system modules can be easily tested, maintained or replaced. GMQL is translated to an intermediate, target-independent representation, based on a Directed Acyclic Graph (DAG), which describes workflows of basic operation nodes. Each node is a primitive operation that implements a specific functionality of GMQL, and operations are mapped to specific target systems; in this way, we can support several implementations to several target systems.

The GMQL engine architecture includes a repository abstraction which is technology-independent, hence several options (local file system, Hadoop File system, scientific database management) are made available by simply including a different repository interface.

Several aspects of the architecture are designed for fast execution on big data sets. This thesis also includes a thorough performance analysis, by comparing GMQL engine V1 to V2, Flink to Spark implementations, and Spark to SciDB implementations. From such studies, we learnt about optimal parameters settings for the scalable genomic algorithms on diverse data sizes and platforms.

We also include some preliminary results of a study which we are conducting in order to field-test GMQL applicability. We implemented a pipeline that uses GMQL for studying gene expression in normal and cancer cells in the context of DNA 3D structure; the study is based on big data sets for about twenty available tissues, thousands of samples for either normal or tumor cases are considered in the study.

In summary, this thesis is a step forward in the development of a systemic approach to scalable genomic data management and processing. Whereas other approaches focused on extracting genomic features from data, our approach is on combining these heterogeneous features so as to solve complex biological problems. We believe that the importance of a systemic approach to genomic data management will grow in the near future, with the availability of huge repositories of genomic data sets.

Contents

Acknowledgments

I would first like to express my gratitude to my supervisor Prof. Stefano Ceri as well as to my co-supervisor Prof. Marco Masseroli for their continuous support, motivation, and patience throughout my research and doctorate studies. I am very grateful to Prof. Ceri for introducing me to the world of Genomics. Without his guidance and persistent help, this thesis would not have been possible.

I would also like to thank Politecnico Di Milano, as well as the GeneData 2020 Project for supporting the funding of this research. Many thanks also go to all the members of DEIB (Department of Electronics, Information and Bioengineering). I thank the Ph.D. international office especially Mr. Marco Simonini and Mrs. Elena Cortiana for all their help throughout the past three years. I also have great gratitude to my colleagues in GeneData 2020 project for team work and for making the work very pleasant; Pietro Pinoli, Vahid Jalili, Francesco Venco, Stefano Perna, Fernando Palluzzi, and Arif Canakoglu.

Last but not least, I am deeply grateful to my parents who raised me to become the person I am today, and continue to support me in all aspects of my life. I also have a deep gratitude to my wife for her support. Their love and encouragement are at the basis of all my achievements.

Storing and processing genome data will exceed the computing challenges of running YouTube and Twitter, biologists warn

Erika Check Hayden, July 2015

1 Introduction

DNA is among the most important scientific discoveries of the last century; it is the foundation of all the forms of life that we known. After the recent advances in technologies for reading the DNA, the next challenge is understanding the signals which are hidden in the DNA. And given that DNA behavior is encoded within big data, cooperation between biologists and computer scientists is much needed. In this chapter we briefly introduce basic notions about DNA organization and processing (sequencing technology, types of experiments and public repositories), then we discuss the motivations and main contributions of the thesis.

1.1 DNA

A proper description of DNA mechanisms is outside the scope of this chapter; we try to give the fundamental concepts necessary to understand the rest of this dissertation. For a much more complete yet concise and simple introduction, the reader is invited to read [1].

1.1.1 Composition and Organization

Deoxyribonucleic acid (**DNA**) is the molecule that encodes the instructions which are necessary for the development and functioning of viruses and all the cells of living organisms. DNA is copied and passed from parents to offspring; sometimes errors or mutations happen and are passed too to future generations, sometimes giving to them advantages. DNA was the first element of life appearing on Earth and for millions of years it evolved, creating the immense variety of forms of life that shaped the face of our planet.

DNA is composed of a series of smaller molecules called nucleotides. There are only 4 types of nucleotides: **adenine** (abbreviated ”**A**”), **thymine** (abbreviated ”**T**”), **guanine** (abbreviated ”**G**”), and **cytosine** (abbreviated ”**C**”). Nucleotides, also called **bases** from one of their components, are very similar to each other. Their chemical structure ensures that they can be chained together, forming a series. The order of the bases is fundamental: specific sub-sequences result in different properties and encode different meanings. In this sense, DNA can be effectively considered a vehicle of information [1].

In its most common organization, DNA is made of two complementary chains of nucleotides, called strands. Bounded together, strands take the form of the well-known double-helix. Being **read** by the biological machinery of cells in opposite directions, they are conventionally defined positive and negative. A fundamental property of these two series is that they are specular. In fact, nucleotides do not only bind chemically to form a chain, but also each base is more weakly connected to another one in the opposing strand. Specifically, A always binds with T and C always binds with G. In this way each strand is a reversed representation of the other; when one is damaged, an organism is able to repair it using the remaining half of the information.

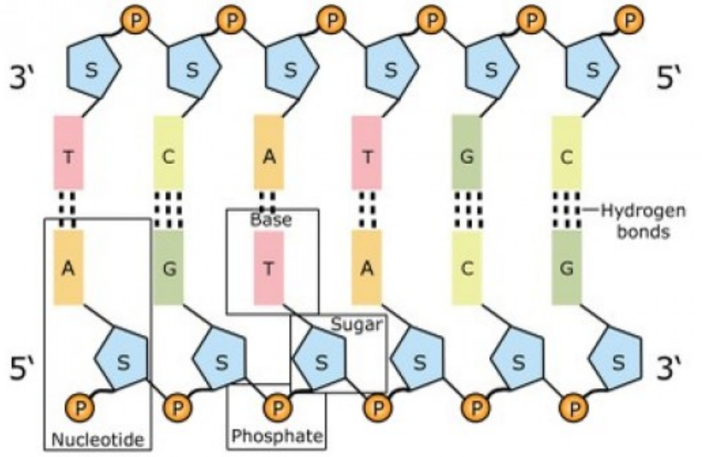


Image adapted from: National Human Genome Research Institute.

Figure 1.1.1: Schematic representation of a segment of DNA molecule

The organization of the DNA varies a lot, but in the majority of organisms DNA is split in one or more *packages*, known as chromosomes, made by a single DNA molecule plus other elements used to maintain a specific organization, a bit like a system of ropes and pulleys. Sometimes, specific parts of a chromosome are so tightly folded that they end up not accessible to other biological machinery present in a cell, while other times zones that are thousands of bases far between each other in the same sequence find themselves adjacent in the 3D space; These spatial characteristics are essential for DNA functioning, as we will see briefly in Chapter 4. The whole genetic content of a cell or organism is called **genome**.

1.1.2 Functioning

To an information scientist, at first sight DNA molecules appears as long strings composed of only 4 different characters. However, the meaning of such characters and their sequence is puzzling. Inside the DNA are **genes**, specific regions that code for proteins, the building blocks of life. In fact, each triplet of nuclotides, called **codon**, has a specific meaning: it can signal the start of a **gene**, the end of it, or one specific **amino acid**, the component of **proteins**. In a very simplified way, genes are sequences of words, and these words explain to a cell how to build its pieces, including DNA molecules themselves. This amazing system is known as **genetic code** [2].

More in details, the genes are copied by the cell machinery into a very similar protein, the ribonucleic acid or **RNA**, contain fragments of DNA information and are free to be carried around the cell. RNA is in fact responsible for transmitting its information to other parts of a cell, for instance the ribosomes, particles that are in charge of finally building the proteins. This process is called **gene expression**.

While fascinating and complex by itself, gene expression does not explain the full functioning of DNA, and genes represent only a part of it. In fact, some proteins are made to bind to specific parts of the DNA, enabling (or disabling) different mechanisms: for instance they could stop genes expression, or else augment it, by increasing the quantity of RNA produced coping a given genes. In many cases, a combination of proteins is necessary to obtain a certain effect. The machinery that copies DNA into RNA is also made by proteins. If DNA was a computer program, it would be one able to modify, regulate and evolve itself and the hardware on which is functioning.

1.1.3 Mutations and Inheritance

As it undergoes many operations and transformations, sometimes DNA can be damaged. In most cases, the cell uses one of the two strands to "backup" the original information, but sometimes it is not possible to do it correctly and **mutations** occur. A mutation can be as little as a single nucleotide taking the place of another one, but sometimes very large sequences are inserted or deleted. In extreme cases entire chromosomes are affected; for example, the Down Syndrome is caused by a chromosome duplication. Some mutations, in particular the smallest ones, do not have any effect and are said to be silent: for example when they happen in non-coding part of the DNA. The genetic code also has some redundancy, and a 'G' becoming a 'A' in a gene could have no effect whatsoever. Other times mutations are fatal and do not permit the cell to function correctly anymore; in such cases, cells destroy themselves or generate tumors. In very rare cases, mutations are beneficial: an organism not only survives, but it has also an advantage. Such changes represent one of the major elements in evolution.

In fact, mutations and the DNA in general are passed from one generation to another, with a mechanism known as inheritance. Organisms reproduce in two ways: either one parent is necessary or two. Single cells and unicellular organisms, like bacteria, reproduce by making a copy of the entire cell DNA. The cell then splits in two independent cells, each one containing an identical copy of the same genome. Complex organisms work differently: in this case two cells containing only half of the original DNA are merged in a new one. Such organisms always have paired chromosomes: one copy from each parent.

Usually in biology the term mutation specifically refers to changes in the DNA in a fully developed organism. However, all organisms belonging to the same species have a different DNA since their birth: the most common variations between individuals are instead known as Single Nucleotide Polymorphism, or **SNP**. More precisely, a SNP is a DNA sequence variation occurring commonly within a population (e.g. 1%) in which a single nucleotide, A, T, C or G , in the genome differs between members of a biological species or paired chromosomes. Such variations are responsible for very evident characteristics, like the color of the eyes, but also more subtle ones, like the capacity to produce a slightly different version of a protein that brings resistance (or weakness) to a specific disease.

1.2 Exposing the Sequence

1.2.1 DNA Sequencing

DNA sequencing is the process of determining the precise order of nucleotides within a DNA molecule. The first sequencing techniques were invented in the 70s: in 1977 the first complete genome of a virus was obtained [3]. In 2001, thanks the so called Shotgun technique, the first draft of a complete human genome was produced [4]. Shotgun sequencing was a process designed for analysis of DNA sequences longer than 1000 base pairs, up to and including entire chromosomes. The method requires the target DNA to be broken into random fragments; after sequencing individual fragments, the sequences can be reassembled on the basis of their overlapping regions.

1.2.2 The Next Generation Sequencing Revolution

Knowing a complete human genome of a specific individual was crucial, but was only the start. After that success, the throughput requirement of DNA sequencing grew by an unpredicted extent and led to laboratory automation and process parallelization. Factory-like enterprises called **sequencing centers** were created, that house hundreds of DNA sequencing instruments operated by cohorts of personnel. However, cost and time remained the bottleneck [5].

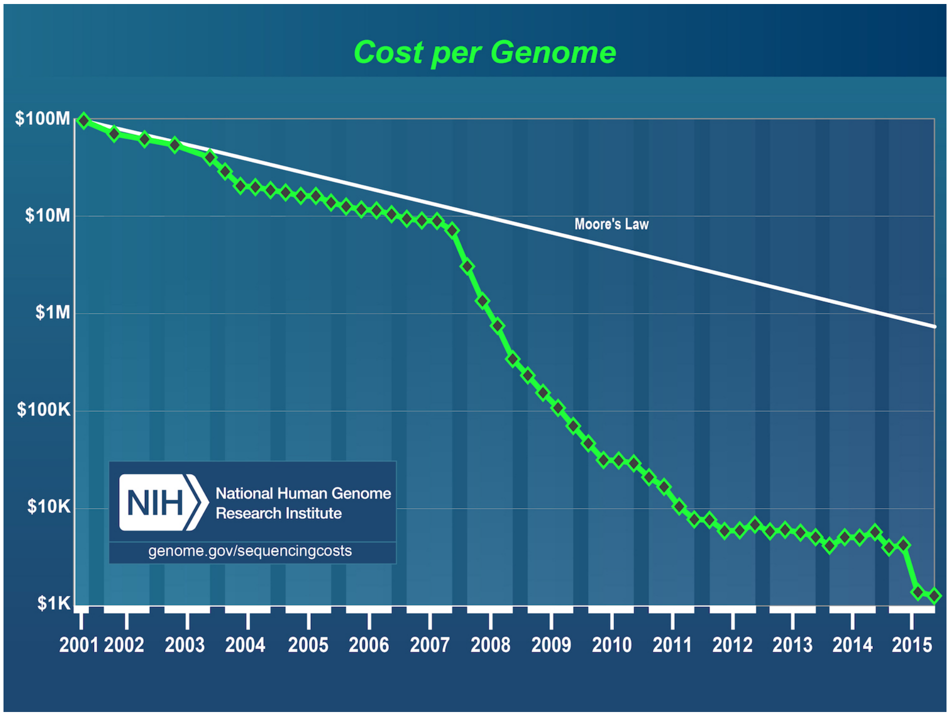


Figure 1.2.1: Genome processing and analysis process.[1]

In the last ten years new techniques emerged, making the sequencing process faster, cheaper and more precise. Such techniques are known as Next Generation Sequencing [6] [7]. Since 2007 the first next-generation sequencing instruments could generate as much data in one day as several hundred of previously used sequencers, and could be operated by a single person [5]. The new techniques also allowed for the first time the identification of all mutations in an organism at the genomic level [5].

By the emergence of NGS, the cost of the genome sequencing dropped from a million to **around one thousand dollars**, see Fig.1.2.1[1]. A great amount of sequenced genomes were generated [8] (the data size trend is doubling with time, Fig.1.3.1.)

Although, the cost of **sequencing** a genome is around **one thousand dollars**[9], The average analysis cost of this sequence could be about**15 thousands dollars**. Having good engineered tools may reduce the efforts spent by bio-informaticians and biologists to analyze the genome, and thus reduce the analysis expense.

A full description of the current sequencing techniques are outside our scope, a nice overview on the subject can be find in [10]. In general, sequencing follows the following steps:

- The genetic material (DNA or RNA) is fragmented and prepared
- The machine isolate the fragments

[1] https://www.genome.gov/sequencingcosts/

- The nucleotides of the fragment are identified in order, producing a file containing short sequences also known as **reads**

The reads can be aligned to each other in order to obtain the so called **genome assembly**, a representation of the original chromosomes from which the DNA originated. Sequence alignment is a complex topic by itself. A survey of sequence alignment algorithms for next-generation sequencing can be found in [11].

1.2.3 NGS Experiments

NGS technology opened the laboratories to many types of biological experiments that permits to study the state of the genome in a cell in particular conditions.

The plain DNA sequencing, sometimes abbreviated as **DNA-seq**, possibly restricted to specific areas of interest, can be used to identify the mutations in a sample respect to a reference genome. After the reads are produced by the machines, they are aligned to an existing assembly, or **reference genome**. The result are specific region **coordinates** on the reference, accompanied with some data; these regions are then directly used or further analyzed with various algorithms.

An important type of NGS experiments are the ones based on Chromatin ImmunoPrecipitation (ChIP), known also as ChIP-sequencing or **ChIP-seq** . These methods make use of antibodies to isolate the regions on which certain proteins are binding. The result of the experiment is a signal which is stronger in the points in which the observed protein is binding more. Various algorithms and tools are used to distinguish the real information from noise, one of the most used is MACS [12].

Another type of experiment we mention is RNA sequencing [13], or **RNA-seq**, also called Whole Transcriptome Shotgun Sequencing. It is used to reveal a snapshot of RNA presence and quantity from a genome at a given moment in time. It is useful to make an estimate of gene expression in a given condition, but also to detect many phenomena, like gene fusions, i.e. when two genes are founded creating a new hybrid gene.

With **DNase-Seq** [14] (DNase I hypersensitive sites sequencing) we indicate a method used to identify the location of regulatory regions, based on the genome-wide sequencing of regions sensitive to cleavage by an enzyme known as **DNase I**. Such sites are thought to be characterized by being highly accessible; therefore, a DNase I sensitivity assay is a widely used methodology in genomics for identifying which regions of the genome are likely to contain active genes.

1.3 Genomic Data Repositories and Consortia

In recent years, numerous public repositories were created for storing and categorizing different types of genomic data. One of the most well-known is the **Gene Expression Omnibus** (GEO) project [15] [16]. GEO is defined as ”a public functional genomics data repository supporting MIAME-compliant data submissions. Array- and sequence-based data are accepted” [17]. The micro-array were experiments that predate the NGS era, when it was only possible to identify few and very specific genomic sequences. MIAME stands for Minimum Information About Micro-Array Experiments and, as the name suggests, MIAME defines a core of data that must be associated to each experiment [18]. GEO works as a public repository were scientists can upload their data to share their experiments within the biology community. It does not have a strong structure a part from the minimum requirements: this provides obvious flexibility but it has also clear limits when it comes to organize or retrieve data from it.

The ENCyclopedia Of DNA Elements project, or in short **ENCODE**, is instead more recent and much more focused on NGS data. The project aims to identify all *functional elements* in the human genome sequence and it is organized as ”an international consortium of computational and laboratory-based scientists working to develop and apply high-throughput approaches for detecting all sequence elements that confer biological function” [19]. The ENCODE Consortium started the pilot phase in September 2003 with the funding of eight projects aimed to the large-scale identification of a variety of functional elements [19]. The results of the pilot phase were published on Nature in 2007 [20] and the produced Data was mainly stored on GEO or on publicly accessible Web sites specifically developed by ENCODE Consortium participants. Files with region data are also accessible though the UCSC Genome Browser, which will be described later in this chapter. An ENCODE portal [21] was created to index all the data, allowing users to query different data types regardless of location. Access to metadata associated with each experiment was also provided. The target of current phase of ENCODE (2012-2016) is to expand the number of cell types, data types and assays and includes the study of both the human and mouse genomes. The last integrative publication on the project can be found at [22].

While ENCODE is focused on studying the functioning of the human (and mouse) genome, the **1000 Genomes Project** is the first project to sequence the genomes of a large number of individuals, to provide a ”comprehensive resource on human genetic variation” [23] [24]. More precisely, the goal of the project is to find the most genetic variants, i.e. with frequencies of at least 1% in the populations studied.

Global Alliance for Genomics and Health [25] is a large consortium of over 200 research institutions with the goal of supporting voluntary and secure sharing of genomic and clinical data; their work on data interoperability has produced a data conversion technology [26].

A wide sector of genomic research is focused on studying cancer. In this respect, a very important project is the The Cancer Genome Atlas, or **TCGA** [27]. TCGA is a USA government project which it was launched in 2006. It grew to include samples from 11,000 patients across 33 tumor types and represents the largest tumor data collection ever to be analyzed for key genomic and molecular

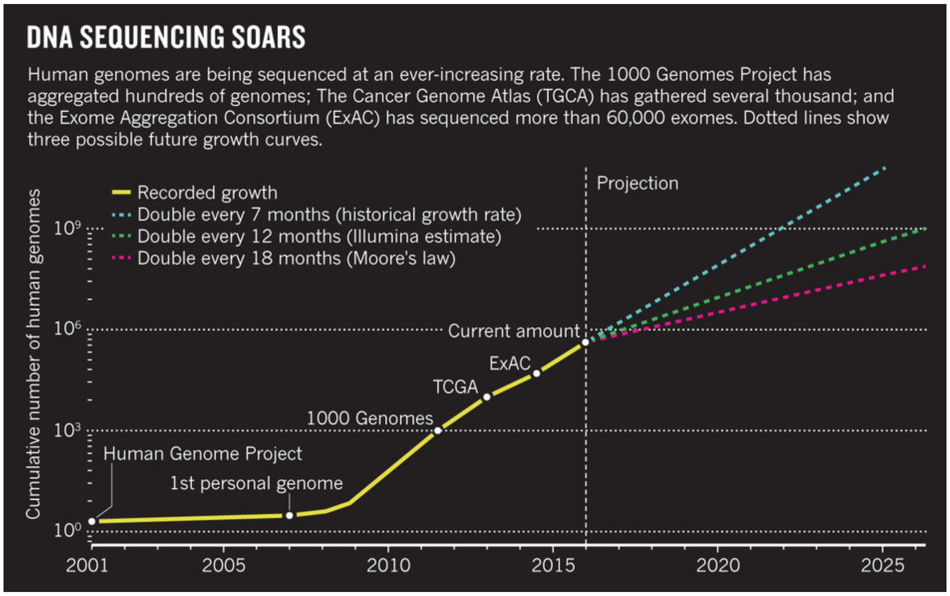


Figure 1.3.1: The trend of PetaBytes of genomic data [8].

characteristics [27]. TCGA data have been used in various studies on comparing cancer types, for example [28].

1.4 Motivation for Thesis work

1.4.1 Tertiary analysis

So far, the bio-informatics research community has been mostly challenged by primary analysis (production of sequences in the form of short DNA segments, or ”reads”) and secondary analysis (alignment of reads to a reference genome and search for specific features on the reads, such as variants/mutations and peaks of expression); but the most important emerging problem is the so-called tertiary analysis, concerned with multi-sample processing, annotation and filtering of variants, and genome browser-driven exploratory analysis. While secondary analysis targets raw data in output from NGS processors by using specialized methods, tertiary analysis targets processed data in output from secondary analysis and is responsible of sense making, e.g., discovering how heterogeneous regions interact with each other.

Tertiary processing consists of integrating DNA features; these can be specific DNA variations (e.g., a variant or mutation in a DNA position), or signals and peaks of expression (e.g., regions with higher DNA read density). Processing can also give structural properties of the DNA, e.g., break points (where the DNA is damaged) or junctions (where DNA creates loops, and then locations which are distant on the 1D string become close in the 3D space), Fig. 1.4.1.

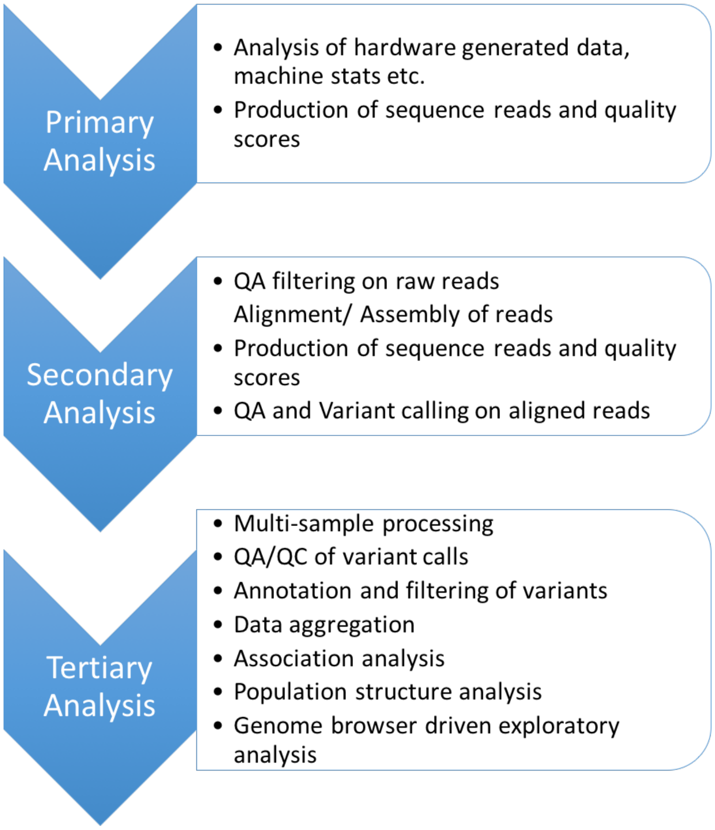

Primary Analysis
• Analysis of hardware generated data, machine stats etc.
• Production of sequence reads and quality scores
Secondary Analysis
• QA filtering on raw reads
Alignment/ Assembly of reads
• Production of sequence reads and quality scores
• QA and Variant calling on aligned reads
Tertiary Analysis
• Multi-sample processing
• QA/QC of variant calls
• Annotation and filtering of variants
• Data aggregation
• Association analysis
• Population structure analysis
• Genome browser driven exploratory analysis

Figure 1.4.1: Genome processing and analysis process.

1.4.2 Limitations in Data Management

While gigantic investments are targeted to sequencing the DNA of larger and larger populations, comparably much smaller investments are directed towards a computational science for mastering tertiary analysis. Bio-informatics resources are dispersed in provisioning a huge number of tools for ad-hoc processing of genomic data, targeted to specific tasks and adapted to technology-driven formats, with little emphasis on powerful abstractions, format-independent representations, and out-of-the-box thinking and scaling. Programming data manipulation operations directly in Python or R is customary.

Another source of difficulty comes from "metadata", which describe DNA region-invariant properties of the biological sample processed by NGS, i.e., the sample cell line, tissue, preparation (antibody used), experimental conditions, and in case of human samples the race, gender, and other phenotype-related traits. This information should be stored in principled data schemes of a "LIMS" (laboratory information management system) and be compliant with standards, but biologists are very liberal in omitting most of it, even in well-cured repositories. Thus managing the **vast amount** of samples coming from different data sources and from a different data formats is challenging.

1.5 Contributions

The GENDATA 2020 project was launched in 2013 by a consortium of nine Italian universities, headed by Politecnico di Milano, to propose and build a new global approach to master the increasing complexity of genomic data. We attempted instead to empower biologists with a high-level, abstract data management paradigm, which could empower them with radically new data processing capabilities.

The effort of the project was on tertiary data analysis, the final steps of the current classical genomic experimental pipeline, after that the DNA has been sequenced (through primary analysis) and after that specific features of the DNA have been identified (through secondary analysis). We understood that it is "mission impossible" for basic computer science to have an impact on primary and secondary analysis: algorithms are biologically driven and already very specialized and efficient. Tertiary data analysis, instead, lacks good systems for "making sense" of genomic information. This work has led to the formalization of GenoMetric Query Language (GMQL). My thesis describes the implementation of GMQL on cloud computing systems.

An implementation of GMQL over cloud computing technologies brings parallelism efficiency, through data distribution (using HDFS [29]) and processing distribution (using MapReduce [30], Spark[31], Flink[32], etc.).

We started to build the GMQL system by translating GMQL script to Apache Pig Latin [33], this produced the first implementation of GMQL – also: GMQL V1 – and allowed us to test the

concept of a query language for genomics in the cloud. GMQL V1 went through several phases of refinement and optimization of the processing and architecture, until it took its shape that it shown in this thesis; but the link between GMQL and the implementation language was too tight. Eventually, we decided that we wanted to have the query language GMQL to be abstracted from the implementation, so that we could change the implementation platform. This lead to a complete redesign of the system and to a second implementation of GMQL – also: GMQL V2- based on an abstract representation of the language, that could then be implemented using several data flow engines.
The innovations that are present in this thesis are summarized below:

- We propose a paradigm shift based on introducing a very simple data model which mediates all existing data formats, and a high-level, declarative query language which supports data extraction as well as the most standard data-driven computations required by tertiary data analysis.

- We built two subsequent systems, GMQL V1 and GMQL V2, thereby achieving the following advantages:

    - The system has a well-designed system architecture with modular organizations; system modules are easily tested, maintained or replaced.
    - Queries are translated into Directed Acyclic Graph (DAG) with operation nodes, that represents an intermediate representation of GMQL. DAG nodes then are implemented in any technology.
    - High level repository abstraction that is technology-independent, hence several repository technologies (local file system, Hadoop File system, Database, MongoDB or, ..) are made available by simply implementing the repository interface.
    - Support a retrieval system that uses inverted index technology [34] for faster file selection. While samples are organized in heterogeneous datasets.
    - The system supports three implementations, on Flink, Spark and SciDB.

- To increase the parallelism of query processing algorithms using data flow engines, we proposed new genomic processing algorithms using dynamic binning of the genomic data. Also versions of the binning algorithms was developed to take advantage of the nature of SciDB, a data base based on multidimensional arrays, by making fast access to slices of the data.

- Algorithms and system optimizations has been proposed and comparisons has been conducted. Performance comparison are conducted for GMQL version one to GMQL version two, Flink to Spark, and Spark to SciDB.

- Implemented a pipeline that uses GMQL for studying gene expression in normal and cancer cells in the context of DNA 3D structure using big data (about twenty available tissues and thousands of patients samples are considered in the study).

1.6 Structure of this thesis

This thesis consists of three parts. **The first part** is called ”GMQL: New paradigm for Data-centric Genomic Computing” and includes three chapters. Chapter 2 presents the GDM data model; Chapter 3 presents the GMQL language and its operations definitions, Chapter 4 shows GMQL in use on a biological application.

The second part of the thesis is called ”GENDATA: System Architecture for Big Data Processing” and also includes three chapters. Chapter 5 provides an introduction to cloud computing technologies used in this thesis work, along with a short introduction to SciDB. Chapter 6 discusses how the GMQL implementation evolved from V1 to V2, and describes the architecture of the repository and the deployment modes of the system. Chapter 7 discusses the implementation of GMQL domain-specific operations with specific algorithms that uses binning strategy.

The third part of the thesis is called ”EVALUATION: Comparing several Implementations of GENDATA” and deals with performance analysis and evaluation. Chapter 8 shows the evaluation of GMQL V1, Chapter 9 shows the evaluation of GMQL V2 along with a comparison to GMQL V1, Chapter 10 shows a comparison between the two implementations in GMQL V2 (Apache Spark [31] and Apache Flink [32]), and finally Chapter 11 shows a comparison between the implementations in Spark [31] and SciDB [35].

1.7 Thesis publications

This thesis includes many contributions which have been recently published:

The vision of GenData 2020 is presented in [2]: Ceri S, Kaitoua A, Masseroli M, Pinoli P, Venco F. ***Data management for next generation genomic computing.*** Proceedings of the 19th Int. Conf. on Extending Database Technology, Bordeaux, March 2016.

The GDM model is presented in [2]: Ceri S, Kaitoua A, Pinoli P, Masseroli M. ***Genomic data modeling for interoperability and next generation genomic data management.*** In: Rojas I, Ortuño F, editors. Proceedings of the 4th International Work-Conference on Bioinformatics and Biomedical Engineering (WBBIO 2016), April 20-22, 2016; Granada, SP. 2016. p. 1-4.

GMQL as a method for genomic processing is presented in: Masseroli M, Kaitoua A, Pinoli P, Ceri S. ***Modeling and interoperability of heterogeneous genomic big data for integrative processing and querying.*** , Methods 2016. DOI: 10.1016/j.ymeth.2016.09.002.

[2] Names are sorted by last name

GMQL is presented in: Masseroli M, Pinoli P, Venco F, Kaitoua A, Jalili V, Palluzzi F, Muller H, Ceri S. ***GenoMetric Query Language: A novel approach to large-scale genomic data management.*** Bioinformatics journal 2015; 31(12): 1881-1888. DOI: 10.1093/bioinformatics/btv048.

GMQL V1 is presented in [2]: Ceri S, Kaitoua A, Masseroli M, Pinoli P, Venco F. ***Data management for heterogeneous genomic datasets.*** IEEE/ACM Transactions on Computational Biology and Bioinformatics 2016. DOI: 10.1109/TCBB.2016.2576447

GMQL V2 is presented in: Kaitoua A., Pinoli P., Bertoni M., Ceri S. ***Framework for Supporting Genomic Operations***, IEEE Transactions on Computers 2016 (in press) DOI 10.1109/TC.2016.2603980

The comparison between Flink and Spark is presented in [2]: Bertoni M., Ceri S., Kaitoua A., Pinoli P. ***Evaluating Cloud Frameworks on Genomic Applications.*** IEEE Big Data Conference, Santa Clara, Nov. 2015.

The comparison between Spark and SciDb is in [2]: Cattaneo S., Ceri S., Kaitoua A., Pinoli P. ***Evaluating Genomic Big Data Operations on SciDB and Spark***, submitted to IEEE Big Data Conference, Washington, Nov. 2016.

Part I

GMQL: New Paradigm for Data-Centric Genomic Computing

Without big data analytics, companies are blind and deaf, wandering out onto the Web like deer on a freeway.

Geoffrey Moore

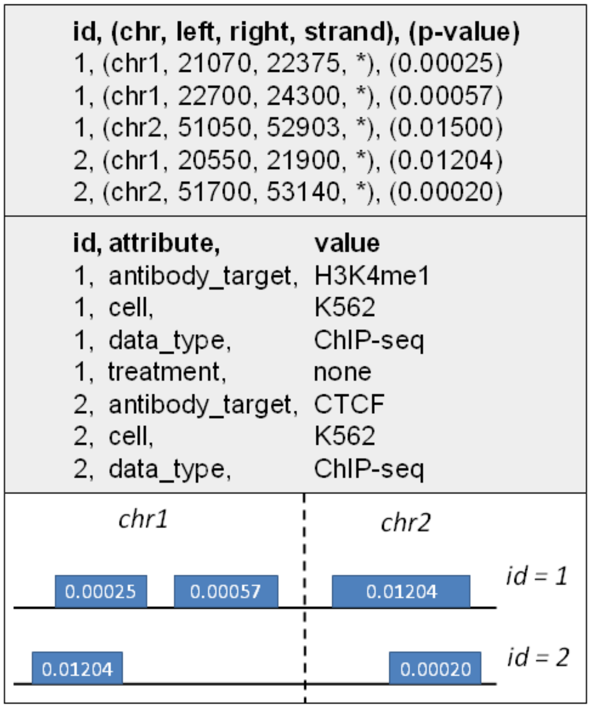
2

Genomic Data Model

A paradigm shift in tertiary genomic data management is brought by the Genomic Data Model (GDM), a simple data model which links genomic features to their associated metadata. This model is able to homogeneously describe semantically heterogeneous data and makes the ground for providing data interoperability, which can be achieved through a high-level, declarative query language for genomic big data. This model is published in [36, 37]

2.1 The Genomic Data Model (GDM)

The Genomic Data Model [38], is based on two entities: the genomic region and the metadata. Regions (upper part of Fig. 2.1.1) have a normalized schema (i.e., a table of typed attributes) where the first five attributes are fixed and the next attributes are variable and reflect the "calling process" that produced them. The fixed attributes include the sample identifier and the region coordinates (the chromosome whom the region belongs to, its left and right ends, and the strand - i.e., the "+" or "–" of the two DNA strands on which the region is read, and "*" if the region is not stranded). The model can be adapted to the rare cases of regions across chromosomes. Metadata (middle part of Fig. 2.1.1) are even simpler. They are arbitrary, semi-structured attribute-value pairs, extended to include the sample identifier. We consider this model a paradigm shift, because a single model describes, though simple concepts, all types of processed data (peaks, signals, mutations, DNA sequences, loops, break points).



id, (chr, left, right, strand), (p-value)
1, (chr1, 21070, 22375, *), (0.00025)
1, (chr1, 22700, 24300, *), (0.00057)
1, (chr2, 51050, 52903, *), (0.01500)
2, (chr1, 20550, 21900, *), (0.01204)
2, (chr2, 51700, 53140, *), (0.00020)

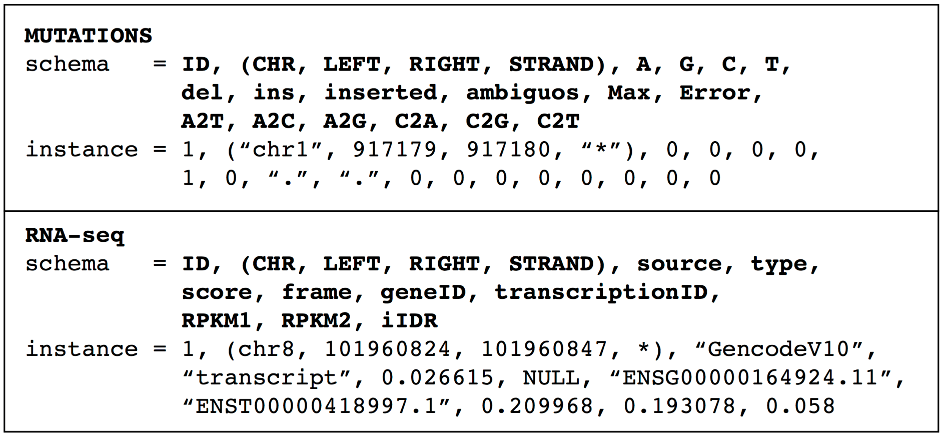
id,	**attribute,**	**value**
1,	antibody_target,	H3K4me1
1,	cell,	K562
1,	data_type,	ChIP-seq
1,	treatment,	none
2,	antibody_target,	CTCF
2,	cell,	K562
2,	data_type,	ChIP-seq

Figure 2.1.1: Genomic Data Model, data files.

The data model is completed by a constraint: data samples can be included into a named dataset when their genomic regions have the same schema. Thus, Fig. 2.1.1 shows the PEAKS dataset for "ChIP-Seq" data with two samples (1 and 2) whose regions fall within two chromosomes (1 and 2) and whose variable part of the schema consists of the attribute P_VALUE (each peak's statistical significance). Note that the sample ID provides a many-to-many connection between regions and metadata of the same sample; e.g., sample 1 has 3 regions and 4 metadata attributes, sample 2 has 2 regions and 3 metadata attributes; regions of the first and the second sample are not stranded (star means not stranded). Metadata tell us that sample 1 has cell "K562" and sample 2 has an antibody target of "CTCF". This example is simple, but we can associate a schema with arbitrarily complex processed data, where typed and named attributes serve the purpose of any numerical or statistical operation across compatible values. An important operation is the schema merging, which allows merging datasets with different schemata (the operation builds a new schema such that fixed attributes are in common and variable attributes are concatenated; in this way, we provide interoperability across heterogeneous processed data.

2.2 Examples

Each dataset is stored within *GenData 2020* using two tables, one for regions and one for metadata; an example of the two tables for representing a particular experiment, called *ChIP-seq*, is shown



```
MUTATIONS
schema   = ID, (CHR, LEFT, RIGHT, STRAND), A, G, C, T,
           del, ins, inserted, ambiguos, Max, Error,
           A2T, A2C, A2G, C2A, C2G, C2T
instance = 1, (“chr1”, 917179, 917180, “*”), 0, 0, 0, 0,
           1, 0, “.”, “.”, 0, 0, 0, 0, 0, 0, 0, 0

RNA-seq
schema   = ID, (CHR, LEFT, RIGHT, STRAND), source, type,
           score, frame, geneID, transcriptionID,
           RPKM1, RPKM2, iIDR
instance = 1, (chr8, 101960824, 101960847, *), “GencodeV10”,
           “transcript”, 0.026615, NULL, “ENSG00000164924.11”,
           “ENST00000418997.1”, 0.209968, 0.193078, 0.058
```

Figure 2.2.1: Examples of schema with one instance for two different types of processed data; coordinates are enclosed within two records.

in Fig.2.1.1. Note that the region value has an attribute P_VALUE of type *float* (representing how significant is the calling of the peak of expression in that genomic region) note also that the ID attribute is present in both tables and provides a many-to-many connection between regions and metadata of a sample; e.g., sample 1 has 3 regions and 4 metadata attributes, sample 2 has 2 regions and 3 metadata attributes[1]. The regions of the two samples are within chromosomes 1 and 2 of the DNA, and both are not stranded.

While the above example is simple, GDM supports the schema encoding of any processed data type, e.g., files for mutations, ChIP-seq, DNA-seq, RNA-seq, ChIA-PET, VCF, and SAM/BAM formats. We use GDM also for modeling *annotations*, i.e. regions of the genome with known properties (such as genes, with their exons and introns). Schema encodings and one exemplar instance of mutations and RNA-seq data samples are decribed in Fig.2.2.1.

[1] Note that the quadruple $\langle id, chr, left, right\rangle$ is not a key of the region table (because a sample can have multiple regions with the same coordinates), and similarly the pair $\langle id, attribute\rangle$ is not a key of the metadata table (because metadata attributes can be multi-valued).

I firmly believe that the next great breakthrough in bioscience could come from a 15-year-old who downloads the human genome in Egypt.

Thomas Friedman

3
The GenoMetric Query Language

The main abstraction for querying genomic dataset have been formalized as a new query language, called GenoMetric Query Language (GMQL); the name derives from its ability of computing distance-related queries along the genome, seen as a sequence of positions. GMQL is a closed algebra over datasets: results are expressed as new datasets derived from their operands. Thus, GMQL operations compute both regions and metadata, connected by IDs; they perform schema merging when needed [39]. A GMQL query (or program) is expressed as a sequence of GMQL operations with the following structure:

```
<variable> = operation(<parameters>) <variables>
```

where each variable stands for a GDM dataset. Operations are either unary (with one input variable), or binary (with two input variables), and construct one result variable.

3.1 General Properties

GMQL operations form a closed algebra: results are expressed as new datasets derived from their operands. All operations produce a result dataset consisting of several samples, whose identifiers are either inherited by the operands or generated by the operation. Each operation separately applies to metadata and to regions; the region-based part of an operation computes the result regions, the metadata part of the operation computes the associated metadata so as to trace the provenance of each resulting sample; identifiers preserve the many-to-many mapping of regions and metadata.

Most GMQL operations, although defined upon two connected data structures, are extension of classic relational algebra operations, twisted to the needs of genomics; they are denoted as *relational*. Three domain-specific operations, called `COVER`, (distal) `JOIN` and `MAP`, significantly extend the expressive power of classic relational algebra.

The main design principles of GMQL are *relational completeness* and *orthogonality*. Completeness is guaranteed by the fact that classical algebraic manipulations are all supported, suitably extended and adapted to comply with region-based calculus. Orthogonality is achieved because no operator can be defined as a suitable expression of all other operators; note that the classic abstractions of *grouping* is supported, with the same semantics, in the unary operations `GROUP` and `COVER`, and similarly *joining* is supported, with the same semantics, in the binary operations `JOIN`, `MAP`, `MERGE` and `DIFFERENCE`.

Compared with languages which are currently in use by the bioinformatic community, GMQL is *declarative* (it specifies the structure of the results, leaving its computation to each operation's implementation) and *high-level* (one GMQL query typically substitutes for a long program which embeds calls to region manipulation libraries); the progressive computation of variables resembles other algebraic languages (e.g. *Pig Latin*, [33]). For all these features, GMQL may inspire a change of paradigm in genomics, along a direction that was indicated long ago by Edward T. Codd's seminal papers.

3.2 Predicates Evaluation

Parameters of several operations include predicates, used to select and join samples; predicates are built by arbitrary Boolean expressions of simple predicates, as it is customary in relational algebra. The region attributes can refer positionally to the schema, i.e., `$0` denotes the first attribute `$1`to the second, and so on. Predicates are either evaluated in the context of regions or of metadata, as follows:

- Predicates on metadata have an existential interpretation over samples: they select the entire sample if it contains some metadata attributes such that the predicate evaluation on their values is true. Formally, for each sample, a simple predicate p expressed as $(A \ comp \ V)$ on metadata M is defined as:

  $$p \iff \exists \ (a_i, v_i) \in M : (a_i = A) \land (v_i \ comp \ V)$$

  When a predicate on metadata uses an attribute which is missing, the predicate is `unknown`; we use three-value (i.e. `true`, `false`, `unknown`) logic for metadata predicates p, and we select samples s for which $p(s)$ is `true` given the above interpretation. The special predicate *missing(A)* is `true` if the attribute A is not present in M.

- Predicates on regions have a classic interpretation: they select the regions where the predicate is true. Legal predicates must use the attributes in the region's schema; when a predicate is illegal, the query is also illegal, and compilation fails.[1] The evaluation of predicates involving two or more regions (essentially join predicates) is defined only when regions have compatible strands; positive and negative strands are incompatible, but they are both compatible with a missing strand.

3.3 Syntactic Conventions

Operations have the general syntax:

```
OUT=OPERATOR(parm-1;..parm-N [; n-parm-1]..[;n-parm-M]) IN-1 [IN-2];
```

Where

- All operations produce an output `OUT`; unari operations apply to a single dataset `(IN-1)`, binary operations apply to two datasets (`IN-1` and `IN-2`).

- `parm` denotes default unnamed parameters for the `OPERATOR`. The semantic of these parameters is inferred from their position.

- `n-parm`: optional parameters that have to be specified in the form of pairs `"name: value"`. The semantics of each one of these parameters is inferred from its name, therefore their position is irrelevant.

Attributes exist in metadata and regions, denoted as follows:

- `<attribute-name>: any-string(.any-string)*` for a generic metadata attribute name.

- `<field-name> : any-string(.any-string)*` for a generic attribute in the region schema.

The prefix `list` denotes a comma-separated list of elements, e.g. <list-field-name> or <list-attribute-name>. For what concerns case sensitivity:

- Region and field names are case sensitive: e.g. `pvalue != pValue != PVALUE`

- GMQL keywords are not case sensitive: e.g. `UPSTREAM == upstream ==UpStReAm`

[1] Region predicates may include metadata attributes, but in such case they are legal iff the metadata attribute is single-valued and not null, and invalid otherwise; in such case, for a given sample, metadata attributes are equivalent to constant values.

3.4 Relational GMQL Operations

We next describe relational operations; they include six unary operations (`SELECT`, `PROJECT`, `EXTEND`, `MERGE`, `GROUP` and `SORT`) and two binary operations (`UNION` and `DIFFERENCE`).

3.4.1 Select

```
<S2> = SELECT([<pm>][;][region: <pr>][;][semijoin: <ps>]) <S1>;
```

Where:

- `<pm>`: Expression whose atomic predicates are in the form: attribute-name $(== \mid > \mid < \mid >= \mid <=)$ `('value' | decimalNumber)`. Atomic predicates are concatenated by means of the OR, AND and NOT operators; e.g. `antibody=='CTCF' AND NOT (weight > 100 OR disease == 'cancer')`.
- `<pr>`: Expression whose atomic predicates are in the form: `field-name` $(== \mid > \mid < \mid >= \mid <=)$ `('value' | decimalNumber)`. Atomic predicates are concatenated by means of the OR, AND and NOT operators; e.g. `pvalue < 0.001 OR label=='promoter'`
- `<ps>`: Semi-join expression in the form: `<list-attribute-name> IN <dataset>`; e.g. `antibody,cell,treatment NOT IN HG_BROAD`

It keeps in the result all the samples which existentially satisfy the predicate on metadata `<pm>` and then selects those regions of selected samples which satisfy the predicate on regions `<pr>`; a sample is legal also when it contains no regions as result of a selection. Identifiers of selected samples of the operand *S1* are assigned to the result *S2*.

`Semi-join` clauses are used to further select samples; they have the syntax: `<A1>..<An> IN <extV>`. Each attribute occurrence `Ai` corresponds to a predicate $p(a_i, a_j)$, where a_i and a_j are attributes with the same name. a_i belongs to the schema of `A`, a_j to the schema of `extV`. The predicate is true for a given sample s_i of *S1* with attribute a_i iff there exists a sample in the variable denoted as `extV` with an attribute a_j and the two attributes a_i and a_j share at least one value. Formally, if M_E denotes the metadata of samples of `extV`, then:

$$p(a_i, a_j) \iff \exists\ (a_i, v_i) \in M_i, (a_j, v_j) \in M_E : v_i = v_j$$

A semi-join clause can be constructed as the conjunction of the above simple metadata predicates that refer to the same variable `extV`. Semi-joins are used to connect variables, e.g., in the example below:

```
OUT = SELECT(semijoin: antibody_target IN EXP2) EXP1;
```

samples of `EXP1` are selected only if they have the same `antibody_target` value as in at least one sample of `EXP2`.

3.4.2 Project

```
<S2> = PROJECT([<Ar1> .. <Arm>]
        [;][metadata: [<Am1> .. <Amn>]
        [;][region_update: <Ur1> AS <f1>, .., <Urh> AS <fh>]
        [;][metadata_update: <Um1> AS <h1>, .., <Umk> AS <fk>]) <S1>;
```

It keeps in the result the metadata (`Am`) and region (`Ar`) attributes expressed as parameters[2]. It can also be used to build new attributes as scalar expressions `fi` (e.g., for metadata the `age` from the `birthdate`; for regions, the `length` of a region as the difference between its `right` and `left` ends). If the name of existing schema attributes are used, the operation updates region attributes to new values. Identifiers of the operand *S1* are assigned to the result `S2`.

3.4.3 Extend

```
<S2> = EXTEND(<Am1> AS <g1>, .., <Amn> AS <gn>) <S1>;
```

It generates new metadata attributes `Am` as result of aggregate functions g applied to region attributes; aggregate functions are applied sample by sample, and resulting tuples are triples with the sample identifier, the attribute name `Am`, and the computed aggregate value. The supported aggregate functions include `COUNT` (with no argument), `BAG` (applicable to attributes of any type) and `SUM, AVG, MIN, MAX, MEDIAN, STD` (applicable to attributes of numeric types). E.g., in the example below:

```
OUT = EXTEND(RegionCount AS COUNT, MinP AS MIN(Pvalue)) EXP;
```

for each sample of `EXP`, two new metadata attributes are computed, `RegionCount` as the number of sample regions, and `MinP` as the minimum `Pvalue` of the sample regions.

3.4.4 Group

```
<S2> = GROUP([<Am1> .. <Amn>]
       [;][meta_aggregate: <Gm1> AS <g1>, .., <Gmn> AS <gn>]
       [;][region_group: <Ar1> .. <Arn>]
       [;][region_aggregate: <Gr1> AS <g1>, .., <Grn> AS <gn>]) <S1>;
```

[2] A syntactic variant (using the keywords `ALL  BUT`) allows to specify only the attributes that are removed from the result; this variant is very useful with datasets having hundreds of metadata.

It is used for grouping both regions and metadata according to distinct values of the grouping attributes. For what concerns metadata, each distinct value of the grouping attributes is associated with an output sample, with a new identifier explicitly created for that sample; samples having missing values for any of the grouping attributes are discarded. The metadata of output samples, each corresponding a to given group, are constructed as the union of metadata of all the samples contributing to that group; consequently, metadata include the attributes storing the grouping values, that are common to each sample in the group. New grouping attributes `Gm` are added to output samples, storing the results of aggregate function evaluations over each group. Examples of typical metadata grouping attributes are the `Classification` of patients (e.g., as cases or controls) or their `Disease` values.

When the grouping attribute is multi-valued, samples are partitioned by each subset of their distinct values (e.g., samples with a `Disease` attribute set both to `'Cancer'` and `'Diabetes'` are within a group which is distinct from the groups of the samples with only one value, either `'Cancer'` or `'Diabetes'`). Formally, two samples s_i and s_j belong to the same group, denoted as $s_i \gamma_A s_j$, if and only if they have exactly the same set of values for every grouping attribute A, i.e.

$$s_i \gamma_A s_j \iff \{v | \exists (A, v) \in M_i\} = \{v | \exists (A, v) \in M_j\}$$

Given this definition, grouping has important properties:

- reflexive: $s_i \gamma_A s_i$
- commutative: $s_i \gamma_A s_j \iff s_j \gamma_A s_i$
- transitive: $s_i \gamma_A s_j \wedge s_k \gamma_A s_i \iff s_k \gamma_A s_j$

When grouping applies to regions, by default it includes the grouping attributes `chr`, `left`, `right`, `strand`; this choice corresponds to the biological application of removing *duplicate regions*, i.e. regions with the same coordinates, possibly resulting from other operations, and ensures that the result is a legal GDM instance. Other attributes may be added to grouping attributes (e.g., `RegionType`); aggregate functions can then be applied to each group. The resulting schema includes the attributes used for grouping and possibly new attributes used for the aggregate functions. The following example is used for calculating the minimum `Pvalue` of duplicate regions:

```
OUT = GROUP(Pvalue AS MIN(Pvalue)) EXP;
```

3.4.5 Merge

```
<S2> = MERGE ([groupby: <AM1>, ..,<AMn>]) <S1>;
```

It builds a dataset consisting of a single sample having as regions all the regions of the input samples and as metadata the union of all the attribute-values of the input samples. When a `GROUPBY` clause is present, the samples are partitioned by groups, each with distinct values of grouping metadata attributes (i.e., homonym attributes in the operand schemas) and the cover operation is separately applied to each group, yielding to one sample in the result for each group, as discussed in Section 3.4.4.

3.4.6 Order

```
<S2> = ORDER([<Am1> [DESC], .., <Amn> [DESC]]
  [;][meta_top: <k> | [;] meta_topg: <k>]
  [;][region_order: <Ar1> [DESC], .., <Arn> [DESC]]
  [;][region_top: <k> | [;] region_topg: <k>]) <S1>;
```

It orders either samples, or regions, or both of them; order is *ascending* as default, and can be turned to *descending* by an explicit indication. Sorted samples or regions have a new attribute Order, added to either metadata, or regions, or both of them; the value of Order reflects the result of the sorting. Identifiers of the samples of the operand *S*1 are assigned to the result *S*2. The clause TOP <k> extracts the first *k* samples or regions, the clause TOPG <k> implicitly considers the grouping by identical values of the first $n - 1$ ordering attributes and then selects the first *k* samples or regions of each group. The operation:

```
OUT = ORDER(RegionCount; meta_top: 5;
         region_order: MutationCount DESC; region_top: 7) EXP;
```

extracts the first 5 samples on the basis of their region counter and then, for each of them, 7 regions on the basis of their mutation counter.

3.4.7 Union

```
<S3> = UNION() <S1> <S2>;
```

It is used to integrate possibly heterogeneous samples of two datasets within a single dataset; each sample of both input datasets contributes to one sample of the result with identical metadata and merged region schema. New identifiers are assigned to each sample.

Two region attributes are considered identical if they have the same name and type; the merging of two schemas is performed by projecting the schema of the second dataset over the schema of the first one. Fields of the first dataset which are missing in the second one are set to NULL value, for all the regions of the second operator. For what concerns metadata, attributes are prefixed with the strings LEFT or RIGHT so as to trace the dataset to which they refer.

3.4.8 Difference

```
<S3> = DIFFERENCE([joinby: <Att1>, .., <Attn>]) <S1> <S2>;
```

This operation produces a sample in the result for each sample of the first operand *S1*, with identical identifier and metadata. It considers all the regions of the second operand, that we denote as *negative regions*; for each sample *s1* of *S1*, it includes in the corresponding result sample those regions which do not intersect with any negative region.

When the JOINBY clause is present, for each sample *s1* of the first dataset *S1* we consider as negative regions only the regions of the samples *s2* of *S2* that satisfy the join condition. Syntactically, the clause consists of a list of attribute names, which are homonyms from the schemas of *S1* and of *S2*; the strings LEFT or RIGHT that may be present as prefixes of attribute names as result of binary operators are not considered for detecting homonyms. We formally define a simple equi-join predicate $a_i == a_j$, but the generalization to conjunctions of simple predicates is straightforward. The predicate is true for given samples *s1* and *s2* iff the two attributes share at least one value, e.g.:

$$p(a_i, a_j) \iff \exists\ (a_i, v_i) \in M_1, (a_j, v_j) \in M_2 : v_i = v_j$$

The operation:

```
OUT = DIFFERENCE(joinby: antibody_target) EXP1 EXP2;
```

extracts for every pair of samples s_1, s_2 of EXP1 and EXP2 having the same value of antibody_target the regions that appear in s_1 but not in s_2; metadata of the result are the same as the metadata of s_1.

3.5 Domain-Specific Operations

We next focus on *domain-specific* operations, which are more specifically responding to genomic management requirements: the unary operation COVER and the binary operations MAP and JOIN.

3.5.1 Cover

```
<S2> = COVER/FLAT/SUMMIT/HISTOGRAM (<minAcc>, <maxAcc>
          [; groupby: <Am1>, .., <Amn>]
          [; aggregate: <Ar1> AS <g1>, .., <Arn> AS <gn>]) <S1>;
```

The COVER operation responds to the need of computing properties that reflect region's intersections, for example to compute a single sample from several samples which are replicas of the same experiment, or for dealing with overlapping regions (as, by construction, resulting regions are not overlapping.)

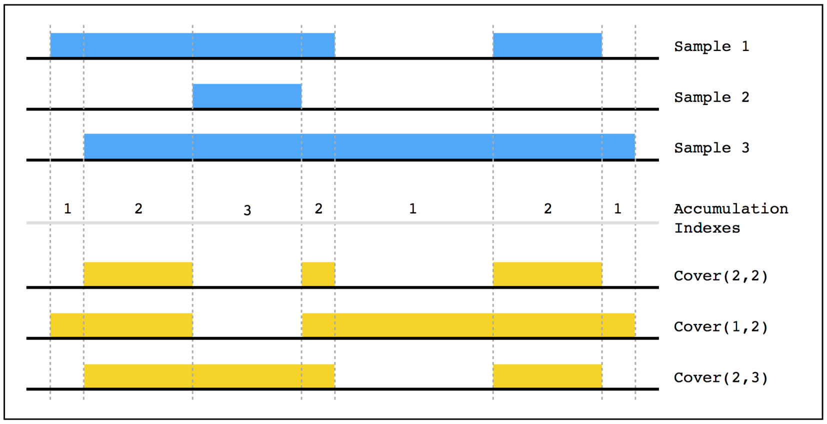


Figure 3.5.1: Accumulation index and COVER results with three different minAcc and maxAcc values.

Let us initially consider the COVER operation with no grouping; in such case, the operation produces a single output sample, and all the metadata attributes of the contributing input samples in *S*1 are assigned to the resulting single sample *s* in *S*2. Regions of the result sample are built from the regions of samples in *S*1 according to the following condition:

- Each resulting region *r* in *S*2 is the contiguous intersection of at least minAcc and at most maxAcc contributing regions r_i in the samples of *S*1 [3]; minAcc and maxAcc are called **accumulation indexes**[4].

Resulting regions may have new attributes *Ar*, calculated by means of aggregate expressions over the attributes of the contributing regions. Jaccard Indexes[5] are standard measures of similarity of the contributing regions r_i, added as default region attributes. When a GROUPBY clause is present, the samples are partitioned by groups, each with distinct values of grouping metadata attributes (i.e., homonym attributes in the operand schemas) and the cover operation is separately applied to each group, yielding to one sample in the result for each group, as discussed in Section 3.4.4.
For what concerns variants:

- FLAT returns the union of all the regions which contribute to the COVER (more precisely, it returns the contiguous region that starts from the first end and stops at the last end of the regions which would contribute to each region of the COVER).

[3] When regions are stranded, cover is separately applied to positive and negative strands; in such case, unstranded regions are accounted both as positive and negative.

[4] The keyword ANY can be used as maxAcc, and in this case no maximum is set (it is equivalent to omitting the maxAcc option); the keyword ALL stands for the number of samples in the operand, and can be used both for minAcc and maxAcc. Cases when maxAcc is greater than ALL are relevant when the input samples include overlapping regions.

[5] The JaccardIntersect index is calculated as the ratio between the lengths of the intersection and of the union of the contributing regions; the JaccardResult index is calculated as the ratio between the lengths of the result and of the union of the contributing regions.

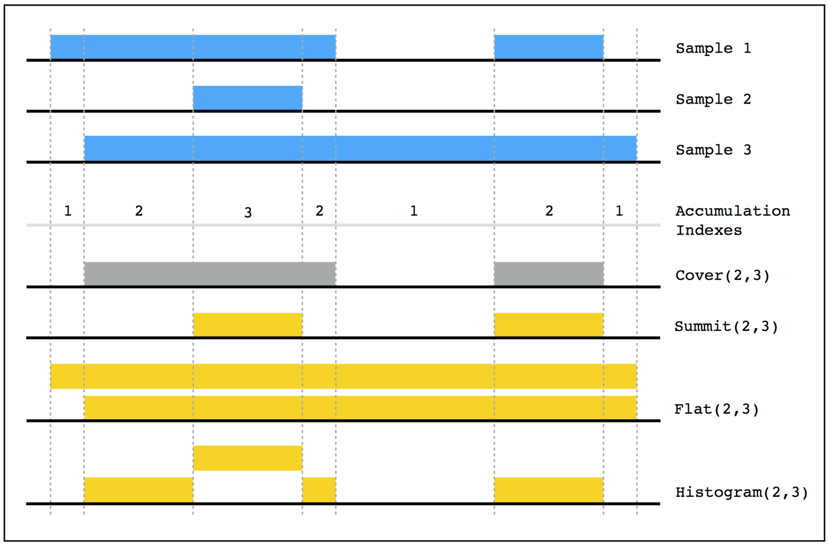


Figure 3.5.2: Accumulation index and `COVER` results with three different `minAcc` and `maxAcc` values. In addition to histogram, Flat, and Summit.

- `SUMMIT` returns only those portions of the result regions of the `COVER` where the maximum number of regions intersect (more precisely, it returns regions that start from a position where the number of intersecting regions is not increasing afterwards and stops at a position where either the number of intersecting regions decreases, or it violates the max accumulation index).

- `HISTOGRAM` returns the nonoverlapping regions contributing to the cover, each with its accumulation index value, which is assigned to the **AccIndex** region attribute.

Example. Fig. 3.5.1 and Fig.3.5.2 show three applications of the `COVER` operation on three samples, represented on a small portion of the genome; the figure shows the values of the accumulation index and then the regions resulting from setting the `minAcc` and `maxAcc` parameters respectively to (2, 2), (1, 2), and (2, 3).

The following `COVER` operation produces output regions where at least 2 and at most 3 regions of EXP overlap, having as resulting region attributes the min `pValue` of the overlapping regions and their Jaccard indexes; the result has one sample for each input `cell`.

```
RES = COVER(2, 3; groupby: cell; aggregate:
            pValue AS MIN(pValue)) EXP;
```

3.5.2 Map

```
<S3> = MAP([<Ar1> AS <g1>, .., <Arn> AS <gn>]
       [;][joinby: <Am1>, .., <Amn>]) <S1> <S2>;
```

MAP is a binary operation over two datasets, respectively called **reference** and **experiment**. Let us consider one reference sample, with a set of reference regions; the operation computes, for each sample in the experiment, aggregates over the values of the experiment regions that intersect with each reference region; we say that *experiment regions are mapped to reference regions*. The operation produces a matrix structure, called **genomic space**, where each experiment sample is associated with a row, each reference region with a column, and each matrix row is a vector of numbers[6]. Thus, a MAP operation allows a quantitative reading of experiments with respect to the reference regions; when the biological function of the reference regions is not known, the MAP helps in extracting the most interesting regions out of many candidates.

We first consider the basic MAP operation, without JOINBY clause. For a given reference sample s_1, let R_1 be the set of its regions; for each sample s_2 of the second operand, with $s_2 =< id_2, R_2, M_2 >$ (according to the GDM notation), the new sample $s_3 =< id_3, R_3, M_3 >$ is constructed; id_3 is generated from id_1 and id_2[7], the metadata M_3 are obtained by merging metadata M_1 and M_2, and the regions $R_3 = \{< c_3, f_3 >\}$ are created such that, for each region $r_1 \in R_1$, there is exactly one region $r_3 \in R_3$, having the same coordinates (i.e., $c_3 = c_1$) and having as features f_3 obtained as the concatenation of the features f_1 and the new attributes computed by the aggregate functions g specified in the operation; such aggregate functions are applied to the attributes of all the regions $r_2 \in R_2$ having a non-empty intersection with r_1. A default aggregate Count counts the number of regions $r_2 \in R_2$ having a non-empty intersection with r_1. For each region, a field named count_LeftDSName_RighDSName is added, storing the result of Count aggregate. The operation is iterated for each reference sample, and generates a sample-specific genomic space at each iteration.

When the JOINBY clause is present, for each sample *s1* of the first dataset *S1* we consider the regions of the samples *s2* of *S2* that satisfy the join condition. Syntactically, the clause consists of a list of attribute names, which are homonyms from the schemas of S1 and of S2; the strings LEFT or RIGHT that may be present as prefixes of attribute names as result of binary operators are not considered for detecting homonyms.

Example. Fig. 3.5.3 shows the effect of this MAP operation on a small portion of the genome; the input consists of one reference sample with 3 regions and three mutation experiment samples, the output consists of three samples, each with the same regions as the reference sample, whose features corresponds to the number of mutations which intersect with those regions. The result can be interpreted as a (3×3) genome space.

In the example below, the MAP operation counts how many mutations occur in known genes, where the dataset EXP contains DNA mutation regions and GENES contains the genes.

[6] Biologists typically consider the transposed matrix, because there are fewer experiments (on columns) than regions (on rows). Such matrix can be observed using heat maps, and its rows and/or columns can be clustered to show patterns.

[7] The implementation generates identifiers for the result by applying hash functions to the identifiers of operands, so that resulting identifiers are unique; they are identical if generated multiple times for the same input samples.

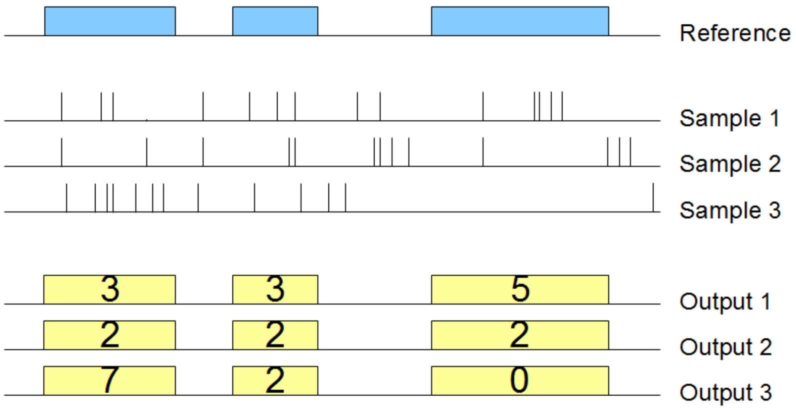


Figure 3.5.3: Example of map using one sample as reference and three samples as experiment, using the Count aggregate function.

```
RES = MAP() GENES EXP;
```

3.5.3 JOIN

```
<S3> = JOIN([<genometric-pred>][;] [output: <coord-gen>]
        [;] [joinby: <Am1>, .., <Amn>]) <S1> <S2>;
```

The JOIN operation applies to two datasets, respectively called **anchor** (the first one) and **experiment** (the second one), and acts in two phases (each of them can be missing). In the first phase, pairs of samples which satisfy the joinby predicate (also called meta-join predicate) are identified; in the second phase, regions that satisfy the **genometric predicate** are selected. The meta-join predicate allows selecting sample pairs with appropriate biological conditions (e.g., regarding the same cell line or antibody); syntactically, it is expressed as a list of homonym attributes from the schemes of S1 and S2, as previously. The genometric join predicate allows expressing a variety of distal conditions, needed by biologists. The anchor is used as startpoint in evaluating genometric predicates (which are not symmetric). The join result is constructed as follows:

- The meta-join predicates initially selects pairs s_1 of *S*1 and s_2 of *S*2 that satisfy the joinby condition. If the clause is omitted, then the Cartesian product of all pairs s_1 of *S*1 and s_2 of *S*2 are selected. For each such pair, a new sample s_{12} is generated in the result, having an identifier id_{12}, generated from id_1 and id_2, and metadata given by the union of metadata of s_1 and s_2.

- Then, the genometric predicate is tested for all the pairs $< r_i, r_j >$ of regions, with $r_1 \in s_1$ and $r_j \in s_2$, by assigning the role of **anchor region**, in turn, to all the regions of *s*1, and then evaluating the genometric predicate condition with all the regions of *s*2. From every pair $< r_i, r_j >$ that satisfies the join condition, a new region is generated in s_{12}.

From this description, it follows that the join operation yields results that can grow quadratically both in the number of samples and of regions; hence, it is the most critical GMQL operation from a computational point of view.

Genometric predicates are based on the **genomic distance**, defined as the number of bases (i.e., nucleotides) between the closest opposite ends of two regions, measured from the right end of the region with left end lower coordinate.[8] A genometric predicate is a sequence of distal conditions, defined as follows:

- UP/DOWN[9] denotes the *upstream* and *downstream* directions of the genome. They are interpreted as predicates that must hold on the region s_2 of the experiment; UP is true when s_2 is in the *upstream genome* of the anchor region[10]. When this clause is not present, distal conditions apply to both the directions of the genome.

- MD(K)[11] denotes the *minimum distance* clause; it selects the K regions of the experiment at minimal distance from the anchor region. When there are ties (i.e., regions at the same distance from the anchor region), regions of the experiment are kept in the result even if they exceed the K limit.

- DLE(N)[12] denotes the *less-equal distance* clause; it selects all the regions of the experiment such that their distance from the anchor region is less than or equal to N bases[13].

- DGE(N)[14] denotes the *greater-equal distance* clause; it selects all the regions of the experiment such that their distance from the anchor region is greater than or equal to N bases.

Genometric clauses are composed by strings of distal conditions; we say that a genometric clause is **well-formed** iff it includes the *less-equal distance* clause; we expect all clauses to be well formed, possibly because the clause DLE(Max) is automatically added at the end of the string, where Max is a problem-specific maximum distance.

Example. The following strings are legal genometric predicates:

[8] Note that with our choice of interbase coordinates, intersecting regions have distance less than 0 and adjacent regions have distance equal to 0; if two regions belong to different chromosomes, their distance is undefined (and predicates based on distance fail).

[9] Also: UPSTREAM, DOWNSTREAM.

[10] *Upstream* and *downstream* are technical terms in genomics, and they are applied to regions on the basis of their *strand*. For regions of the *positive strand* (or for *unstranded regions*), UP is true for those regions of the experiment whose right end is lower than the left end of the anchor, and DOWN is true for those regions of the experiment whose left end is higher than the right end of the anchor. (Remaining regions of the experiment are overlapping with the anchor region.) For the *negative strand*, ends and disequations are exchanged.

[11] Also: MINDIST, MINDISTANCE.

[12] Also: DIST $<=$ N, DISTANCE $<=$ N.

[13] DLE(-1) is true when the region of the experiment overlaps with the anchor region; DLE(0) is true when the region of the experiment is adjacent to or overlapping with the anchor region.

[14] Also: DIST $>=$ N, DISTANCE $>=$ N.

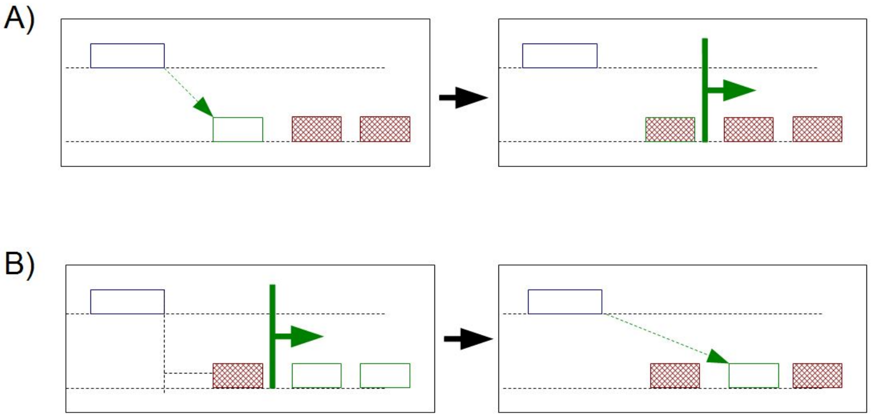


Figure 3.5.4: Different semantics of genometric clauses due to the ordering of distal conditions; excluded regions are gray.`A: MD(1), DGE(100)`; `B: DGE(100), MD(1)`

```
DGE(500), UP, DLE(1000), MD(1)
DGE(50000), UP, DLE(100000), (S1.left - S2.left > 600)
DLE(2000), MD(1), DOWN
MD(100), DLE(3000)
```

Note that different orderings of the same distal clauses may produce different results; this aspect has been designed in order to provide all the required biological meanings.

Examples. In Fig. 3.5.4 we show an evaluation of the following two clauses relative to an anchor region: `A: MD(1), DGE(100)`;`B: DGE(100), MD(1)`. In case A, the `MD(1)` clause is computed first, producing one region which is next excluded by computing the `DGE(100)` clause; therefore, no region is produced. In case B, the `DGE(100)` clause is computed first, producing two regions, and then the `MD(1)` clause is computed, producing as result one region[15].

Similarly, the clauses `A: MD(1), UP` and `B: UP, MD(1)` may produce different results, as in case A the minimum distance region is selected regardless of streams and then retained iff it belongs to the upstream of the anchor, while in case B only upstream regions are considered, and the one at minimum distance is selected.

Next, we discuss the structure of resulting samples. Assume that regions r_i of s_i and r_j of s_j satisfy the genometric predicate, then a new region r_{ij} is created, having merged features obtained by concatenating the feature attributes of the first dataset with the feature attributes of the second dataset as discussed in Section 3.4.7. The coordinates c_{ij} are generated according to the `coord-gen` clause, which has four options [16]:

[15] The two queries can be expressed as: *produce the minimum distance region iff its distance is less than 100 bases* and *produce the minimum distance region after 100 bases*.

[16] If the operation applies to regions with the same strand, the result is also stranded in the same way; if it applies to regions with different strands, the result is not stranded.

1. LEFT assigns to r_{ij} the coordinates c_i of the anchor region.

2. RIGHT assigns to r_{ij} the coordinates c_j of the experiment region.

3. INT assigns to r_{ij} the coordinates of the intersection of r_i and r_j; if the intersection is empty then no region is produced.

4. CAT (also: CONTIG) assigns to r_{ij} the coordinates of the concatenation of r_i and r_j (i.e., the region from the lower left end between those of r_i and r_j to the upper right end between those of r_i and r_j).

Example. The following join searches for those regions of particular ChIP-seq experiments, called histone modifications (HM), that are at a minimal distance from the transcription start sites of genes (TSS), provided that such distance is greater than 120K bases[17]. Note that the result uses the coordinates of the experiment.

```
RES = JOIN(MD(1), DGE(120000); output: RIGHT) TSS HM;
```

3.6 Utility Operations

3.6.1 Materialize

```
MATERIALIZE <S1> INTO file_name;
```

The MATERIALIZE operation saves the content of a dataset S1 in a file, whose name is specified, and registers the saved dataset in the system to make it seamlessly usable in other GMQL queries. All datasets defined in a GMQL query are, by default, temporary; to see and preserve the content of any dataset generated during a GMQL query, the dataset must be materialized. Any dataset can be materialized, however the operation is time expensive; for best performance, materialize the relevant data only.

[17] This query is used in the search of *enhancers*, i.e., parts of the genome which have an important role in gene activation.

We wish to discuss a structure for the salt of deoxyribose nucleic acid. (D.N.A.). This structure has novel features which are of considerable biologic interest.

Rosalind Franklin[a]

[a] Rosalind Elsie Franklin was an English chemist and X-ray crystallographer who made contributions to the understanding of the molecular structures of DNA, RNA, viruses, coal, and graphite.

4

GMQL Case Study: Mapping Gene Expression of Normal and Cancer Cells to Topological Domains, using GMQL

At actual size, a human cell's DNA totals about 3 meters in length, so how does it fit in a cell nucleus? and does the 3D structure affect its functionality? In mammals, DNA is packed in the cell nucleus (just like a zip file); three meters of DNA is packed in a very efficient way.

Recent investigations have shown that our genome and those of other mammals is partitioned into large functional units called topologically associated domains, or TADs for short, see Fig.4.0.1. TADs are very long DNA sections containing one or more genes and their regulatory elements. These TADS are constructed by the binding of CTCF protein to certain regions on the DNA sequence, forming loops, see Fig.4.0.1. An important function of TADs appears to be the formation of self-contained areas of gene regulation, which are at the same time isolated these from neighbouring areas.

The human genome contains around 20,000 protein-coding genes. Surprisingly, the small roundworm C. elegans measuring just one millimetre in length has almost the same number of genes in spite of the fact that humans and roundworms differ radically in their biological complexity. This is because humans are able to better exploit their genetic potential, firstly by modifying gene products and secondly by using the same genes for a number of different functions.

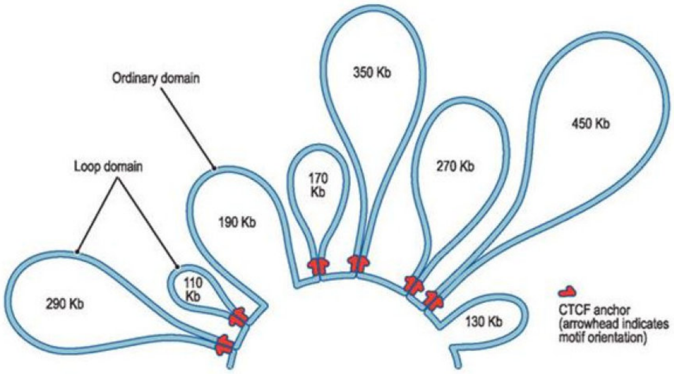


Figure 4.0.1: This illustrative rendering of a 2.1 Mb region on chromosome 20, shows eight domains, six of which are demarcated by loops between convergent CTCF-binding sites located at the domain boundaries. Approximately 10,000 of these loops are present in the human genome.[40]

TADs, topologically associated domains, play a key role in this respect. Each TAD comprises one or more genes together with all their regulatory elements. Their structure has been well conserved throughout evolutionary history and can be found in various cell types, as well as in various species. Regulatory elements within a TAD act only within ”their” TAD; conversely, genes in neighbouring TADs are isolated from their influence. How the separation of one TAD from another is accomplished remains to be shown, but there is increasing evidence that so called boundary elements prevent the contact between TADs.

On the basis of three rare diseases in humans, scientists have now shown that shifts in the boundaries of TADs can lead to significant changes in the regulation of associated genes. TADs are therefore crucial for the proper functioning of genes. The researchers' findings show that hereditary diseases can be caused not only by changes in coding genes themselves but also, surprisingly, by changes in non-coding regions located far from those genes[1].

4.1 Contributions

A recent paper [41] has revealed evidence of the relationship between a specific brain cancer (Glioma induced by specific mutations) and TAD boundaries disruption, an oncogene which causes the Glioma is deregulated by an active enhancer in a contiguous TAD due to the disruption of a TAD boundary acting as insulator, Fig.4.1.1. We believe that other types of cancer might be related to TADs disruption, thus we started a systematic study, using big data as indicators of loops and differential gene expression in normal and tumor cells as in function of boundary disruption (currently ongoing). In what follows, we discuss the use of GMQL for studying the relationships between gene activity in

[1] http://medicalxpress.com/news/2015-05-rare-diseases-destruction-functional-boundaries.html

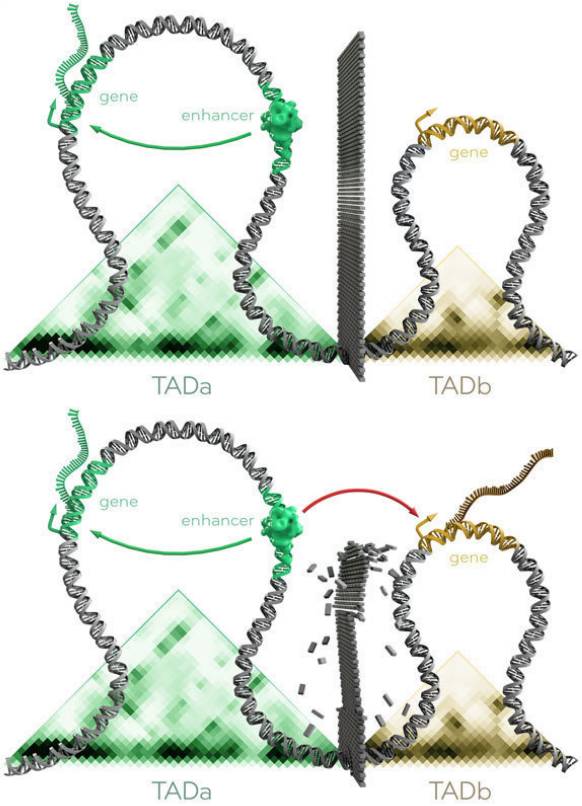


Figure 4.1.1: Illustration of two TADs in a healthy genome. Above: The presence of a "wall" between the two regions means that the regulator (= enhancer) in TADa can only influence the gene in TADa but not the gene in TADb. Below: If the boundary between two TADs is altered or shifted due to a mutation, the regulator/enhancer can also influence genes that are normally shielded from it.

normal and tumor cells and the TADs organization. We built a processing pipeline that uses GMQL to analyse 22 tissues, each with thousands of samples, and each considered for both normal and tumor cells.

4.2 Data sources

Two types of data are needed: RNASeq data for gene expression in normal and cancer cells for a set of patients, and TADs regions. RNASeq data are collected from two repositories: The Cancer Genome Atlas (TCGA)[42] and the Genotype-Tissue Expression (GTEx) [43].

- GTEx data is organized in a two dimensional array with rows as genes, columns as patients samples and cells values as the expression. An additional meta file is attached to this matrix and contains information including the samples' tissue type and if it is a cancer cell or normal cell table 4.2.1.

Genes	$patient_1$	$patient_2$	$patient_3$	...
$Gene_1$	exp_{11}	exp_{12}	exp_{13}	..
$Gene_2$	exp_{21}	exp_{22}	exp_{23}	..
$Gene_3$	exp_{31}	exp_{32}	..	..
$Gene_4$	..	..	..	..
...	..	..	..	..

Table 4.2.1: GTEx matrix data.

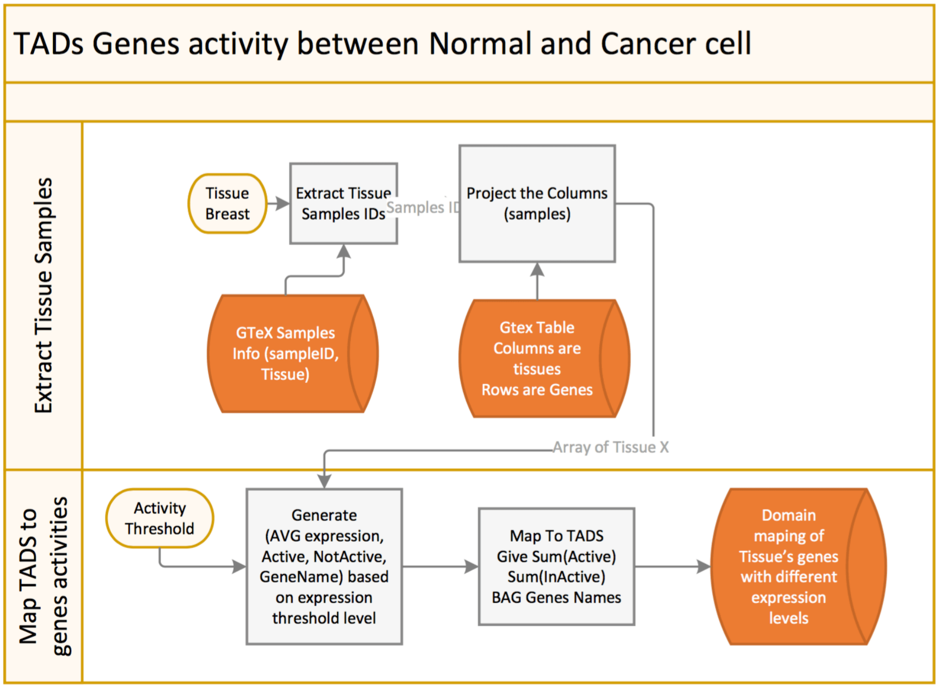
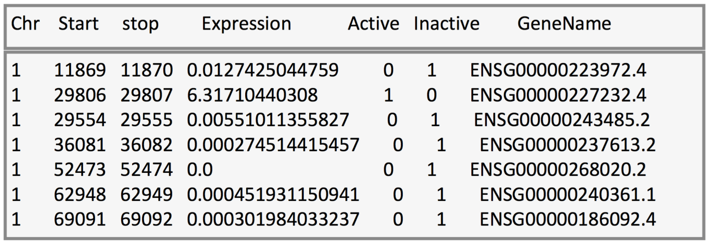

TADs Genes activity between Normal and Cancer cell
Extract Tissue Samples
Tissue Breast
Extract Tissue Samples IDs
Samples ID
Project the Columns (samples)
GTeX Samples Info (sampleID, Tissue)
Gtex Table Columns are tissues Rows are Genes
Array of Tissue X
Map TADS to genes activities
Activity Threshold
Generate (AVG expression, Active, NotActive, GeneName) based on expression threshold level
Map To TADS Give Sum(Active) Sum(InActive) BAG Genes Names
Domain maping of Tissue's genes with different expression levels

Figure 4.3.1: Pipeline for extracting TADs' Genes activity in cancer and normal cells.

- TCGA data is downloaded from local repository in BED format [44], which is suitable for use in GMQL input[2].

TADs data are regions, each with a start and stop coordinate, collected from Gene Expression Omnibus (GEO) repository [3]. GEO stores TADs of nine cell types, for our experiments we used only HMEC and IMR90.



Chr	Start	stop	Expression	Active	Inactive	GeneName
1	11869	11870	0.0127425044759	0	1	ENSG00000223972.4
1	29806	29807	6.31710440308	1	0	ENSG00000227232.4
1	29554	29555	0.00551011355827	0	1	ENSG00000243485.2
1	36081	36082	0.000274514415457	0	1	ENSG00000237613.2
1	52473	52474	0.0	0	1	ENSG00000268020.2
1	62948	62949	0.000451931150941	0	1	ENSG00000240361.1
1	69091	69092	0.000301984033237	0	1	ENSG00000186092.4

Figure 4.3.2: Genes classified to Active and InActive, in respect to the average gene expression.

4.3 Pipeline

Our pipeline has three stages, data extraction, filtering and mapping. The pipeline, Fig.4.3.1, includes modules written in GMQL, R and shell. The data extraction stage applies to the expression data of GTEx, which are accessed by a specific tissue and cell status (normal or tumor). As result of this process, GTEX data table 4.2.1 is partitioned into smaller bi-dimensional tables of genes and patients based on the tissue and the cell status (normal or tumor). This step is performed simply by a shell code that iterates on the tissues names and builds the corresponding matrix. TCGA data is directly downloaded using the TCGA2BED [45] project, having as objective the transformation of TCGA into a GMQL compatible format.

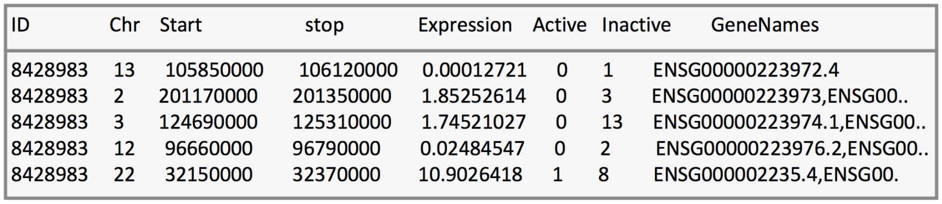
The GMQL code selects the genes from annotation dataset then map them to the data from TCGA2BED, then we take the average of all the experimental samples expression for the same gene region using the command **Cover(ALL,ALL)** with average evaluation. The next step is to classify the genes to active and not active using two **Select** statements and two **Extend** operations, the result from this step is shown in Fig. 4.3.2.

Finally, using GMQ, we map the TADs with the list of genes from the previous step. The map operation shows how the genes overlap with each TAD, and computes the number of active and number of inActive genes in each TAD. The output of this step looks like Fig.4.3.3, and the GMQL code is simply:

```
G1 = SELECT(cell_type==”Liver”) TCGA2BED_RNASeqV2;
L1 = SELECT(type==”IMR_90”) GEO_Domains;
GENES = SELECT(Feature == ’genes’ AND Prov == ’UCSC’) ANNOTATIONS;
G2 = MAP(BAG(geneNames) as Genes, avg(expressison) as expression) GENES   G1;
GENES_EXP_AVG_ALL_SAMPLES = COVER(ALL,ALL,AVG(expression) as GeneAvgExp) G2;
ACTIVE_GENES = SELECT(region:GeneAvgExp > 0) GENES_FROM_ALL_SAMPLES;
INACTIVE_GENES = SELECT(region:GeneAvgExp == 0) GENES_FROM_ALL_SAMPLES;
ACTIVE_GENES1 = EXTEND(region: ACTIVE as 1, INACTIVE as 0) ACTIVE_GENES;
```

[2] http://bioinf.iasi.cnr.it/tcga2bed/

[3] http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE63525



ID	Chr	Start	stop	Expression	Active	Inactive	GeneNames
8428983	13	105850000	106120000	0.00012721	0	1	ENSG00000223972.4
8428983	2	201170000	201350000	1.85252614	0	3	ENSG00000223973,ENSG00..
8428983	3	124690000	125310000	1.74521027	0	13	ENSG00000223974.1,ENSG00..
8428983	12	96660000	96790000	0.02484547	0	2	ENSG00000223976.2,ENSG00..
8428983	22	32150000	32370000	10.9026418	1	8	ENSG000002235.4,ENSG00.

Figure 4.3.3: Number of Active and Inactive genes in each TAD.

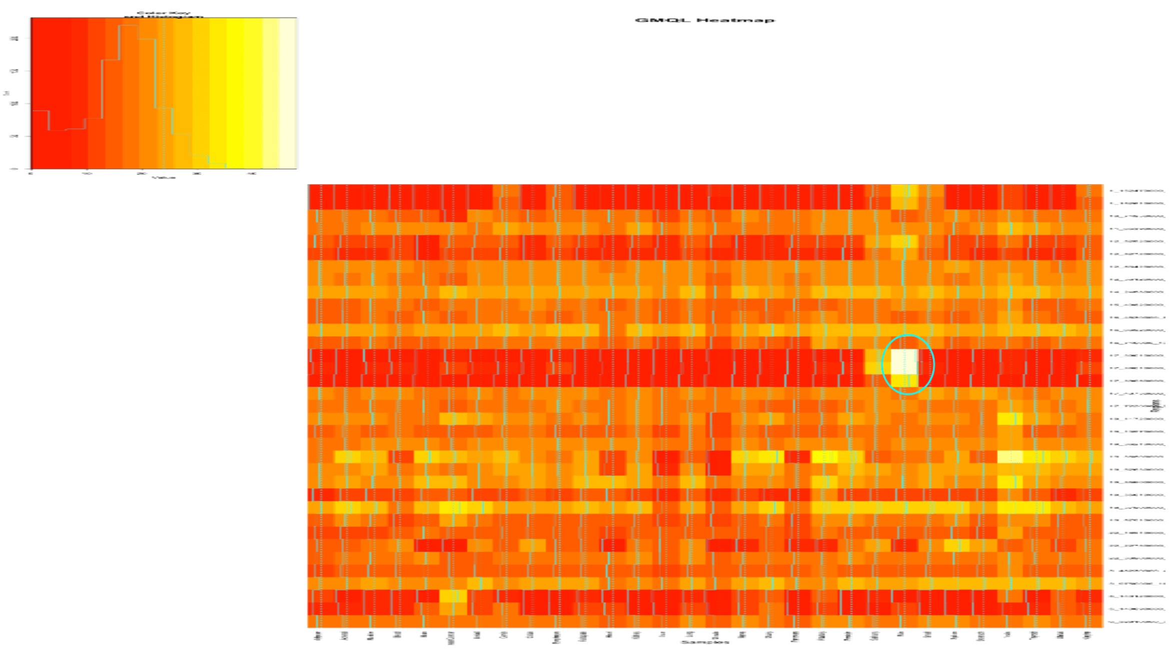


Figure 4.4.1: Heatmap of TADs vs Tissues active genes for normal cells from GTEx.

INACTIVE_GENES1 = **EXTEND**(region: ACTIVE as 0, INACTIVE as 1) INACTIVE_GENES;
GENES_ACTIVITY = **UNION**() ACTIVE_GENES1 INACTIVE_GENES1;
MAPPING = **MAP**(AVG(GeneAvgExp) AS TADAvgExp, SUM(ACTIVE) AS Active,
SUM(INACTIVE) AS InActive) L1 GENES_ACTIVITY;
MATERIALIZE MAPPING **INTO** outActive;

The last step is to get all the tissues in a single two dimensional matrix whose rows are TADS, columns are tissues, and the cells are either active genes number or the ratio of active vs inactive genes.

4.4 Heatmaps

We plotted three heat maps of the results generated by the MAP operation, one related to normal cells using GTEx, and two using TCGA and respectively normal and tumor cells; Fig. Fig.4.4.1 shows genes activity using GTEX for all tissues in 40 TADs, within chromosome 1 (out of a total of

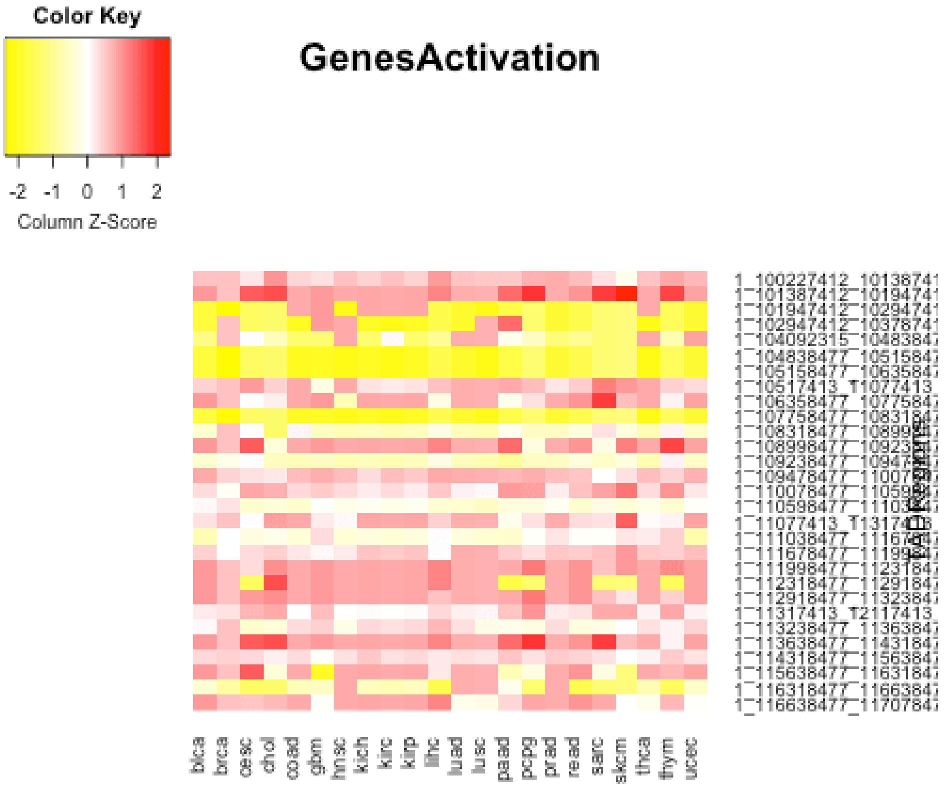


Figure 4.4.2: Heatmap of TADs vs Tissues active genes for TCGA normal patients. 30 TADs are considered of chromosome 1.

3000 TAds).

The comparison of activity levels in tumor (Fig.4.4.3) and normal (Fig.4.4.2) tissues is performed using TCGA both for normal and tumor cells so that the levels of expressions are measured in the same way (we used about **9000** samples). Given the two representations, we can explore for patterns of change, i.e. seeing certain tissues or certain TADS where gene activity rises or drops, and also look for outliers within a specific TAD (relative to all tissues) or tissue (relative to all TADS). Comparative heat maps of gene activity over TADS in normal and tumor cells have not been described so far in the literature, thus this very simple visual description can be used as start point for further investigation by biologists; in our pipeline, we can easily change the choice of TADs and of gene expression level in order to regenerate the two heat maps.

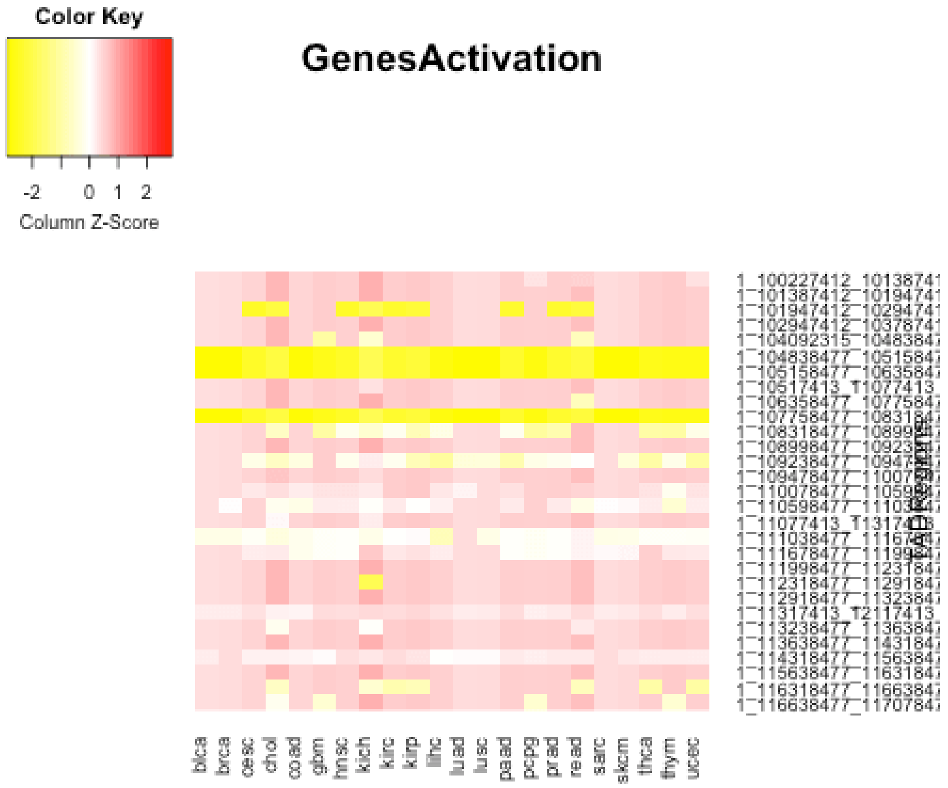


Figure 4.4.3: Heatmap of TADs vs Tissues active genes for TCGA cancer patients. 30 TADs are considered of chromosome 1.

Part II

GENDATA: System Architecture for Big Genomic Data processing

There were 5 exabytes of information created between the down of civilization through 2003, but that much information is now created every 2 days.

Eric Schmidt, Google, 2010

5

Introduction to Big Data technologies

GMQL is designed to run on top of data flow engines. Most of the data flow engines run on top of Hadoop, which is considered as a distributed operating system for cloud engines. We implemented GMQL on top of several cloud engines such as: Apache Pig[33], Apache Spark[31, 46], and Apache Flink[32]; in addition, we also developed an implementation of GMQL on SciDB[35], a multidimensional database designed for big scientific data analysis. In addition to the data management engines, we used some other systems for remote interaction and monitoring, such as Apache Knox[47], Livy[48] and Ganglia[49]. Understanding the features of these engines and systems is essential for designing the software architecture of GMQL; data distribution and shuffling have a great affect on the performance of any cloud computing application, including GMQL.

In this chapter, we show the Hadoop resource management framework and the distributed file system of Hadoop (HDFS[29, 50]) and YARN[51] to describe how application code and data are distributed to a cluster, and how processing is scheduled. We also discuss Livy and Knox services. Then we introduce the Spark, Flink, Pig and SciDB data processing engines.

5.1 Apache Hadoop

Apache Hadoop [52] is an open source framework for distributed storage and processing. Hadoop supports fault tolerance on cluster of machines and distributed storage with the assumption that it is common to have failures on cluster of nodes. The initial versions of Hadoop 1.x consist

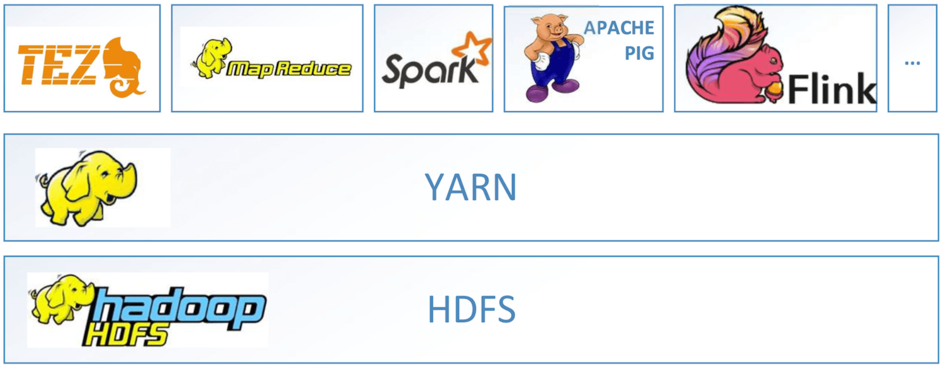


Figure 5.1.1: Part of Hadoop echo system.

only of Hadoop Distributed File System (HDFS) for storage and Map reduce Engine for processing. The subsequent versions of Hadoop 2.x consist of HDFS for storage, Yarn for resource-management, and any other processing engine to run on top of Yarn in addition to MapReduce. Part of the echo system of Hadoop 2.x is shown in Fig.5.1.1.

Historically, Google published a paper on Google File System (GFS) in 2003 [53] and MapReduce at 2004 [30]. Based on these papers the open source community started the development of an open source Distributed File System (DFS) and a distributed processing engine like MapReduce of Google, that in 2006 evolved into Hadoop.

A Hadoop cluster consists of a single master node and a set of slaves nodes. Although Hadoop is considered a framework of a *single point of failure* for HDFS because of the centralized master node, a secondary NameNode offers Hadoop a back up for the distributed file system in case of failure of the master node. Hadoop setup for efficient performance is not a trivial, since there are more than a hundred configurations to be set[54]. Setting Hadoop configurations [55, 56] affects the overall cluster processing. Keeping Hadoop on the default configurations leads to a very poor performance and low resource utilization.

5.1.1 Hadoop Distributed File System

HDFS[29, 50] is a distributed file system that provides high-throughput access to application data. It has many similarities with existing distributed file systems, but also significance differences from them. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets[29].

HDFS breaks down user's files into blocks of data and stores them in distributed places to supports the parallel processing on the data. Block size is configured when the cluster is installed, the default is 64MB; every file with the size greater than the block size is broken down into several blocks.

The files' blocks are replicated into number of copies on the machines, the replication factor is set in HDFS configurations. Increasing the replication factor increases also the system fault tolerance when a node goes offline. HDFS consists of one nameNode process that resides on the master node in Hadoop Cluster and several dataNodes processes that resides on the slave machines (one on each slave machine, master machine can be both master and slave node).

- HDFS NameNode is responsible for storing a look up table that contains information on the files blocks' location in the cluster. This information includes; files blocks and location of blocks in the cluster, replication number and the locations of the replicas for each block.
- A DataNode is responsible to manage data locally on each slave node. DataNode is also responsible to report the data health of the node by managing read/write processes of the node.

5.1.2 YARN as a Distributed Operating System

YARN[51] is the prerequisite for running Enterprise Hadoop, providing resource management and a central platform to deliver consistent operations, security, and data governance tools across Hadoop clusters. YARN also extends the power of Hadoop to new technologies within a data center so that they can take advantage of cost effective, *linear-scale* storage and processing. It provides developers a consistent framework for writing data access applications that run in Hadoop, as described in Fig.5.1.1.

With the introduction of YARN, applications are no longer managed from the Job manager, but from the Application master that can resides on any node in the cluster. The ApplicationMaster is a framework-specific entity that negotiates resources from the ResourceManager and works with the NodeManager(s) to execute and monitor the component tasks. The ResourceManager is the ultimate authority that arbitrates resources among all applications in the system. The ResourceManager has a scheduler, which is responsible for allocating resources to the various applications running in the cluster, according to constraints such as queue capacities and user limits.

Each ApplicationMaster has responsibility for negotiating appropriate resource containers from the scheduler, tracking their status, and monitoring their progress. From the system perspective, the ApplicationMaster runs as a normal container. The NodeManager is the per-machine slave, which is responsible for launching the applications' containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager [51].

5.2 Apache Knox

Apache Knox Gateway [47] is a system that provides a single point of authentication and access for Apache Hadoop services in a cluster. The goal is to simplify Hadoop security both for users (i.e.

who access the cluster data and execute jobs) and operators (i.e. who control access and manage the cluster). The gateway runs as a server (or cluster of servers) that provide centralized access to one or more Hadoop clusters. In general the goals of the gateway are as follows:

- Provide perimeter security for Hadoop REST APIs to make Hadoop security easier to setup and use.
    - Provide authentication and token verification at the perimeter.
    - Enable authentication integration with enterprise and cloud identity management systems.
    - Provide service level authorization at the perimeter.
- Expose a single URL hierarchy that aggregates REST APIs of a Hadoop cluster.
    - Limit the network endpoints (and therefore firewall holes) required to access a Hadoop cluster.
    - Hide the internal Hadoop cluster topology from potential attackers.

5.3 Livy

Livy (currently an alpha release) is a service that enables easy interaction with an Apache Spark cluster over a REST interface. It enables easy submission of Spark jobs or snippets of Spark code, synchronous or asynchronous result retrieval, as well as SparkContext management, all via a simple REST interface or a RPC client library [48]. Livy also simplifies the interaction between Spark from application servers, thus enabling the use of Spark for interactive web/mobile applications. Additional features include:

- support of long running SparkContexts, that can be used for multiple Spark jobs, by multiple clients.
- Sharing of cached RDDs or Dataframes across multiple jobs and clients.
- Multiple SparkContexts can be managed simultaneously, and they run on the cluster (YARN/Mesos) instead of the Livy Server for improved fault tolerance and concurrency.
- Jobs can be submitted as precompiled jars, snippets of code, or via Java/Scala client API.
- Ensure security via secure authenticated communication.

5.4 Apache Pig

Apache Pig [33] is a high-level platform for creating programs that run on Apache Hadoop using a language called Pig Latin. Apache Pig was originally developed at Yahoo Research around 2006 for researchers creating and executing MapReduce jobs on very large data sets. In 2007, it was moved into the Apache Software Foundation. Pig Latin abstracts the programming from the Java MapReduce idiom into a notation which makes MapReduce programming high level, similar to that of SQL for RDBMSs. Pig Latin can be extended using User Defined Functions (UDFs) which the user can write in Java, Python, Java Script, Ruby or Groovy and then call directly from the language. Apache Pig engine translates the Pig Latin script into a set of nodes in a Directed Acyclic Graph (DAG) for pipeline execution. The DAG Nodes are implemented in either MapReduce [30], Apache Tez [57], or Apache Spark [31]. Apache Pig uses lazy execution of the DAG and executes the pipeline in splits instead of sequentially. Pig Latin as a plan text should be submitted to Pig engine for execution, submission can be through Pig shell APIs or PigServer instance using Java application.

Apache Pig can be executed on a single Java Virtual Machine (JVM), which means running all Pig engine's processes as threads in a single process, usually this used for development process while writing Pig Latin code. The single JVM execution receives the input data from Local File System LFS or HDFS. The cluster execution mode is deployed on Hadooop for execution and resource management. For small data sizes (allocating to one experiment in GMQL up to 20 mega bytes), the single JVM mode (called Local execution) is preferable because it skips the complexity of allocating resources and running on a cluster, while it is not suitable for big data processing.

Apache Pig work flow starts by building the execution plan out of the Pig script and then set the execution environment to either Local (single JVM), MapReduce, Spark, or Tez. Then the Pig compiler compiles the Pig Script to the target execution environment. The UDF along with the compiled code are shiped to HDFS for execution. The output result will reside on HDFS.

5.5 Apache Spark

Spark[31, 46, 58, 59] was initially developed at Berkeley University as part of the AMP (Algorithms, Machines, People) [1] and became an open-source project in 2009; it is now a much larger Apache project, with more than 400 developers from over 50 companies [31, 46, 60].

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

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

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 consists in selecting algorithms based on the partitioning option that minimizes data transfer between workers.

Spark includes set of operators including *Map, flatMap, mapPartition, Reduce, Repartition, Filter, Union, cartesian, coGroup, SortByKey, CountByKey*; The above operations are also denoted as *transformations*, 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* [58] version, that supports structured queries over distributed dataset DataFrame[58], 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) that facilitate key-based operations.

- Support of check pointing of operations [31], 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.

5.6 Apache Flink

Flink[32] was developed as a cooperative project within Technical University (TU) and Humboldt University (HU) in Berlin. It is now developed as an open-source Apache project. 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 to optimize 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 can be used as a *streaming engine* because it is able to send data from one operation to the next tuple by tuple, without waiting the filling of intermediate buffers (or micro batches). 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 [61], 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.

5.7 SciDB

SciDB [62, 63] is a new open-source data management system intended primarily for use in application domains that involve very large scale array data; for example scientific applications such as astronomy, remote sensing and climate modelling, bio-science information management, as well as commercial applications such as risk management systems in the financial services sector, and the analysis of web log data. A specific SciDB extension is applied to genomic data.

SciDB's Multidimensional Array Clustering (MAC) storage subsystem is built to efficiently store multi-attribute multidimensional arrays, exceeding tens of terabytes in size, in a distributed DBMS, while facilitating array-style slicing and lookup operations as fast as possible [35].

The most common operations that SciDB should be efficiently support are: slices, returning all relations for some values of a certain dimension, multiple horizontal slices, with projections that only returns a subset of the attributes, and sub-region selections, rectilinear regions defined by dimensions. Fig.5.7.1, Fig.5.7.2, and Fig.5.7.3 show these three operations in order to understand what

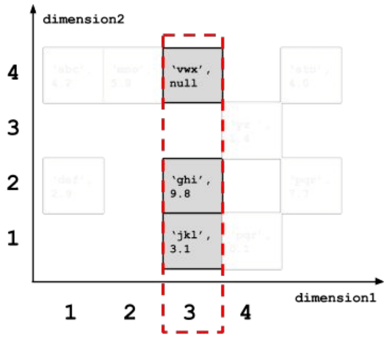


Figure 5.7.1: Slice on dimension.

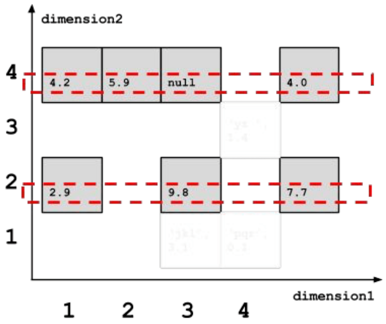


Figure 5.7.2: Slice and projection.

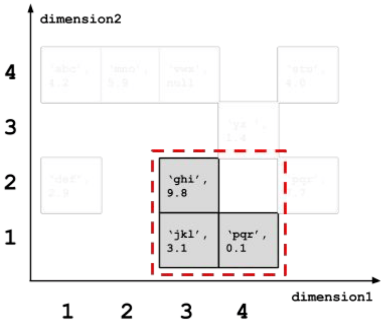


Figure 5.7.3: Sub-region selection.

they mean on an array data structure. In order to meet the above objectives, MAC was designed using the following features.

- Columnar storage with respect to attribute. Even if an array has hundreds of attributes, only the attributes requested by the query are read off on the disk.

- Algebraic indexing. SciDB can very quickly figure out the on-disk physical location of a cell or a requested block of cells. The indexing method utilizes a combination of hashing as well as lookup structures that are automatically maintained as the data sizes grow.

- Clustering. SciDB clusters data in it's chunks so that co-local regions of the logical array are co-located in the physical data. This ensures that, for a *slice* or *between* query, the number of disk locations that need to be visited to retrieve the required data is minimized.

SciDB run as a network of processes, or *instances*, each responsible for a subset of the overall data, usually on a cluster of multiple physical computers, or *nodes*. Each instance keeps data in its own file system directory, which is usually located on an independent storage device, but may also be part of a large, shared storage subsystem.

Every MAC works is broken into a grid of fixed-size rectilinear chunks that partition the multidimensional space of the array. Each chunk is then assigned to a particular instance using a hash function over the chunk's coordinates in the array space. Chunks for different attributes are stored separately.

Figure 5.7.4 shows an example. The attribute `a1` and `a2` are stored separately. The chunk size is set to 3 × 3 and there is a total of 9 chunks' worth of data for each attribute. The diagram shows the chunks at position `{0,0}` are assigned to instance 1, whereas the chunks at `{3,3}` are assigned to instance 2 by virtue of hashing. Prior to being written to disk, chunks are run-length encoded, compressing out frequently repeated values. There is also a hidden *empty bitmap* (EBM) attribute

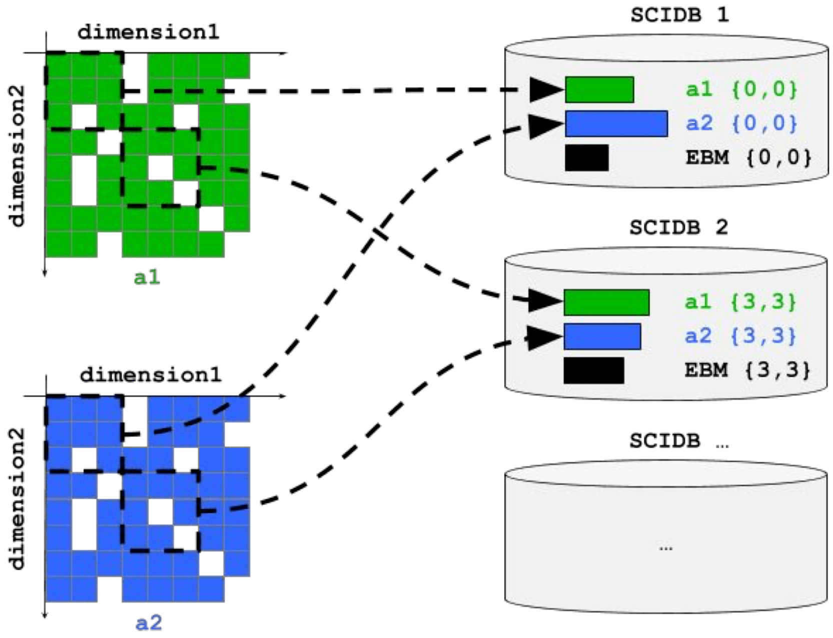


Figure 5.7.4: Chunking and distribution mechanism, from *Paradigm4* documentation.

stored as a run-length encoded bitmask that encodes the positions of non-empty cells inside each chunks.

The chunking aspect of the architecture ensures that only a few chunks are required to satisfy a particular dimensional query. For example, to project `a1` from a slice along `dimension1=4`, we would only need to scan 3 of the chunks: `a1{0,3}`, `a1{3,3}` and `a1{6,3}`. Attribute `a2` is not touched since it is not requested. Only 3 chunks out of 18 total would have to be read off the disk. Moreover, the hashing ensures that the three chunks are likely on separate SciDB instances, so that the disk reads can happen in parallel.

In DBMS-theoretic terms, SciDB is said to *automatically index and cluster data on dimensions*. *Indexing* means that, given particular dimension coordinates, the system can retrieve data at those coordinates without having search through most of the data; it is just a quick matter of locating the right chunks. *Clustering* means that data which are close to each other in the array coordinate system are likely stored in the same region on disk.

The first 90 percent of the code accounts for the first 90 percent of the development time...The remaining 10 percent of the code accounts for the other 90 percent of the development time.

Tom Cargill

6
GMQL System Architecture

The main contribution of this thesis is the GMQL Engine[36, 38, 39], that was built and improved through several stages during the course of this Ph.D. thesis. We started by building a simple translator from GMQL to Pig Latin language (GMQL version 1.x). Then, we developed a language-independent system architecture of GMQL in which the GMQL compiler generates directed acyclic graphs (DAGs) as intermediate representation, and then DAG nodes are implemented using different cloud computing technologies (GMQL version 2.x). Along the development of GMQL engines V1 and V2, we improved the GMQL repository (from 1.x to 3.x), the web services (from 1.x to 2.x) and the web interface (from 1.x to 2.x).

The rest of this chapter is about the GMQL engine development through its different versions, then the repository architecture and finally the web services and web application. We also discuss the different deployment modes that we developed in order to adapt to different hardware architectures.

6.1 GMQL V1

GMQL v1 [38] uses a syntax-directed translation from GMQL into Pig Latin, implemented in Racket language[64]. Every GMQL command is translated to a set of Pig Latin commands. In order to implement an efficient retrieval system for GMQL data, we developed a java application that orchestrates the data retrieval (as described in section 6.4), the data management (repository) and the data processing (Pig engine). The GMQL V1 system architecture in shown in Fig.6.1.1. It

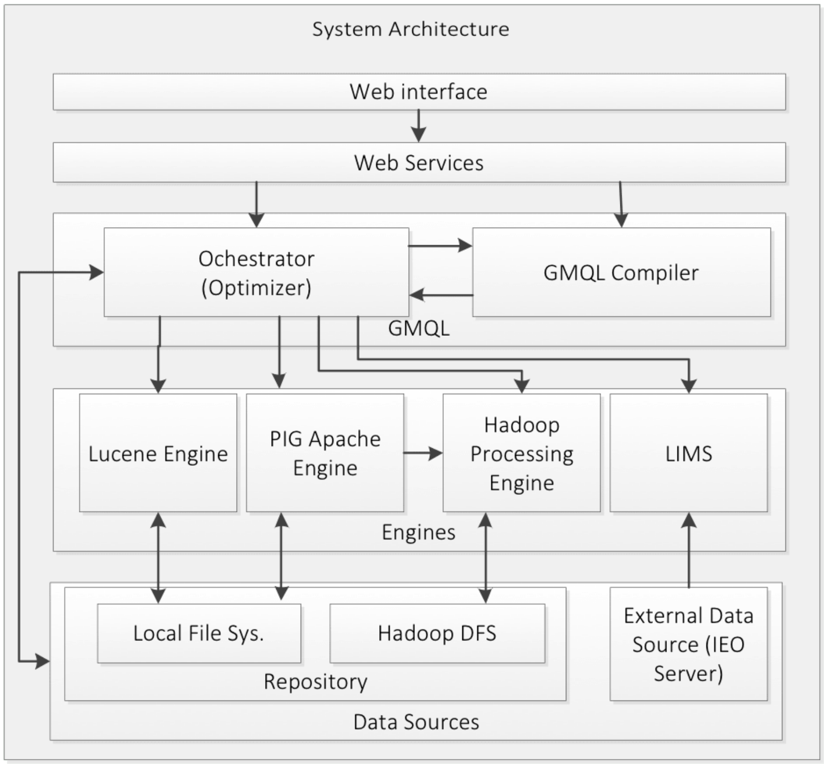
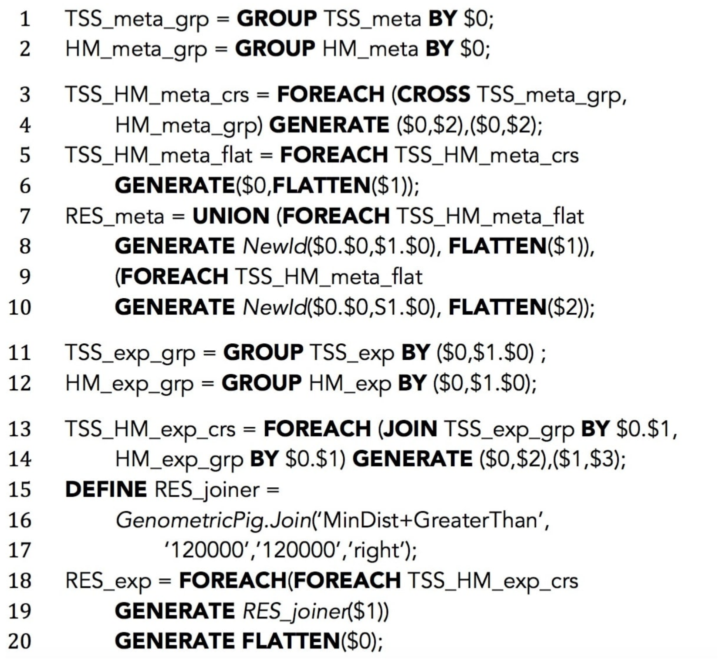


Figure 6.1.1: GMQL V1 Architecture.

includes the **repository layer**, the **engine layer** and the **GMQL layer**, which in turn consists of an **orchestrator** and a **compiler**, and is accessible through a **web service API**. We published this system in [38],[39].

6.1.1 Translator

Our developed translator has two components, the *lexer* and the *parser*. The former one scans the GMQL query and generates a list of tokens; the latter one identifies sub-sequences of the token list which correspond to grammar rules, using a LALR(1) algorithm [65]. When a statement is semantically valid, the compiler first infers the schema of the newly introduced variable, then updates the internal state and finally emits the Apache Pig code that performs the requested operation. The internal state contains the name and schema of each variable which is either generated or mentioned in the query. We implemented the GMQL translator in Racket [64], a general-purpose functional programming language in the Lisp/Scheme family associated with a powerful set of tools; Racket has advanced macro system and higher order functions, which facilitate the production of a concise, clean and safe code. Fig. 6.1.2 shows the translation of the `JOIN` in the following example code. The following GMQL program searches for those regions, of particular ChIP-seq experiments, called histone modifications (HM), that are at a minimal distance from transcription start sites of genes (TSS),



```
1   TSS_meta_grp = GROUP TSS_meta BY $0;
2   HM_meta_grp = GROUP HM_meta BY $0;
3   TSS_HM_meta_crs = FOREACH (CROSS TSS_meta_grp,
4         HM_meta_grp) GENERATE ($0,$2),($0,$2);
5   TSS_HM_meta_flat = FOREACH TSS_HM_meta_crs
6         GENERATE($0,FLATTEN($1));
7   RES_meta = UNION (FOREACH TSS_HM_meta_flat
8         GENERATE NewId($0.$0,$1.$0), FLATTEN($1)),
9         (FOREACH TSS_HM_meta_flat
10        GENERATE NewId($0.$0,S1.$0), FLATTEN($2));
11  TSS_exp_grp = GROUP TSS_exp BY ($0,$1.$0) ;
12  HM_exp_grp = GROUP HM_exp BY ($0,$1.$0);
13  TSS_HM_exp_crs = FOREACH (JOIN TSS_exp_grp BY $0.$1,
14        HM_exp_grp BY $0.$1) GENERATE ($0,$2),($1,$3);
15  DEFINE RES_joiner =
16        GenometricPig.Join('MinDist+GreaterThan',
17              '120000','120000','right');
18  RES_exp = FOREACH(FOREACH TSS_HM_exp_crs
19        GENERATE RES_joiner($1))
20        GENERATE FLATTEN($0);
```

Figure 6.1.2: Translation of a GMQL JOIN into Pig Latin.

provided that such distance is greater than 120K bases[1].

```
RES = JOIN(MINDISTANCE AND DISTANCE > 120000;
               RIGHT) TSS HM;
```

In Fig. 6.1.2, lines 1-10 are concerned with metadata and produce the data bag RES_meta. The metadata of the two operands are first grouped by sample and then the cross product of samples is generated and flattened; each pair in the cross product is associated with a new sample having as metadata the union of the metadata of the two operands.

Lines 11-20 work on regions. We encoded in Java programming language a fast-join algorithm which searches for matching regions at minimal and bound distance. First, samples are grouped and paired (as in the case of metadata). Then, the RES_joiner class is defined as result of invoking the MinDist+GreaterThan Java code; the defined function is invoked on each pair of samples and the result is finally flattened. The linking of metadata and regions of each output sample is guaranteed by the use of the same hash function on the two ids of the input pairs, at lines 8 and 10 for the metadata and within the Join function for the regions.

[1] This query can be used in the search for *enhancers*, i.e., parts of the genome which have an important role in gene activity regulation; the complete example, with a similar query, is in [39], Section 3.2.

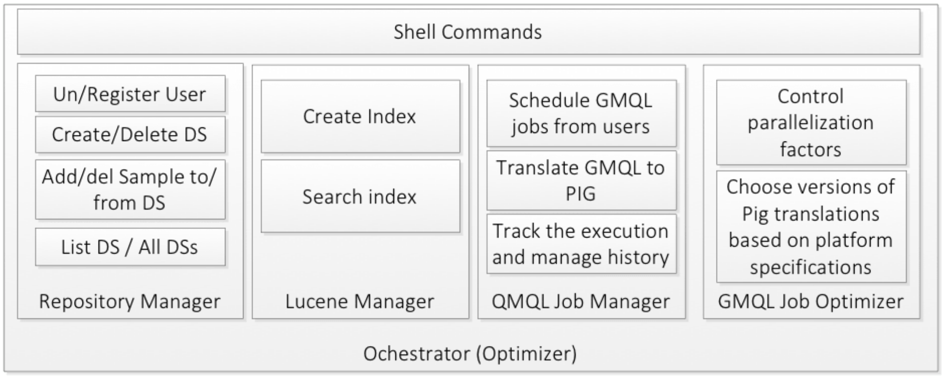


Figure 6.1.3: GMQL V1 orchestrator.

6.1.2 Orchestrator

The orchestrator controls the processing flow of the GMQL code, including compilation, data selection from the repository, scheduling of the efficient execution of Pig Latin code over the *Apache Pig* engine [33], and storing of the resulting datasets in the repository in standard format. The orchestrator has four components: the **Repository Manager**, for registering users and creating, deleting and changing datasets and their samples; the **Index Manager**, for creating and searching metadata indexes; the **GMQL Job Manager**, for launching the GMQL compiler, scheduling GMQL jobs and reporting about the status of GMQL jobs to users; and the **GMQL Job Optimizer**, for controlling the parallelization factors and choosing the version of Pig Latin translator as shown in Fig.6.1.3. When a user submits a GMQL query, the orchestrator uses the job manager to call the GMQL compiler, which produces the query translation into Pig Latin and the search criteria for loading the relevant samples from the repository. Then, the orchestrator uses the index manager to search the index and select the samples that comply to the search criteria, produces a list of the URIs of the samples to be loaded and invokes the job optimizer, which sets the execution parameters (such as the parallelization factors discussed in the next section); eventually, the orchestrator manages the outcome of the computation, including indexing of the result and storage in the user space. The system supports two types of execution, a *Local* mode and a *Map-Reduce* mode; the former one is suggested only for small data sizes, during the setup and debugging of GMQL programs.

The orchestrator can be invoked through different interfaces, including:

- Linux shell commands, each supported by suitable APIs, for managing the repository (adding users, adding/deleting datasets, and adding/deleting samples from existing datasets) and for compiling, running and tracking the execution of GMQL queries.

- RESTful web-services, which use the standard HTTP protocol and JSON files, thereby enabling the access to GMQL from within bioinformatics software and workflow engines, such as Galaxy [66, 67].

6.2 GMQL V2

6.2.1 From V1 to V2

GMQL V1 was a good prototype to prove the functionalities of GMQL. The performance of GMQL V1 was good in comparison to the state of the art applications, (see section 8). However, it showed several several technical and performance limitations, which led to the development of GMQL V2. The most important goal of GMQL V2 is **extensibility**. GMQL V1 was limited in extensibility along with other technical limitations:

- V1 was limited to a single target engine - Pig Latin. Apache Pig started to fade because of the introduction of faster and more flexible engines.
- The only connection between GMQL and Apache Pig was through Pig Server using string queries of pig Latin. This limits our control on the back-end execution and the reporting of errors.
- The ability to connect GMQL to data mining and analysis tools was limited since Pig was developed as a data retrieval and processing engine only.
- Optimizations were limited since we have no control of the output of Pig Latin operations. The best we could do is to run the code in chunks and materialized every output, which is not efficient.
- GMQL V1 was hard to install as it requires several other tools attached to it.

GMQL V1 has other performance limitations:

- Scripts are translated to long Pig Latin scripts which is in turn are translated to long chains of MapReduce jobs, that consume huge time to execute. Our example in section 6.1.1 contains only one GMQL command that is translated to 20 lines of Pig Code and around 16 MapReduce jobs. This makes the execution time of V1 very long in comparison to in typical runs of cloud computing engines.
- Meta operations consume small amount of time to be executed, in comparison to region operations, but this is not exploited in V1. The translator of Meta operations to Pig Latin generates several MapReduce jobs that consume time to execute, which is not efficient for the reserved resources.

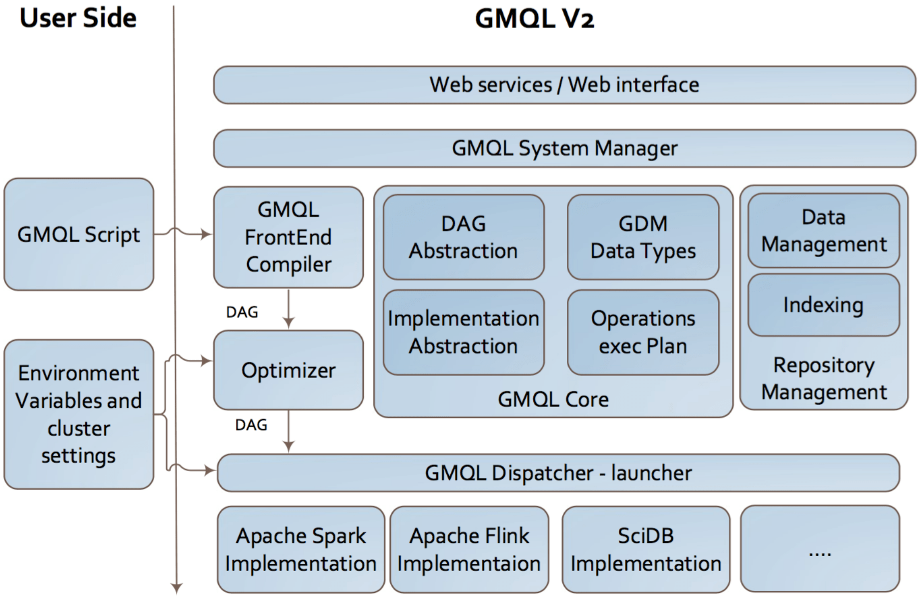


Figure 6.2.1: GMQL V2 Architecture.

GMQL V2 architecture achieved the following advantages over GMQL V1 implementation:

- Flexibility in implementation. We built an intermediate representation of GMQL operations that then translated to the implementing language.

- Optimization. By using GMQL V2, we were able to optimize the execution of GMQL queries by optimizing the Directed Ascyclic Graphs (DAGs) with language-indipendent omizations (such as; performing meta operations before regions operations and then optimize the execution of region operations based on the meta results).

- High performance due to the use of in memory cloud engines such as Apache Flink and Apache Spark. The low level implementations in Flink and Spark results in a small number of stages in comparison to the number of MapReduce jobs generated in GMQL V1. Less stages lead to lower execution time.

- Wide range of libraries that can be used for data analysis and mining. Both Spark and Flink are connected to machine learning libraries that have good performance for machine learning on the cloud.

- The ability to use different data sources and not just HDFS. GMQL V2 architecture was built to be both flexible with the levels of the engine implementation, repository type (file system, No SQL DB, or normal data bases) and repository manager implementation; this extendability is discussed in the next section.

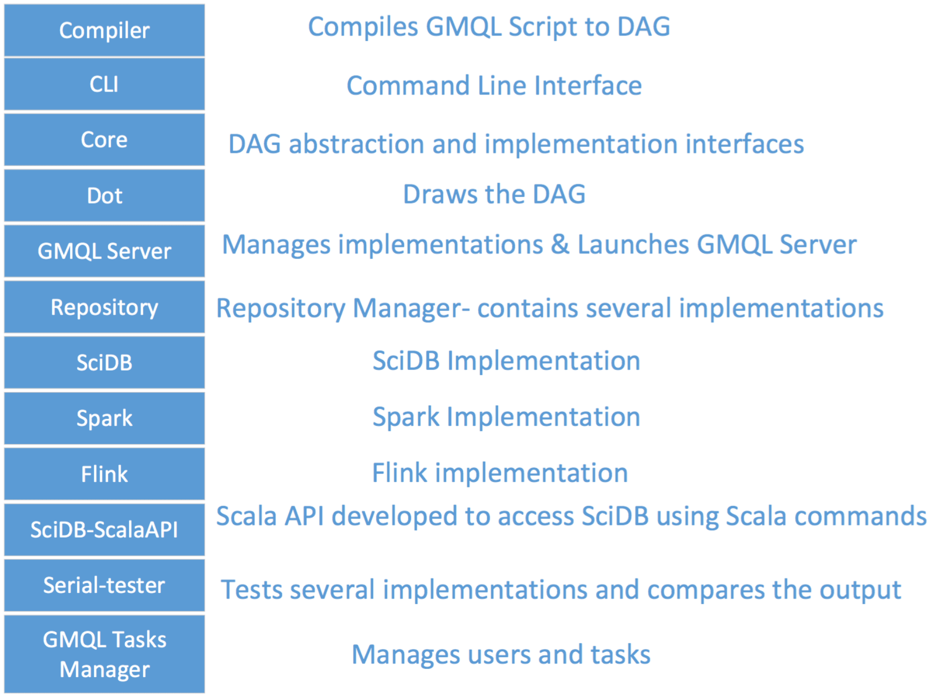

Compiler
Compiles GMQL Script to DAG
CLI
Command Line Interface
Core
DAG abstraction and implementation interfaces
Dot
Draws the DAG
GMQL Server
Manages implementations & Launches GMQL Server
Repository
Repository Manager- contains several implementations
SciDB
SciDB Implementation
Spark
Spark Implementation
Flink
Flink implementation
SciDB-ScalaAPI
Scala API developed to access SciDB using Scala commands
Serial-tester
Tests several implementations and compares the output
GMQL Tasks Manager
Manages users and tasks

Figure 6.2.2: GMQL V2 Modules.

6.2.2 Architecture

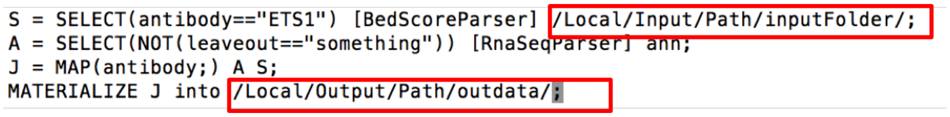
GMQL V2 architecture is shown in Fig.6.2.1. GMQL V2 is coded in Scala 2.10, consists of a set of modules described in Fig.6.2.2. The system receives as input the GMQL queries with environment variables and parameters, such as the choice of the running implementation (Spark, Flink, SciDB) and the deployment mode for the engine and the repository (local, with Yarn on the same machine, or with Yarn on remote machine), and the output files type (GTF [68] or tab delimited).

Several implementations can not co-exist in the same GMQL deployment, because of the conflict of dependencies between the implementation engines, for instance Spark and Flink would uses different version of Netty [69].

GMQL was built as distinct modules and the connections between the modules are minimized as much as possible to make it easy to replace or maintain one module with no effect to other modules (or at least with a traceable effect that can be managed). Thus GMQL can be published in several different versions, every published versions of the engine should include the **Kernel Modules**,i.e. the core module, GMQL Server module, CLI module and one of the implementations modules. Note that GMQL compiler is not a kernel module: GMQL v2.x can run directly by calling the Scala APIs of the DAGs. The five packaging versions are published:

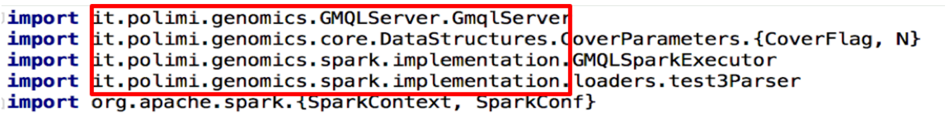
(A) Shell only package

(B) Shell with Repository package



```
S = SELECT(antibody=="ETS1") [BedScoreParser] /Local/Input/Path/inputFolder/;
A = SELECT(NOT(leaveout=="something")) [RnaSeqParser] ann;
J = MAP(antibody;) A S;
MATERIALIZE J into /Local/Output/Path/outdata/;
```

Figure 6.2.3: GMQL script shows how to use GMQL with no repository.



```
import it.polimi.genomics.GMQLServer.GmqlServer
import it.polimi.genomics.core.DataStructures.CoverParameters.{CoverFlag, N}
import it.polimi.genomics.spark.implementation.GMQLSparkExecutor
import it.polimi.genomics.spark.implementation.loaders.test3Parser
import org.apache.spark.{SparkContext, SparkConf}
```

Figure 6.2.4: Importing GMQL packages.

(C) Web Services Package

(D) Web interface Package

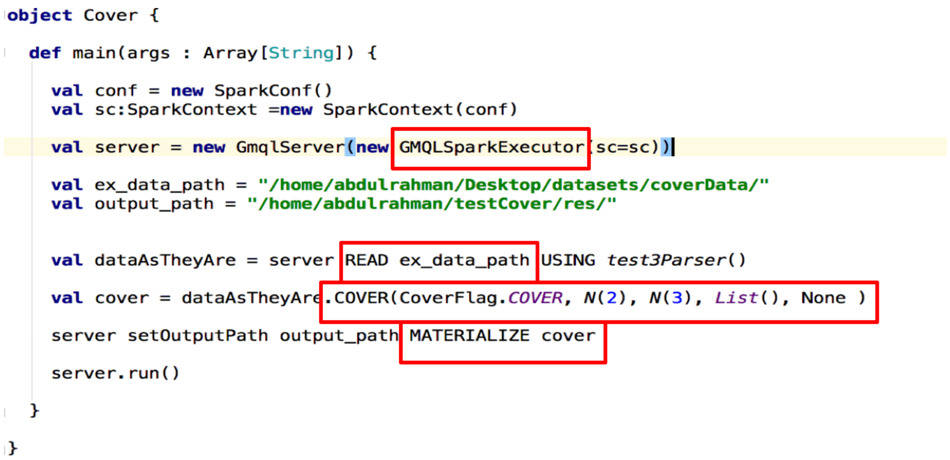
(E) Java API Package

The packages are described in details in the following.

(A) *Shell only package.* This package contains the Kernel modules in addition to the compiler module; it does not contain a repository so there is no record of execution, and no track of users output data. The user specifies the directories of the input and the directories of the output inside GMQL scrip, see Fig.6.2.3. This is the easiest installation package of GMQL but the users should specify the data parser explicitly in GMQL code as shown in Fig.6.2.3.

(B) *Shell with Repository package.* This package is the same as Shell only package with the addition of a repository module. This package supports multiple users and does not force the GMQL script composer to specify the data parser or the location of the files, it is enough to specify the dataset name for input and output. Set of shell commands are added to Command line Interface to manage the repository by showing the users datasets and navigating the datasets samples. Each user has his own work space in addition to the public work space. Samples can be shared between users.

(C) *Web Services Package.* This package is mostly used inside a genomic pipeline such as Galaxy [66, 67]. The full list of the web services is described in section 6.5. This package contains the Kernel Modules along with the compiler module, the repository module, GMQL Task Manager and the web services. The web services are developing the Play framework[70] in Java.



```
object Cover {

  def main(args : Array[String]) {

    val conf = new SparkConf()
    val sc:SparkContext =new SparkContext(conf)

    val server = new GmqlServer(new GMQLSparkExecutor(sc=sc))

    val ex_data_path = "/home/abdulrahman/Desktop/datasets/coverData/"
    val output_path = "/home/abdulrahman/testCover/res/"

    val dataAsTheyAre = server READ ex_data_path USING test3Parser()

    val cover = dataAsTheyAre.COVER(CoverFlag.COVER, N(2), N(3), List(), None )

    server setOutputPath output_path MATERIALIZE cover

    server.run()

  }

}
```

Figure 6.2.5: GMQL example in Scala without GMQL Compiler, direct call to GMQL operations (Cover operation).

(D) *Web interface Package.* This package contains a a web interface that connects to the GMQL system thorough the web services interface using RESTFul web service[71] and the HTTP protocol. The web interface is shown in section 6.6.

(E) *Java/Scala API Package.* GMQL can be called as a set of libraries inside Java or Scala code. The user is responsible to import the Kernel libraries and the compiler if needed, as shown in Fig.6.2.4. By using the API package, the developer can call the GMQL operations directly, without writing a GMQL script and compiling the GMQL script, as shown in Fig.6.2.5. Fig.6.2.5 shows how to setup GMQL environment by creating an instance of GMQL server and selecting the implementation (GMQLSPARKExecutor). All the parameters of the cover operation are found in GMQL Core library.

Dot Module and Serial tester. The Dot module and serial tester module, shown in Fig.6.2.2, are used for **GMQL debugging**. The Dot module uses graph visualization software called Graphviz [72]. GMQL serial tester is shown in Fig.6.2.6. The tester generates data, then runs Spark and Flink implementations one after the other; at the end, it compares the results. This allows us to compare executions and to verify the system after changes that affect each of the implementations. The serial tester also reports the execution time for each implementation, for performance tuning.

GMQL Task Management

The GMQL task manager is the lowest level in Fig. 6.2.2, it takes care of the multi users execution tracking and reporting. GMQL task manager calls the GMQL compiler that compiles GMQL

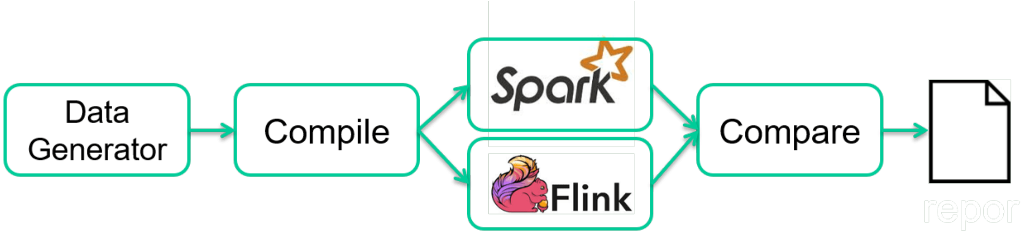


Figure 6.2.6: Comparing Flink and Spark execution engines.

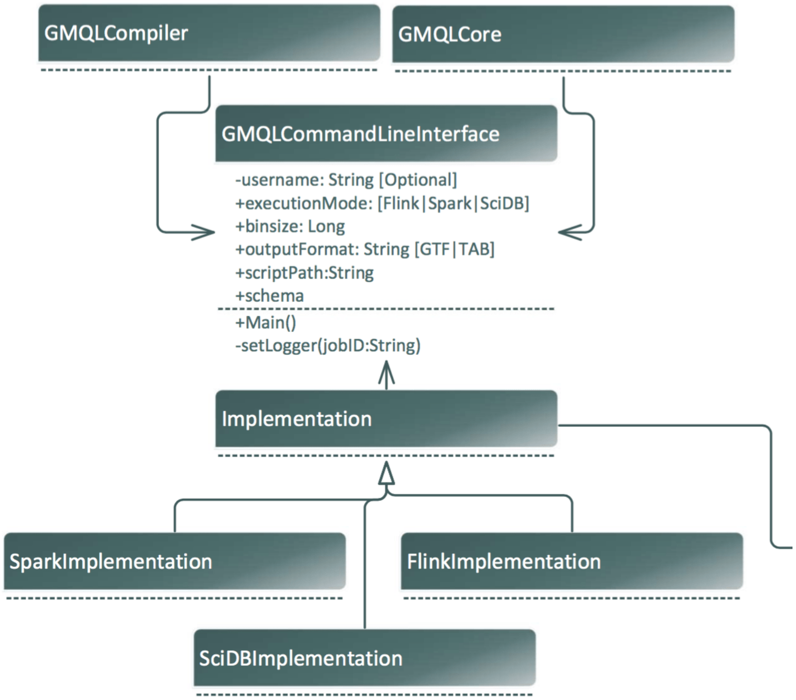


Figure 6.2.7: Task manager

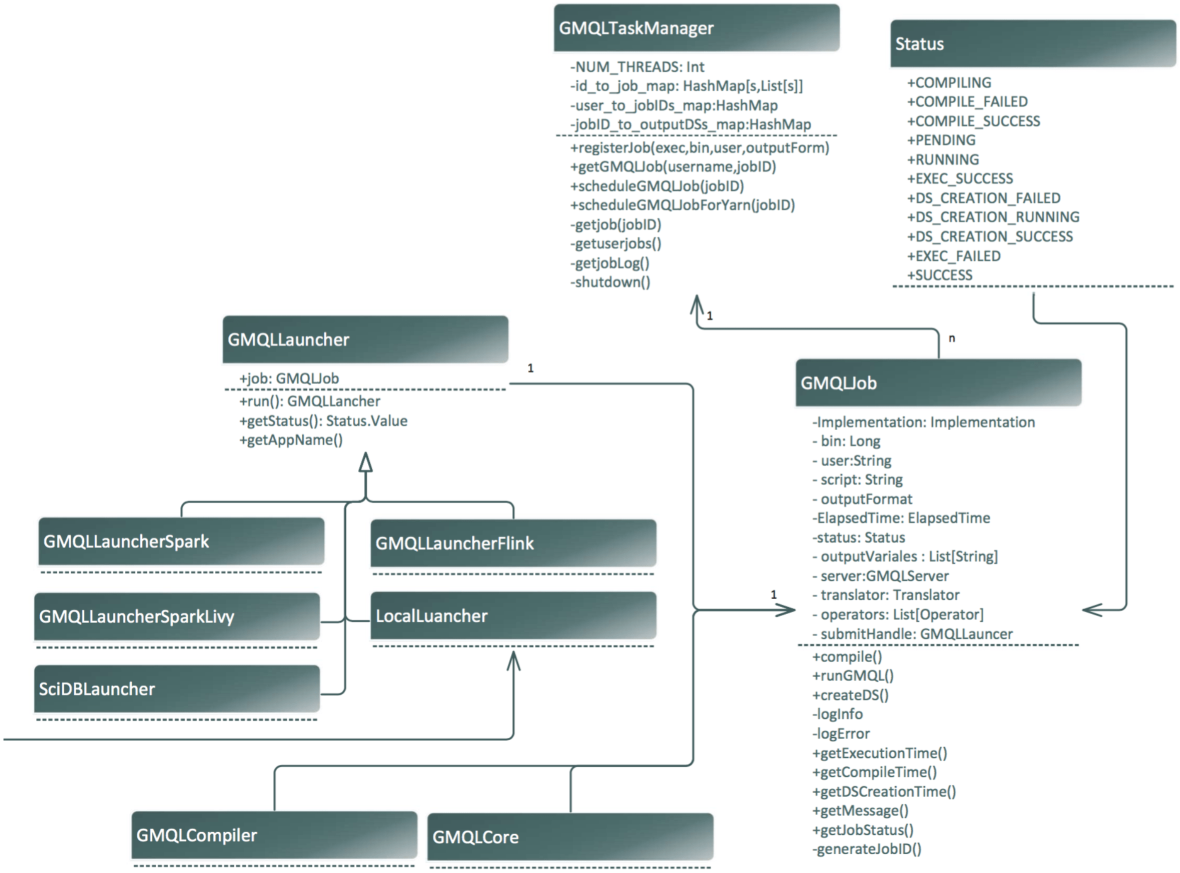


Figure 6.2.8: Task manager

script generating two sets of DAGs (meta and regions), these DAGs then executed by calling the implementation of the DAG nodes. The task manager receives the execution request from the web service and registers the user request after checking the user in the system (GMQLTasManager). Each GMQL script runs a GMQLJob that contains all the information about the job execution, the output datasets, deployment launcher and the repository manager (deployment modes will be discussed in section6.7.) Some of the functionality supported by the task manager are described in the class diagrams shown in Fig.6.2.8 and Fig. 6.2.7. Fig.6.2.8 is connected to Fig.6.2.7 by the link from the Local execution to the implementation class.

Execution of GMQL Queries

DAG operators apply separately to metadata and to regions, hence each GMQL operator is mapped to at least two (one for region and one for meta data) or possibly more DAG operators, as illustrated in Table 6.2.1; due to the language orthogonality, most GMQL operators require the introduction of specific DAG operators; however, the `MetaJoin` and `MetaGroup` clauses of GMQL are highly reused by many different operators as much as some meta data operators like CombineMD.

GMQL Operation	DAG Operators
SELECT	SelectMD, SelectRD, SemiJoinMD
PROJECT	ProjecMD, ProjectRD
EXTEND	ExtendMD, AggregateMD
MERGE	MergeMD, MergeRD
GROUP	GroupMD, GroupRD
ORDER	OrderMD, OrderRD, PurgeRD
UNION	UnionMD, UnionRD
DIFFERENCE	JoinMD, DifferenceRD
COVER	GroupMD, MergeMD, GenometricCoverRD
MAP	JoinMD, CombineMD, GenometricMapRD
JOIN	JoinMD, CombineMD, GenometricJoinRD
STORE	StoreMD, StoreRD

Table 6.2.1: DAG Operators used for each GMQL operations.

The most relevant feature of DAGs is that they illustrate the dependencies between DAG operators; every DAG node includes as parameters the pointers to the DAG nodes that it depends from.

The entire translation of GMQL operations requires 28 DAG nodes in total; each node is implemented in Flink, Spark, and SciDB.

Fig. 6.2.9 shows the DAG constructed for the following query, which includes five `SELECT`s, two `JOIN`s, and one `DIFFERENCE`; in this example, all samples are extracted from global datasets, named `PEAKS` and `ANNOTATIONS`.

```
AC = SELECT(Antibody == 'AcK27') PEAKS;
    ME1 = SELECT(Antibody == 'me1K4') PEAKS;
    ME3 = SELECT(Antibody == 'me3K4') PEAKS;
    GENES = SELECT(Feature == 'genes' AND Prov == 'UCSC') ANNOTATIONS;
    PE = JOIN(DLE(0); CONTIG) AC ME1;
    E = DIFFERENCE() PE ME3;
    AX = JOIN(MD,DLE(100000); LEFT) E GENES;
    R = SELECT(LogFCgene > 1.5 AND LogFCen >1.5) AX;
    MATERIALIZE R;
```

Note that the query variables are either extracted from the repository (in this case the `PEAK` and `ANNOTATION` variables) or defined by operations before being used by other operations, and such precedence relationship determines the edges of DAGs. Note also that region loaders are invoked after the loading of the corresponding metadata, so that they load just the regions of selected samples.

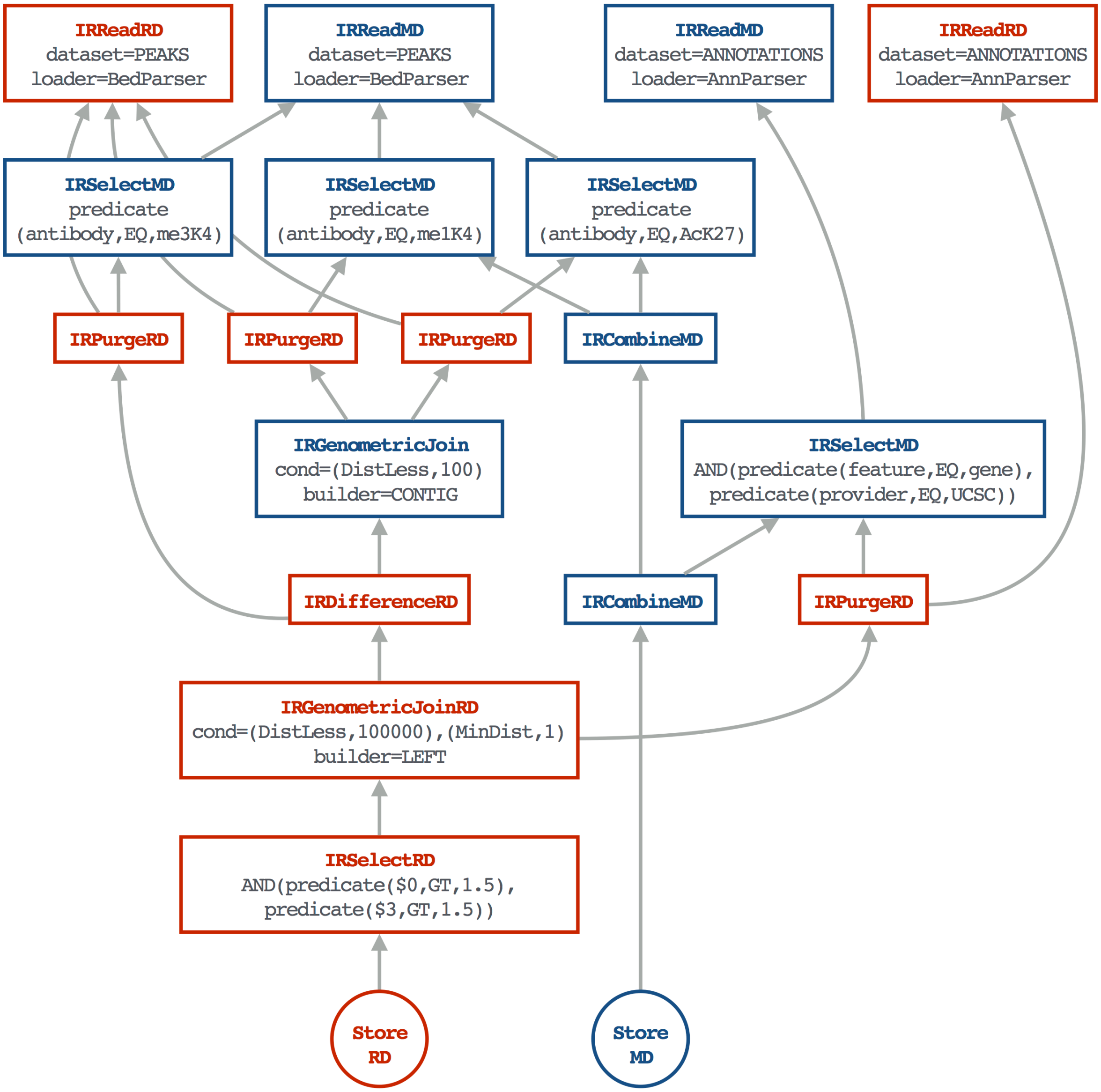


Figure 6.2.9: DAG for a GMQL Query

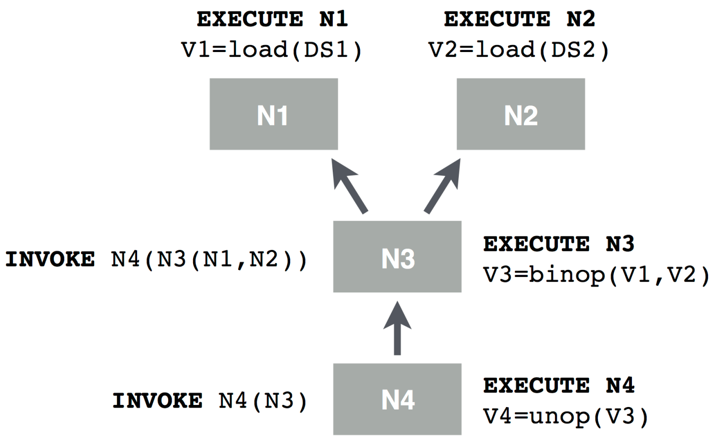


Figure 6.2.10: Recursive descent of the DAG

The DAG execution is triggered by the GMQL operation STORE. Consider queries with a single STORE operation[2]; we denote the set of nodes which are reachable from StoreMD as *"meta-DAG"*. The DAG has two roots, called StoreMD and StoreRD; the translator adds a *dummy root* node which has StoreMD and StoreRD as direct precedences, and then invokes the execution on such dummy root. The execution of any node cannot occur until all the nodes from which it depends are executed, and this induces a partial ordering of node executions. Note that the DAG creates only one node for each query variable, and therefore the operation that computes a variable is executed once, even if a variable is used multiple times in a query. A simple case is illustrated in Fig. 6.2.10: the execution of node $N4$ recursively invokes the execution of $N3$ and then of $N1$ and $N2$; such nodes directly invoke loaders of two datasets $DS1$ and $DS2$. Then, the execution of $N3$ can be concluded, and finally the execution of $N4$ can be concluded.

6.3 Repository management

Data supported by GMQL are organized in datasets, they are collections of homogeneous biological samples. From the system's viewpoint, each dataset corresponds to a variable in the GMQL language and each sample in the dataset corresponds to two files (sample file and meta file). For tertiary analysis, our interest is centered on results of dnaseq, rnaseq or chipseq experiments, thus the input file formats that we consider are standard bed files, narrow peaks, big peaks, bedgraph, VCF, GTF, or any tab delimited format. The objectives of GMQL repository are:

- Ease of use: the user should register the files to GMQL with an operation demanding no technical skill and no overhead.

- Privacy protection: each user should have a private space where his data are initially loaded; each user can designate data resulting from GMQL operations as persistent, and these are added to his private space. In addition, users can be part of a group which have shared access to the group datasets.

- Read-only access to public data: each user should, in addition to her private data, have access to public data, which however cannot be manipulated by individual users.

- Transparency: the user should not to be aware of how files are managed by the system.

- Extensibility: the architecture must be easy to install and maintain, e.g. due to the addition of new data types.

Two GMQL Repository versions have been developed:

[2] When a DAGs has multiple STORE operations, at least one of them does not depend on any stored variable; this induces a partial order of materializations.

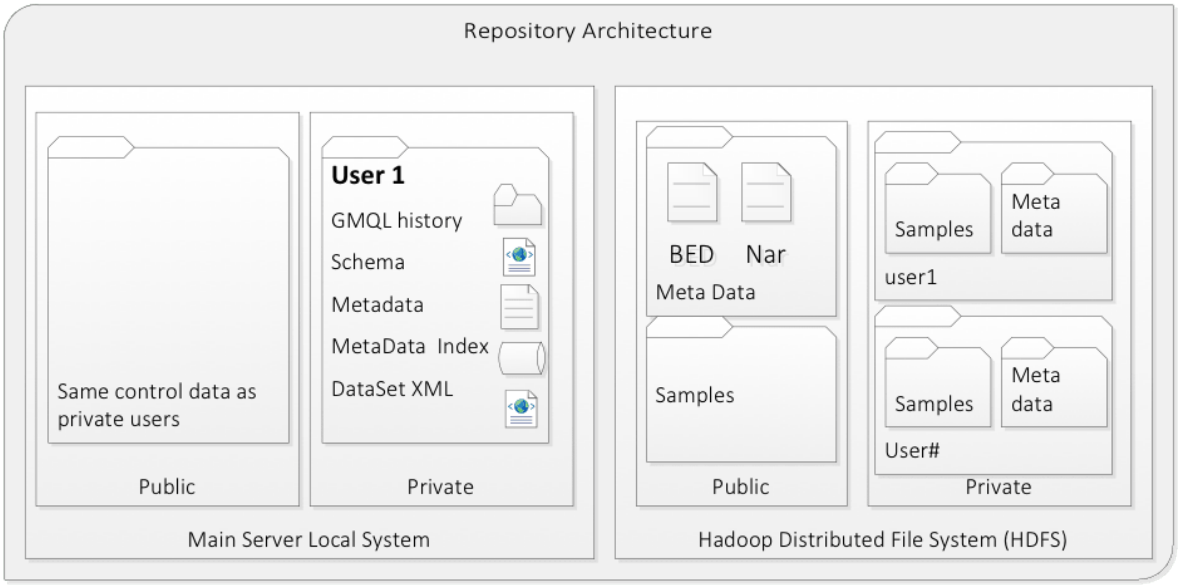


Figure 6.3.1: Repository V1 architecture.

- GMQL repository V1, shown in Fig.6.3.1, includes only a **Local File System** (LFS), organized within the Linux file system of the master node of the computing framework, and an **Hadoop Distributed File System** (HDFS) [29], shared among all the computing nodes. GMQL repository V1 is tightly connected to both LFS and HDFS and can not work with any other storage system or file system.

- GMQL repository V2 uses the structure of V1 for storage in LFS, but without the dependency on HDFS, as in V2 repository we can use any file system or data base. The class diagram structure of the GMQL repository is shown in Fig. 6.3.2.

Datasets are subdivided in metadata and region data; *Apache Lucene* [34] indexes metadata. Both file systems have a public and a private space. Besides the genomic data, the LFS stores system-controlled information, encoded in XML, about the registered users, their security control and privileges, their saved queries and the location of their private resources[3].

In both V1 and V2 repositories, all the datasets are stored in their original text format, as usually these files must be concurrently available to users for other computations. The datasets that are selected by a GMQL query are serialized by suitable adapters and translated to the internal binary GDM format on demand, so as to be loaded in the engine before query execution; at that point, they can be managed by GMQL engine. In this way, we **do not replicate data** in the native and GDM

[3] In the system installation at IEO-IIT (https://www.ieo.it/en/ http://genomics.iit.it/), a center of excellence in oncology research,we connected the repository to a Laboratory Information Management System (LIMS) designed for storing both the raw data after NGS and the workflows for producing processed data into the HDFS [73].

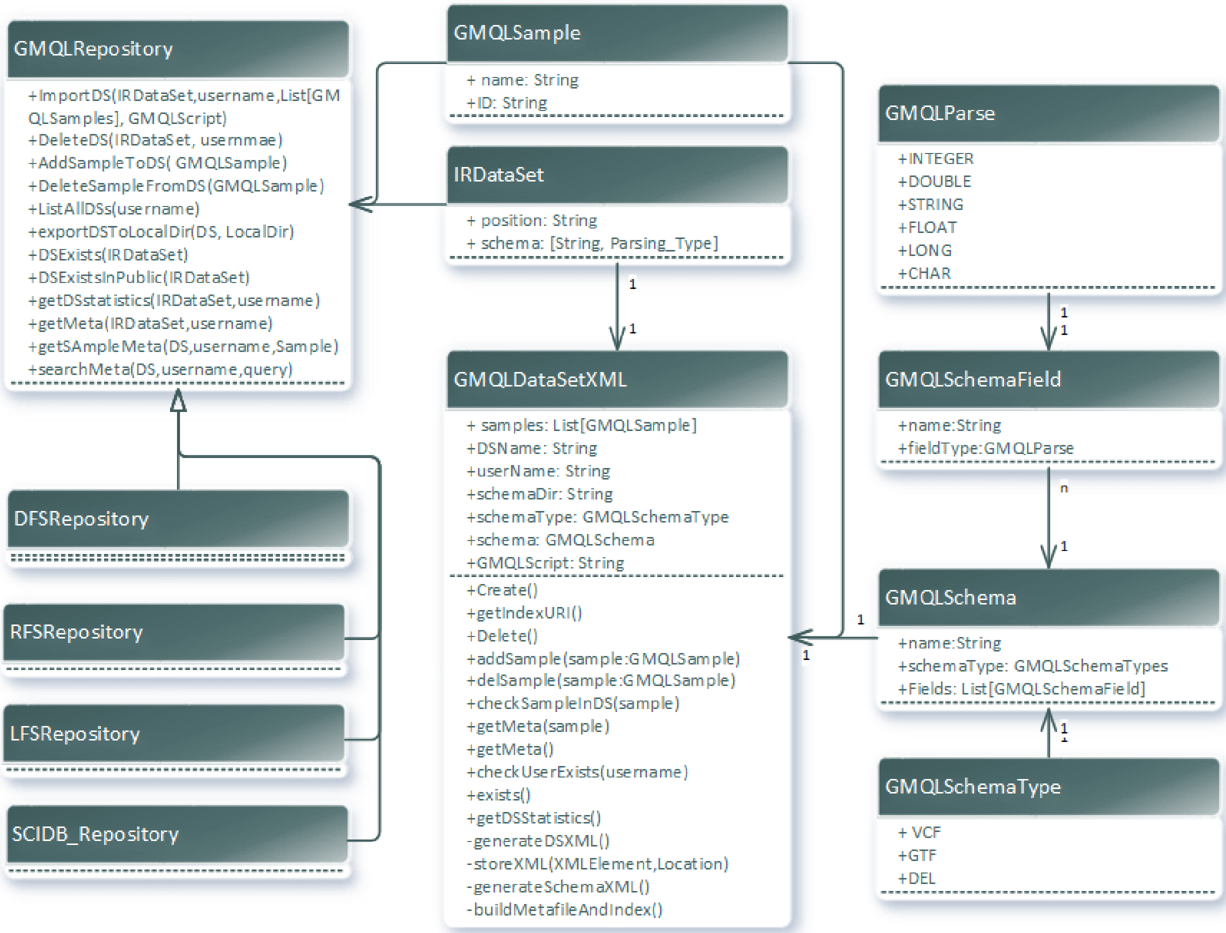


Figure 6.3.2: Repository V2 architecture.

formats and we minimize data translations from native into GDM format.

Datasets in GMQL repository are represented by an XML file, that keeps information about:

- The dataset files names (samples' names) along with the IDs.
- Repository type (LFS, HDFS, Remote File System-RFS, SCIDB, or DB).
- Location of the GMQL script that generated this dataset (if any).

```
<dataset name="HG19_ENCODE_BED", Location = "RFS",
  GMQL_SCRIPT="/home/…./example1.GMQL",
  schemaDir="/to/path /global.schemata">
    <url id=1>"/to/path /myBed.bed"</url>
    <url id=2>"/to/path /myOtherBed.bed"</url>
    …
    …
</dataset>
```

The schema file is provided in a XML file where each field has a data type and a name, the line order of the fields should be the same as the positions of the fields in the sample file:

```
<?xml version="1.0" encoding="UTF-8"?>
<gqlSchemaCollection name="GMQL_SCHEMA">
 <gqlSchema name="HG19_ENCODE_BED" type="bed">
        <field type="STRING">Chromosome</field>
        <field type="LONG">start</field>
        <field type="LONG">stop</field>
        <field type="STRING">name</field>
        <field type="CHAR">strand</field>
        <field type="FLOAT">score</field>
 </gqlSchema>
</gqlSchemaCollection>
```

Fig.6.3.2 shows that dataset and sample for GMQL are represented by an intermediate representation **IRdataSet, GMQLSample** classes. By using an intermediate representation for the dataset in GMQL engine, we make it possible to change the physical representation of the dataset from being an XML files stored locally records in a database that contains users datasets and permissions, with out affecting the system functionality. A simple implementation of the dataset XML files (GMQL-DataSetXML class) to simplifies the installation of the system (no additional tools to install or configure connections) this representation is effective in a distributed processing system like the GMQL engine.

In Fig.6.3.2 we can see the types that the system can parse (Integer, float, double, string and char) we already have parsers for the following text files: tab delimited format (DEL), general transfer format (GTF) [68], and variant call format (VCF) [74].

The GMQL repository interface already have four implementations; LFS, HDFS, RFS and SCIDB, and can be expended so as to support other implementations (e.g. data bases such as MngoDB [75] or any other NO-SQL engines).

6.3.1 Integrated Access to heterogeneous Public Repositories

Very large-scale sequencing projects are emerging; as of today, the most relevant ones include:

- The Encyclopedia of DNA elements (ENCODE) [22], the most general and relevant worldwide repository for basic biology research. It provides public access to more than 4,000 experimental datasets, including the just released data from its Phase 3, which comprise hundreds of epigenetic experiments of processed data in human and mouse;

- The Cancer Genome Atlas (TCGA) [42], a full-scale effort to explore the entire spectrum of genomic changes involved in human cancer;

- The 1000 Genomes Project [76], aiming at establishing an extensive catalogue of human genomic variations from 26 different populations around the globe;

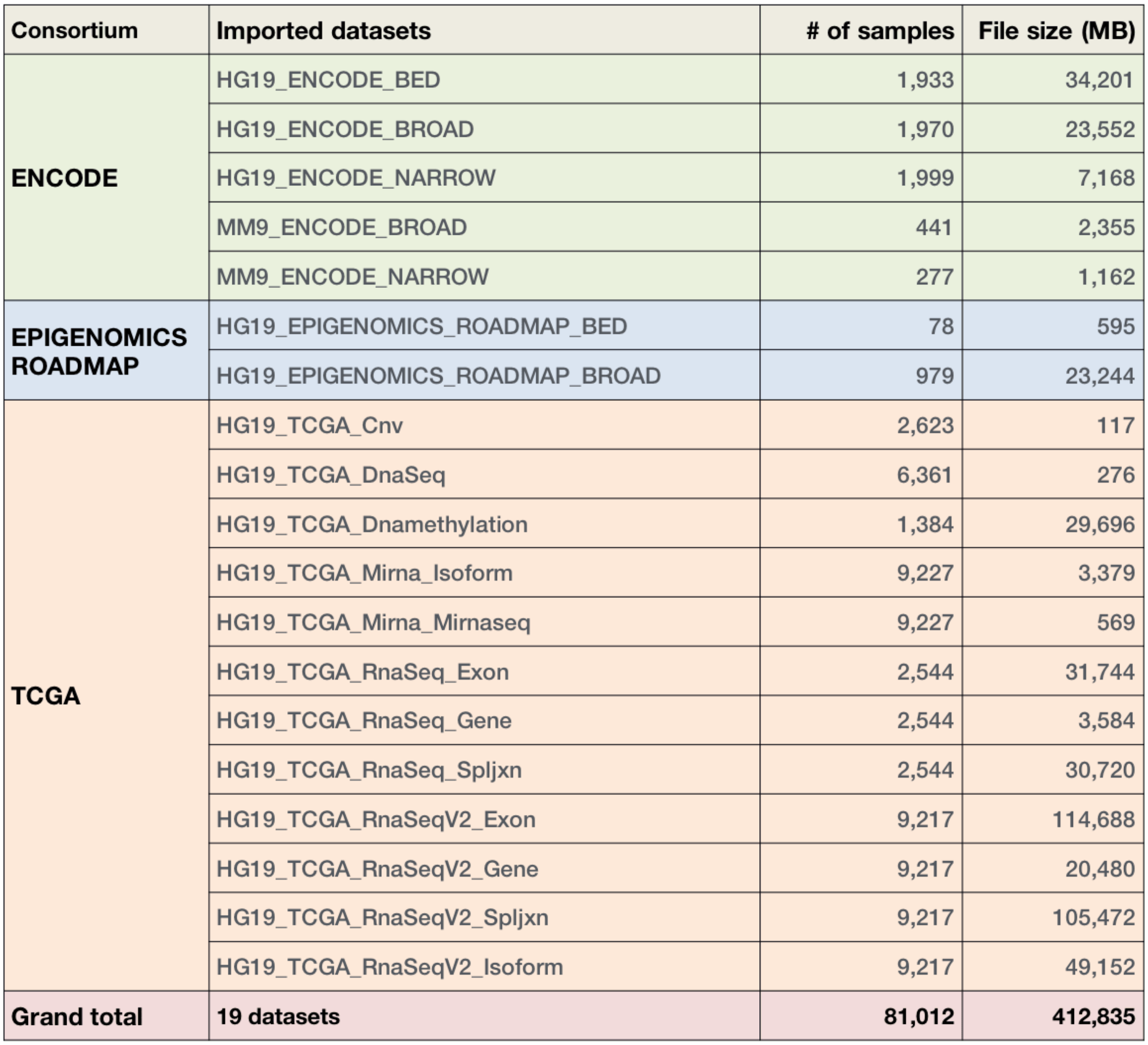
- The Epigenomic Roadmap Project [77], a repository of "normal" (not involved in diseases) human epigenomic data from NGS processing of stem cells and primary ex vivo tissues.

Our data repository contains thousands of files and almost half a TeraByte of data of public datasets, as shown in 6.3.3.

6.4 Retrieval system - the use of an inverted index

GMQL is a retrieval and processing engine. The simplest way to perform data retrieval, using cloud computing technologies, is to load the dataset and then *filter the samples*. The filtering condition is either a region condition applied to the sample regions, or meta condition applied to the meta data. To implement the retrieval system, we perform *region selection* by filtering regions data (samples files) based on the regions condition, and we also perform *meta selection* by filtering the meta data based on the meta condition. The meta data filtering is applied only after downloading the data, so it is not effective in producing the subset of samples IDs that are needed for processing (see Fig.6.3.3). This filtering operation consumes a considerable execution time and blocks the resources.

For this reason, we decided to use Lucene in order to index the metadata. We perform the retrieval in GMQL by:



Consortium	Imported datasets	# of samples	File size (MB)
ENCODE	HG19_ENCODE_BED	1,933	34,201
	HG19_ENCODE_BROAD	1,970	23,552
	HG19_ENCODE_NARROW	1,999	7,168
	MM9_ENCODE_BROAD	441	2,355
	MM9_ENCODE_NARROW	277	1,162
EPIGENOMICS ROADMAP	HG19_EPIGENOMICS_ROADMAP_BED	78	595
	HG19_EPIGENOMICS_ROADMAP_BROAD	979	23,244
TCGA	HG19_TCGA_Cnv	2,623	117
	HG19_TCGA_DnaSeq	6,361	276
	HG19_TCGA_Dnamethylation	1,384	29,696
	HG19_TCGA_Mirna_Isoform	9,227	3,379
	HG19_TCGA_Mirna_Mirnaseq	9,227	569
	HG19_TCGA_RnaSeq_Exon	2,544	31,744
	HG19_TCGA_RnaSeq_Gene	2,544	3,584
	HG19_TCGA_RnaSeq_Spljxn	2,544	30,720
	HG19_TCGA_RnaSeqV2_Exon	9,217	114,688
	HG19_TCGA_RnaSeqV2_Gene	9,217	20,480
	HG19_TCGA_RnaSeqV2_Spljxn	9,217	105,472
	HG19_TCGA_RnaSeqV2_Isoform	9,217	49,152
Grand total	**19 datasets**	**81,012**	**412,835**

Figure 6.3.3: Repository data.

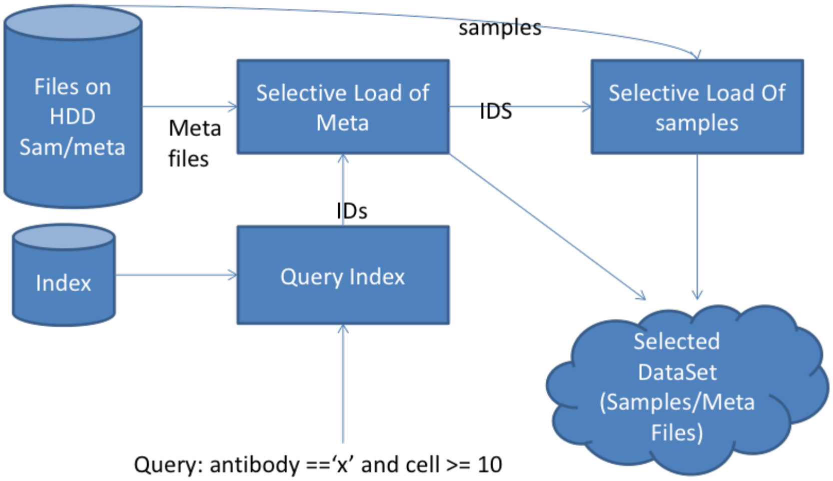

samples
Files on HDD Sam/meta
Meta files
Selective Load of Meta
IDS
Selective Load Of samples
IDs
Index
Query Index
Selected DataSet (Samples/Meta Files)
Query: antibody =='x' and cell >= 10

Figure 6.4.1: Retrieval system with the selective load and Lucene index.

- First, query the dataset's Lucene index, which results in a list of samples IDs.
- Then, use these IDs to **selectively load** the samples files needed for processing.
- Selection on regions is more efficient by pushing the regions filtering to the load phase, that loads only the regions needed for processing.

When adding a dataset to the repository, the Lucene [34] index for the added dataset meta files is built on the fly. Usually indexing the meta data does not consume much time since the meta data is usually small in size in comparison to the sample files. For example, the total datasets meta size of Narrow Peaks, Broad peaks, and RnaSeqV2 Genes are respectively 5.4MB, 6MB, and 50MB, while the number of samples in each is respectively 1999, 1970, and 9217). The index is built once for each dataset. All the generated datasets from GMQL has its **Lucene index built in parallel** for its meta data. The use of Lucene index decrease the meta selection time from over two minutes for selection on RnaSeqV2 Genes dataset to **only 300 milliseconds**, which equals to 400X speed up.

In order to see the the performance improvement of using the selective load and Lucene index over the traditional filtering, we did a test on the narrow peaks dataset shown in Fig.6.3.3. The performance of the selection shown in table 6.4.1. The sizes of inputs and outputs is shown in Fig.6.4.2.

6.5 Web Services

While simple and efficient, the command line interface provided by the GMQL it is not flexible and complete enough to provide a satisfactory user experience. For instance, the user must manually load its own files on the server, or find out where GQML saves the results. Even worse, there

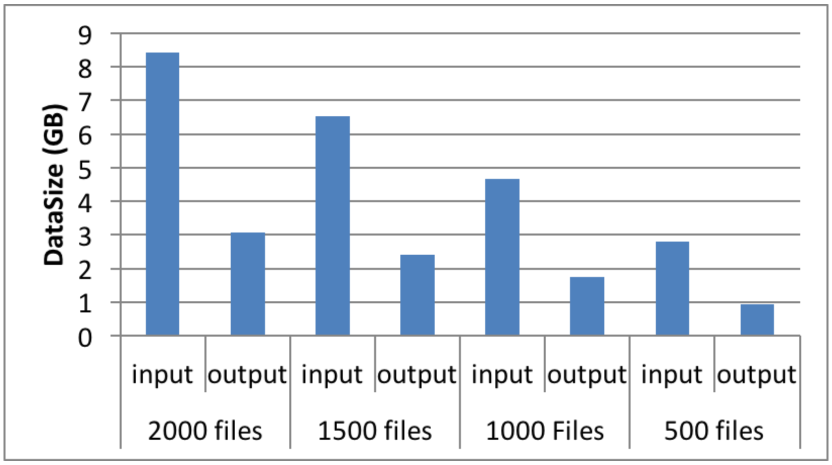


Figure 6.4.2: Select input and output data sizes.

Dataset (# of samples)	Complete Load (sec)	Selective Load with Lucene (sec)
2000	150	0.315
1500	125	0.315
1000	104	0.315
500	80	0.315

Table 6.4.1: Selection performance with Selective load and Lucene.

is nothing that helps to write queries; in particular, composing the select predicates is particularly challenging without knowing in advance all the meta-data.

We decided to build on the top of the system a set of web services to be able to manage the data in a much simpler way. We designed them as RESTFul that uses HTTP protocol[78].

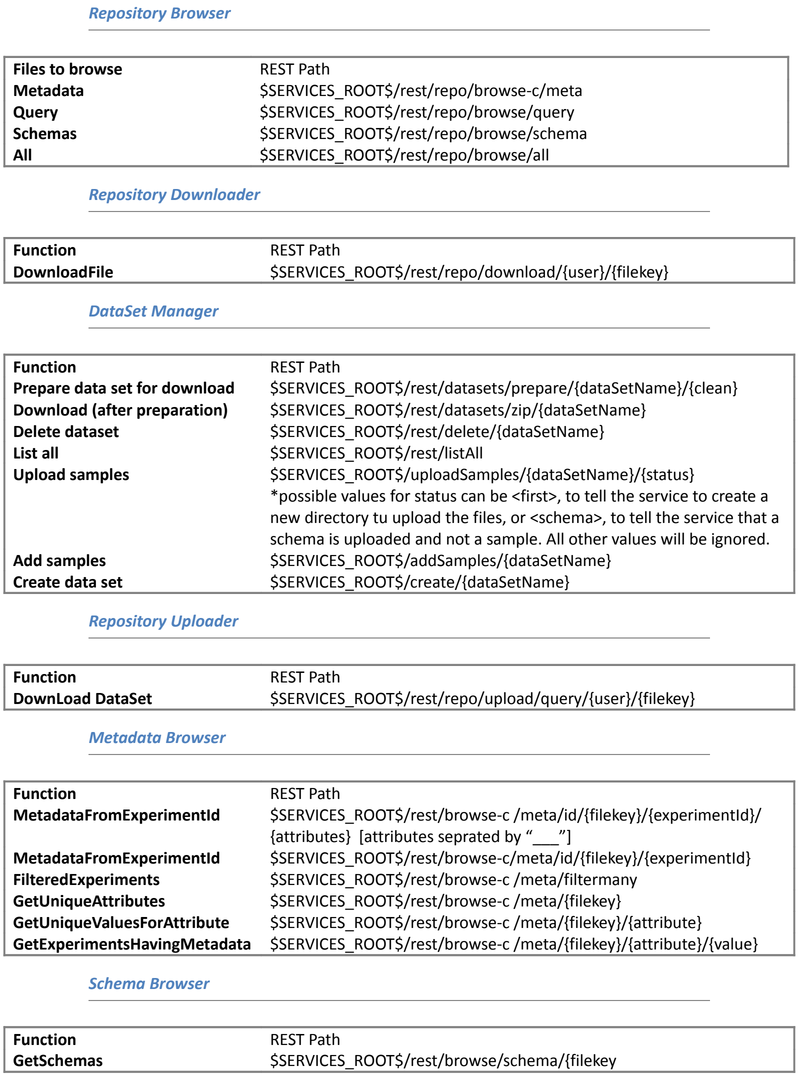
The advantage of exposing Web Services relies on the fact that multiple applications can use them without the need to worry about the specific implementation or the necessity to access directly the elements of a system. Moreover, the use of web-services to access cloud-computing resources is by now a common approach [79].

Between the various technologies, we adopted the Representational State Transfer (REST) architecture style [71]. RESTful systems communicate very easily over the Hypertext Transfer Protocol with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) used by web browsers to retrieve web pages and send data to remote servers.

The complete list of web services we expose is shown in figure 6.5.1. In synthesis, the operations that can be performed on the system are:

- Upload of files. This includes meta-data and samples, queries, and schema xml files.

- Run queries and track their progress.



Repository Browser

Files to browse	REST Path
Metadata	$SERVICES_ROOT$/rest/repo/browse-c/meta
Query	$SERVICES_ROOT$/rest/repo/browse/query
Schemas	$SERVICES_ROOT$/rest/repo/browse/schema
All	$SERVICES_ROOT$/rest/repo/browse/all

Repository Downloader

Function	REST Path
DownloadFile	$SERVICES_ROOT$/rest/repo/download/{user}/{filekey}

DataSet Manager

Function	REST Path
Prepare data set for download	$SERVICES_ROOT$/rest/datasets/prepare/{dataSetName}/{clean}
Download (after preparation)	$SERVICES_ROOT$/rest/datasets/zip/{dataSetName}
Delete dataset	$SERVICES_ROOT$/rest/delete/{dataSetName}
List all	$SERVICES_ROOT$/rest/listAll
Upload samples	$SERVICES_ROOT$/uploadSamples/{dataSetName}/{status} *possible values for status can be <first>, to tell the service to create a new directory tu upload the files, or <schema>, to tell the service that a schema is uploaded and not a sample. All other values will be ignored.
Add samples	$SERVICES_ROOT$/addSamples/{dataSetName}
Create data set	$SERVICES_ROOT$/create/{dataSetName}

Repository Uploader

Function	REST Path
DownLoad DataSet	$SERVICES_ROOT$/rest/repo/upload/query/{user}/{filekey}

Metadata Browser

Function	REST Path
MetadataFromExperimentId	$SERVICES_ROOT$/rest/browse-c /meta/id/{filekey}/{experimentId}/{attributes} [attributes seprated by "___"]
MetadataFromExperimentId	$SERVICES_ROOT$/rest/browse-c/meta/id/{filekey}/{experimentId}
FilteredExperiments	$SERVICES_ROOT$/rest/browse-c /meta/filtermany
GetUniqueAttributes	$SERVICES_ROOT$/rest/browse-c /meta/{filekey}
GetUniqueValuesForAttribute	$SERVICES_ROOT$/rest/browse-c /meta/{filekey}/{attribute}
GetExperimentsHavingMetadata	$SERVICES_ROOT$/rest/browse-c /meta/{filekey}/{attribute}/{value}

Schema Browser

Function	REST Path
GetSchemas	$SERVICES_ROOT$/rest/browse/schema/{filekey

Figure 6.5.1: List of the Web Services exposed by the system

- Download of files. Datasets can be downloaded after being prepared and compressed in a zip file.
- Browse queries, meta-data files, schema files.
- Browse meta-data in details: it is possible to retrieve, for each data-set, the meta-data of all the samples that respect some simple query on their meta-data.

The services are built on top of the Orchestrator, are coded in Java and run on a Tomcat7 server [80]; they require the user to log in, thus is possible to add new users by simply registering them on Tomcat [4].

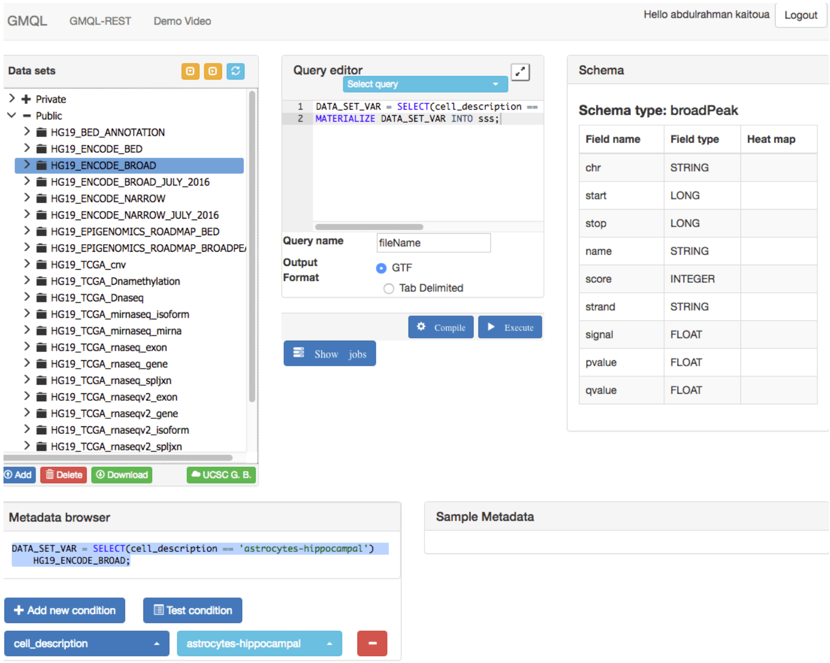
6.6 Web Interface

Fig.6.6.1 shows the latest version of the web interface of GMQL. This web interface is built in Scala using the Play Framework [70] . A previous version of GMQL had the same components but in separated pages and it was less user friendly.

All the connections between GMQL and the web interface are handled by a request call to the web services using HTTP protocol and a replay in a JSON file. The web interface uses javascript, JQuery and AJAX technologies. The web interface consists of:

- Query editor where the user can enter the GMQL script and run the Job.
- The dataset browser: shows the private and public datasets. In the private space, the user can show samples, add datasets or samples, delete datasets or samples, update datasets, and download datasets.
- The schema browser: shows the schema of a dataset. When a user click on a dataset in the dataset browser, the schema browser get updated automatically.
- Meta Browser: allows the user to navigate the meta data so as to progressively create the selection statement. The meta browser shows the result of the selection that is being built of predicates generated using a drop-down list selection.
- Job tracker: Shows all the jobs that this user is running or ran in the past.
- UCSC browsing: by clicking on the browsing button the system will send the regions of a dataset to UCSC browser for viewing using the APIs provided by the browser.

[4] Tomcat users will not be able to use the command line or access to the server outside the interface, providing a simple but effective layer of security



- Private
- Public
  - HG19_BED_ANNOTATION
  - HG19_ENCODE_BED
  - HG19_ENCODE_BROAD
  - HG19_ENCODE_BROAD_JULY_2016
  - HG19_ENCODE_NARROW
  - HG19_ENCODE_NARROW_JULY_2016
  - HG19_EPIGENOMICS_ROADMAP_BED
  - HG19_EPIGENOMICS_ROADMAP_BROADPE
  - HG19_TCGA_cnv
  - HG19_TCGA_Dnamethylation
  - HG19_TCGA_Dnaseq
  - HG19_TCGA_mirnaseq_isoform
  - HG19_TCGA_mirnaseq_mirna
  - HG19_TCGA_rnaseq_exon
  - HG19_TCGA_rnaseq_gene
  - HG19_TCGA_rnaseq_spljxn
  - HG19_TCGA_rnaseqv2_exon
  - HG19_TCGA_rnaseqv2_gene
  - HG19_TCGA_rnaseqv2_isoform
  - HG19_TCGA_rnaseqv2_spljxn

```
DATA_SET_VAR = SELECT(cell_description ==
MATERIALIZE DATA_SET_VAR INTO sss;
```

Query name: fileName

Output Format: ● GTF ○ Tab Delimited

Schema type: broadPeak

Field name	Field type	Heat map
chr	STRING	
start	LONG	
stop	LONG	
name	STRING	
score	INTEGER	
strand	STRING	
signal	FLOAT	
pvalue	FLOAT	
qvalue	FLOAT	

```
DATA_SET_VAR = SELECT(cell_description == 'astrocytes-hippocampal')
    HG19_ENCODE_BROAD;
```

cell_description — astrocytes-hippocampal

Figure 6.6.1: GMQL V2 Web interface

6.7 GMQL V2 Deployment modes

As we have seen that GMQL v2 engine can have several implementation of GMQL using different big data engines such as Pig, Flink, spark or SciDB. By having GMQL repository being able to connect to different file systems or databases, we have almost many to many relationship in the choose of the implementing engine and the repository type. Spark and Flink can connect to LFS, HDFS, DB, MongoDB and so on, while SCIDB connects to a SciDB repository, as shown in Fig.6.7.1.

In the SciDB deployment a master GMQL machine is always connected to a remote SciDB cluster, thus the following discussion applies only on Flink and Spark; we will take Spark as an example.

GMQL deployment modes are *Local* and *Remote* deployment modes; in the GMQL Local mode, GMQL engine is installed on the master node of Hadoop and Spark cluster. While in the GMQL Remote mode, GMQL engine is installed on a separated machine (GMQL master machine) connected remotely to Hadoop and Spark cluster master node. We describe these modes in details in the following subsections.

6.7.1 Spark Deployment

In order to have good understanding of GMQL deployment modes, we show the deployment modes of the underlying cloud computing engine (Spark in this case). Spark has several deployment

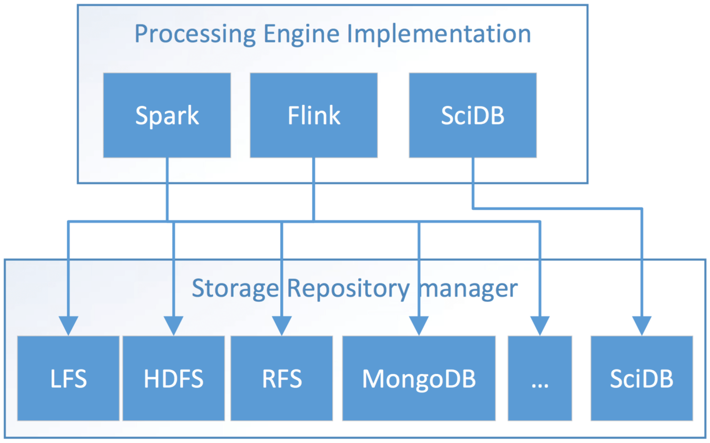


Figure 6.7.1: GMQL System deployment, the selection of engine and repository manager implementations.

modes, here we mention the most relevant ones:

- *Single Process deployment*: In operating systems, applications run in a separated processes and each process can run several threads inside. The simplest Spark deployment is to run Spark application with Spark engine in a single system process. In this deployment mode the Spark Context class, creates thread for every component of Spark (job tracker and task trackers). Running Spark Application pragmatically (call spark from inside a scala/java code) will create a single process deployment. In this deployment mode Spark can read/write from LFS, since it is in a single machine.

- *Standalone Deployment*: in this mode we have to install a compiled version of Spark on each node on the cluster. Spark cluster can be a single machine or more. To run Spark on a cluster of more than one machine, Spark uses a distributed storage such as HDFS, see Fig.6.7.2. In this mode Spark controls resources management over the cluster.

- *Spark over Yarn*: Spark is installed on the master node of Yarn cluster only. Spark submit the applications to Yarn, while Yarn control the resource management, see Fig.6.7.2. In this mode, Yarn serves as an Operating System, manages resources between several cloud computing engines including Spark.

6.7.2 GMQL Deployment on a Single Machine or on local cluster

Spark has several deployment modes, leaded to more flexibility in deploying GMQL engine.

- GMQL as a Single Java/Scala Application: GMQL uses *Spark Single process* deployment mode, mentioned in the previous subsection. In this mode, GMQL Java application runs the Spark

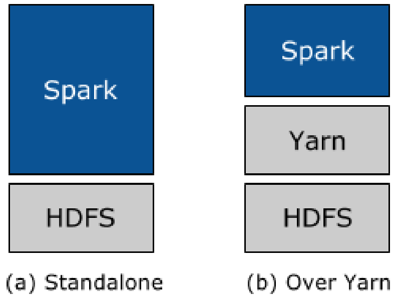


Figure 6.7.2: Spark deployment.

implementation programmatically. In this mode, the GMQL application is a single JAR file (java ARchive), that is executed as normal Java application using Java shell command. This deployment is good in case of having single server and we have light GMQL queries.

- GMQL Over Yarn: Here, we use *Spark over YARN* deployment mode from Spark, while GMQL application is submitted to Spark using Spark-Submit shell command or using it Spark Launcher Server (mentioned bellow), which was introduced in Spark 1.5.1 and later versions. In this deployment mode GMQL Application JAR will be loaded in YARN, resources will be reserved in YARN and finally the application will run; This process takes from 30 to 90 seconds delay before running GMQL application, which is not suitable for light GMQL queries. This deployment is good when we have more than one machine for GMQL and GMQL application runs on the master machine of the cluster.

GMQL Spark deployment can skip HDFS and run on LFS (Local File System) in case of a single machine installation with local process or single node cluster running. This installation is good for small data size and for debugging.

6.7.3 Spark Launcher Server

The Spark launcher server (L. Server) listens locally for connections from client launched by the library. Each client has a secret that it needs to send to the server to identify itself and establish the session. Clients have a limited time to connect back to the server, otherwise the server will ignore the connection. The architecture of Spark launcher server is shown in Fig.6.7.3.

The launcher server is used when Spark apps are launched as separate processes than the calling app. It looks more or less like the following: The server is started on demand and remains active while there are active or outstanding clients, to avoid opening too many ports when multiple clients are launched. Each client is given a unique secret, and have a limited amount of time to connect back (SparkLauncher#CHILD_CONNECTION_TIMEOUT), at which point the server will throw away that client's state. A client is only allowed to connect back to the server once.

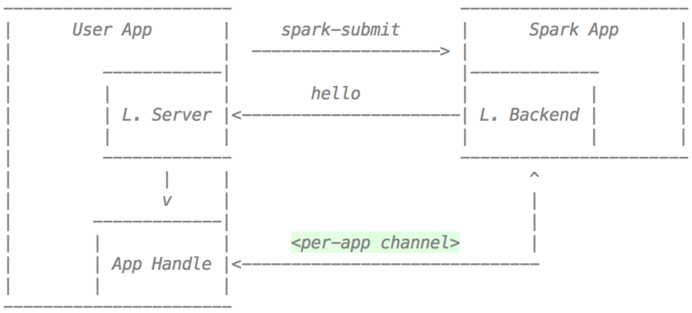


Figure 6.7.3: Spark launcher Server Architecture, available in Spark 1.5.1 and later.

The launcher server listens on the localhost only, so it doesn't need access controls (aside from the per-app secret) nor encryption. It thus requires that the launched app has a local process that communicates with the server. In cluster mode, this means that the client that launches the application must remain alive for the duration of the application (or until the app handle is disconnected).

6.7.4 Deployment on Remote Cluster

This type of deployment was used when we installed GMQL in CINECA cluster[81]. The deployment consists of a master node that holds GMQL Application and a cluster of Hadoop machines, shown in Fig.6.7.4.

We use Livy[48], described in section5.3, to connect to the remote cluster of Spark as shown in Fig.6.7.4; we call GMQL launcher that uses Livy, *SparkLivyLauncher*, as shown in the class diagram of the server manager in Fig.6.2.8. Livy uses web service client to connect to Livy server (Livy server is installed on Hadoop cluster master node). SparkLivyLauncher launches GMQL job and ping remote Livy server for the status of the job and when the job ends with either success or fail, finally SparkLivyLauncher request the logs to show to the GMQL user.

Spark runs on Hadoop remotely, in order to connect to HDFS on the remote cluster, we use Knox server[47], described in sub-section5.2, that allows us to move data in/out the remote HDFS; we created an implementation of the repository that uses Knox, called *Remote Reposiotry Manager (RFS)*, as shown in the class diagram of the repository in Fig.6.3.2. All file system operations of listing files of a directory, copying file, or moving files are handled using a POST, GET, and CREATE commands of HTTP protocol though Knox.

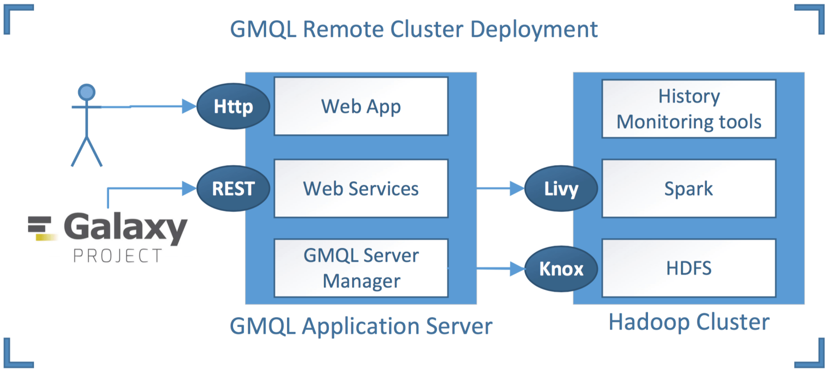


Figure 6.7.4: Deployment on an application server and Remote Hadoop cluster.

Information is the oil of 21th century, and analytics is the combustion engine

Peter Sondergaard, Gartner Research

7

Scaling-out GMQL operators on Data Flow Engines

Parallelizing genomic operations is an essential ingredient of the GMQL system. This chapter overviews the various basic approaches to parallel computing for scientific data, and then dwells into the implementation used in GMQL, which is based on binning the genome. We focus on domain-specific operations: join, map, and cover, as they are computationally very heavy.

7.1 State of the art of interval intersection algorithms

There are three main approaches to interval intersection ; the use of Linear Sweep algorithm, of R-trees, and of Map reduce; we show them below in details.

7.1.1 Linear Sweep Algorithm

In computational geometry, a **sweep line algorithm** [82] or plane sweep algorithm is a type of algorithm that uses a conceptual sweep line or sweep surface to solve various problems in Euclidean space. It is one of the key techniques in computational geometry.

The idea behind algorithms of this type is to imagine that a line (often a vertical line) is swept or moved across the plane, stopping at some points. Geometric operations are restricted to geometric objects that either intersect or are in the immediate vicinity of the sweep line whenever it stops, and the complete solution is available once the line has passed over all objects.

Our problem is defined for interval intersection as follows: Given a set I of n intervals $[l_i, r_i] \subset R, 1 \leq i \leq n$, compute all pairs of intervals from that intersect. This can be solved in $O(nlogn + k)$ time and O(n) space, where k is the number of intersecting pairs from $\binom{I}{2}$.

First observe that two real intervals intersect if and only if one contains right endpoint of the other.

Sort the set $\{(l_i, 0)|1 \leq i \leq n\} \cup \{(r_i, 1)|1 \leq i \leq n\}$ in increasing lexicographic order and denote the resulting sequence by P. Store along with each point from P its origin (i). Walk through P from start to end while maintaining a list L of intervals that contain the current point $p \in P$.

When ever $p = (l_i, 0), 1 \leq i \leq n$, insert i into L. When ever $p = (r_i, 1), 1 \leq i \leq n$, remove i from L and then report for all $j \in L$ the pair i,j as intersecting.

By keeping L intervals in memory, the maximum length of L cannot exceed the number of intervals; thus, the in-memory implementation is at risk of errors with big set of intervals.

Simpler implementation of this algorithm is adopted by the state of the art tools for region intersection, i.e. BedOps [83] and BedTools version 2.18 [84]. In this implementation, the interval is kept as a pair $I_i(l, r)$ and the sorting is performed based on the left end of the interval only. Then, the intersection is checked sequentially with no regards to the position of the right end of the interval. This implementation is fast and does not consume much memory; as we have P sorted intervals by left end, the algorithm checks the current interval p_i in the sequence with the next adjacent interval p_{i+1} and if p_i does not intersect with p_{i+1} then we consider the p_i interval does not intersects with all intervals with index greater than i+1 so there is no need to keep it in memory for farther comparison. In this way, containing intervals (intervals that have one or more intervals inside it) can all be skipped except one (the first one sorted by left end).

7.1.2 Binning algorithms Using Trees

R-Trees[85] (Rectangular Trees) are used in three of the most famous tools for region (interval) intersection in genomics: the UCSC genome browser[86, 87] (with the Generic Model Organism Database[88] - GMOD), SAMTOOLS[89] and BedTools versions less than 2.8. R-Trees partitions intervals from datasets into hierarchical bins. Intervals from another dataset are then compared within matching bins, thus reducing the search space of the intersection check.

Constructing R-Trees starts by grouping the nearby points and represent them with their minimum bounding rectangle in the next higher level of the tree. Since all points lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. At the leaf level, each rectangle describes a single object; at higher levels the aggregation of an increasing number of objects. This can also be seen as an increasingly coarse approximation of the data set.

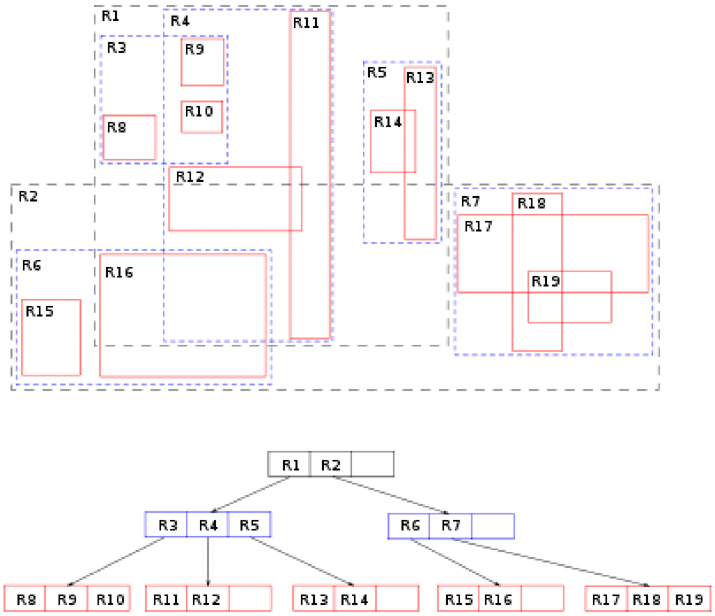


Figure 7.1.1: R-Trees simple example.

R-Tree is balanced just like a B-Tree, with maximum number of entries for leaf nodes (the best performance is observed with minimum number of entries equal to 30 to 40 percent of the maximum umber of entries.) R-Trees are stable for large databases and datasets, where nodes can be paged to memory when needed, but the whole tree cannot be kept in main memory.

R-Trees binning approach is shown in Fig.7.1.1[1]. In addition to the construction time of R-tree, a major problem in this approach is thread divergence, which occurs when intervals are not uniformly distributed inside the bins, leading to unbalanced bin sizes. If we represent the intervals with left and right ends as coordinates, we would have something similar to the example in Fig.7.1.1.

Several refinements of R-Trees are proposed to increase the performance of either the construction or the query time, where minimization of both coverage and overlap is crucial to the performance. Overlap means that, on data query or insertion, more than one branch of the tree needs to be expanded (due to the way data is being split in regions which may overlap). Minimizing coverage improves the pruning performance, allowing to exclude whole pages from search more often.

The R+ Tree[90] is a refinement of R-trees that looks for data using location of (x,y) on earth surface; it avoids overlapping of internal nodes by inserting an object into multiple leaves if necessary. R+ trees differ from R trees in the following aspects: (1) nodes are not guaranteed to be at least half filled, (2) the entries of any internal node do not overlap, (3) an object ID may be stored in more than one leaf node.

R*-trees are a variant of R-trees used for indexing spatial information. R*-trees have slightly higher construction cost than standard R-trees, as the data may need to be reinserted; but the resulting

[1] https://en.wikipedia.org/wiki/Rtree

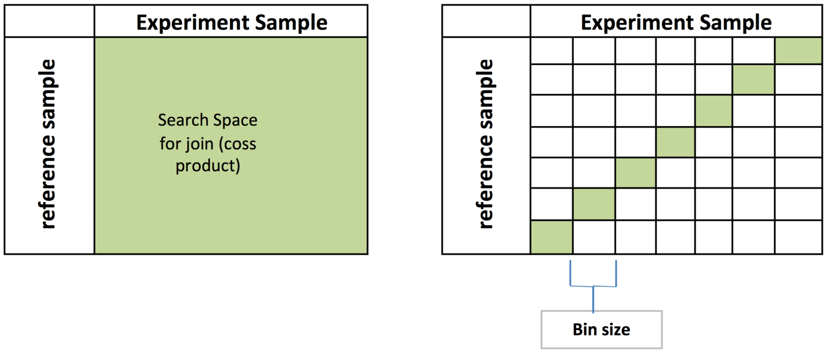


Figure 7.1.2: Binning search space. All intervals from reference that falls in one bin (bin n) will be intersected only with intervals fall in (bin n).

tree will usually have a better query performance. Like the standard R-tree, it can store both point and spatial data. When a node overflows, a portion of its entries are removed from the node and reinserted into the tree. This has the effect of producing more well-clustered groups of entries in nodes, reducing node coverage.

7.1.3 Interval intersection using Map reduce

Some attempts were made to do interval intersection using map-reduce, others using directly the data flow engines Pig, Spark, or Flink.

BioPig [91] provides a set of Pig Latin extensions for specific processing of data files produced by next generation sequencing (NGS) machines. In BioPig, a software developer can write user defined functions (UDFs) in Java programming language, but then s/he has to manage the Pig Latin scripts manually. SeqPig in [92] also uses Pig Latin for scripting operations for genomics. SeqPig and BioPig use the Hadoop-BAM [93] library for casting the different genomics data types before Hadoop processing.

Data management systems developed so far concentrate on secondary analysis (e.g., [94–96]); Adam [97], an offset of Spark dedicated to genomics, is also focused on secondary analysis. Functionality of binning and regions operations are more general (Map, Join, Cover, Flat, Histogram, and summit) in GMQL than doing only region intersection by Adam.

Authors in [98], proposed a MapReduce work-flow that starts from the data from the sequencing machine and performs short read alignment using a parallel version of BWA aligner.

7.1.4 Discussion

Linear sweeping produces good performance (complexity of O(NLogN)) but since it is a linear scanning on a sorted intervals, parallelism is difficult to achieve without binning the data. Because those intervals have diverse lengths and are intersecting with each other. R-tree algorithms have a

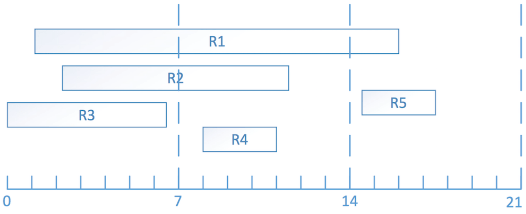


Figure 7.1.3: Regions before binning.

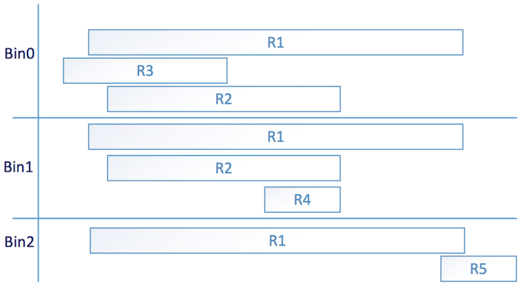


Figure 7.1.4: Regions of Fig.7.1.3 after assigning bins.

good parallelism since they bin the data as leafs of a tree, but they require a considerable construction time.

Building index with R-Trees is a good solution when a portion of the experiment data is queried only (not all the leafs); this is the case when we query a single interval or a small set of intervals over an R-Tree while the target intersecting intervals resides as well in a small number of leafs of the R-Tree. Maintaining and storing the index of the R-Tree is troublesome when we have big data because of the balancing mechanism of the R-Trees.

When the results of several queries (reference intervals) on an R-Tree are distributed on all the leafs (this is the case when we map a genes annotation reference sample to experiment sample, the genes annotation sample span on almost all the encoding genome space), then all the leafs should be scanned, possibly several times. The intervals in the reference set (each interval is considered a query on the R-Tree) are independent queries for the R-Tree, thus, the R-Tree is queried for each interval separately. Independent queries are good for parallelism but we may end up scanning the same bin (leaf) twice for two regions which intersects within the same bin. No optimization is performed to merge the queries targeting the same (bin) leaf in one query.

For the above reasons, we did not use R-Trees for binning but we designed our own binning method. The method is quite simple: we bin all the samples of GMQL operands (typically called reference and experiment) using the same bin size. All the regions that span more than one bin are replicated to all the bins which they span. This basic binning operation is slightly different for each GMQL operation, for example the min-distance and map operations use slightly different binnings; this will be discussed in the following sections. The basic binning operation is shown in Fig.7.1.3 and Fig.7.1.4.

We next present the domain-specific GMQL operations based on binning: Join, Map and Cover.

7.2 Join

Before discussing the join implementation, we discuss the clause evaluation order, the binning strategy, and the interaction between the binning strategy and the query-specific search space.

7.2.1 Introduction

The `JOIN` operation applies to two datasets, respectively called **anchor** and **experiment**; the operation produces a result sample for every pair of samples of the operand datasets, whose identifier is obtained by applying a hash function to the identifiers of the operand samples; the regions within each result sample are generated from the regions of the operand samples that satisfy a genometric predicate; their coordinates are computed according to four region composition options and their values are obtained by concatenating the values of the regions of the operands, where a full account of the join operation is presented, including region composition options, join partitioning, and metadata management. Thus, the join operation produces results that can grow quadratically both in the number of samples and of regions; hence, it is the most critical GMQL operation from a computational point of view.

Genometric predicates are based on the notion of **genomic distance**, defined as the number of bases (i.e. nucleotides) between the closest opposite ends of two regions, measured (using a numeric type, e.g. `Integer`) from the right end of the region with left end lower coordinate.[2] A genometric predicate is a sequence of distal conditions, defined as follows:

- `UP/DOWN` denotes the *upstream* and *downstream* directions of the genome. They are interpreted as predicates that must hold on the region of the experiment; `UP` is true when it is in the *upstream genome* of the anchor region[3]. When this clause is not present, distal conditions apply to both the directions of the genome.

- `MD(K)` denotes the *minimum distance* clause; it selects the *K* regions of the experiment at minimal distance from the anchor region. When there are ties (i.e. regions at the same distance from the anchor region), regions of the experiment are kept in the result even if they exceed the *K* limit.

- `DLE(N)` denotes the *less distance* clause; it selects all the regions of the experiment such that

[2] With our choice of interbase coordinates, intersecting regions have distance less than 0 and adjacent regions have distance equal to 0; if two regions belong to different chromosomes, their distance is undefined (and predicates based on distance fail).

[3] *Upstream* and *downstream* are technical terms in genomics, and they are applied to regions on the basis of their *strand*. For regions of the *positive strand*, `UP` is true for those regions of the experiment whose right end is lower than the left end of the anchor, and `DOWN` is true for those regions of the experiment whose left end is higher than the right end of the anchor. For the *negative strand*, ends and disequations are exchanged.

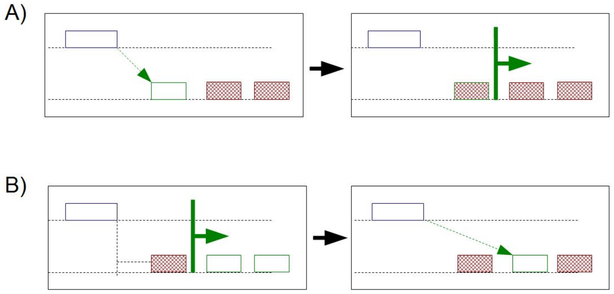


Figure 7.2.1: Different semantics of genometric clauses due to the ordering of distal conditions. The vertical bar is set at distance 100 from the reference region. In case (A) the minimum distance region is first selected (on the left) and then excluded by the distance predicate (on the right), therefore no region is produced. In case (B) the distance predicate selects two regions (on the left), out of which the minimum distance region is selected (on the right).

their distance from the anchor region is less than or equal to N bases[4].

- `DGE(N)` denotes the *greater distance* clause; it selects all the regions of the experiment such that their distance from the anchor region is greater than or equal to N bases.

Genometric clauses are composed by strings of distal conditions; a genometric clause is **well-formed** only if it includes the *less distance* clause; we expect all clauses to be well formed, possibly because the clause `DLE(Max)` is automatically added at the end of the string, where `Max` is a problem-specific maximum distance.

Examples. The following strings are legal genometric predicates:

```
DGE(500), UP, DLE(1000), MD(1)
DGE(50000), UP, DLE(100000)
DLE(2000), MD(1), DOWN
MD(100), DLE(3000)
```

Note that different orderings of the same distal clauses may produce different results; this aspect has been designed in order to provide all the required biological meanings, and is further discussed in Section 7.2.2, where we discuss distal clause evaluation.

Example. In Fig. 7.2.1 we show an evaluation of the following two clauses relative to an anchor region: `A: MD(1),DLE(100), B: DLE(100),MD(1)`. In case A, the `MD(1)` clause is computed first, producing one region which is next excluded by computing the `DLE(100)` clause; therefore, no

[4] DGE(-1) is true when the region of the experiment overlaps with the anchor region; DGE(0) is true when the region of the experiment is adjacent to or overlapping with the anchor region.

region is produced. In case B, the DLE(100) clause is computed first, producing two regions, and then the MD(1) clause is computed, producing as result one region[5].

Similarly, the clauses A: MD(1),UP and B: UP,MD(1) may produce different results, as in case A the minimum distance region is selected regardless of streams and then retained iff it belongs to the upstream of the anchor, while in case (B) only upstream regions are considered, and the one at minimum distance is selected.

7.2.2 Evaluation Steps

The order of execution of distal conditions influences the result; this depends on the fact that the min distance clause (MD) clause is not commutative with the greater distance clause (GLE) and with the stream clause (UP/DOWN); the less distance clause DLE is commutative with all other clauses, and stream and greater distal clauses are commutative with each other. Thus. the evaluation of a genometric predicate requires a sequence of 3 steps, where clauses within each step are commutative and each step can be missing:

- *Step 1* includes the DLE clause of the query and the stream and greater distal clauses which preceed the MD clause; if a query-specific DLE clause is not present, then DLE(Max) is added, where Max denotes the maximum biological distance[6].
- *Step 2* includes the MD clause.
- *Step 3* includes the stream and greater distal clauses after the MD clause.

Examples. The genometric predicate:

```
DGE(500), MD(10), UP
```

produces the following three steps:

```
Step 1: DGE(500), DLE(Max)
Step 2: MD(10)
Step 3: UP
```

The genometric predicate: DOWN, MD(10), DGE(2000), DLE(5000) produces the following three steps:

[5] The two queries can be expressed as: *produce the minimum distance region iff its distance is less than 100 bases* and *produce the minimum distance region after 100 bases*.

[6] If a query includes the clause DLE(M1) and *M*1 > *Max*, the clause is turned into DLE(M) by the execution engine; users can set the maximum biological distance of each query execution.

```
Step 1: DOWN, DLE(5000)
Step 2: MD(10)
Step 3: DGE(2000)
```

Some simpler predicates may require a single step, e.g. `DGE(50000), UP, DLE(100000)` is mapped to Step 1.

7.2.3 Binning and Search Space

The process of binning splits every chromosome of the genome into several bins of equal size S; for each chromosome, bins are progressively numbered starting from 0 and the *i-th* bin spans from $S \times i$ to $S \times (i+1) - 1$. For a given bin size S, a point placed at i bases from the chromosome start is assigned to the bin $b(i) = \lfloor i/S \rfloor$. Intervals between a left end l_i and a right end r_i are assigned to the bins between $b(l_i)$ and $b(r_i)$.

In order to effectively evaluate distal clauses, each anchor regions is associated with its **search space**, consisting of intervals of bins that may include matching regions of the experiment; search spaces are built according to the distal conditions of Step 1; it includes all potential matches, as Steps 2 and Step 3 are filters of the regions produced by Step 1. Consider an anchor region with left end l and right end r; let M be the maximum distance and let B_c denote the last bin of each chromosome c [7]. Then:

- If the clause is $LTE(d)$, then the search space is the interval of bins between $b(l-d)$ (excluding bins with $i < 0$) and $b(r+d)$ (excluding bins with $i > B_c$).
- If the clause is $LTE(d_1)$ and $GTE(d_2)$, with $d_1 > d_2$, then the search space is the two intervals of bins between $b(l - d1)$ and $b(l - d2)$ (excluding bins with $i < 0$) and between $b(r + d_2)$ and $b(r + d_1)$ (excluding bins with $i > B_c$).
- If the clause is $GTE(d_1)$, then the search space is the two intervals of bins between $b(l - M)$ and $b(l - d_1)$ (excluding bins with $i < 0$) and between $b(r + d_1)$ and $b(r + M)$ (excluding bins with $i > B_c$).

When the `UP`/`DOWN` clause is present, the search space is limited to the upstream/downstream directions of the genome. A representation of the search space for the anchor region as effect of the `DLE` and `DGE` clauses is shown in Fig. 7.2.2 (cases 1 and 2); the third case shows the effects of combining the `DLE`, `DGE` and `DOWN` clauses.

[7] Given that chromosomes have different sizes, B_c is a specific number for each chromosome.

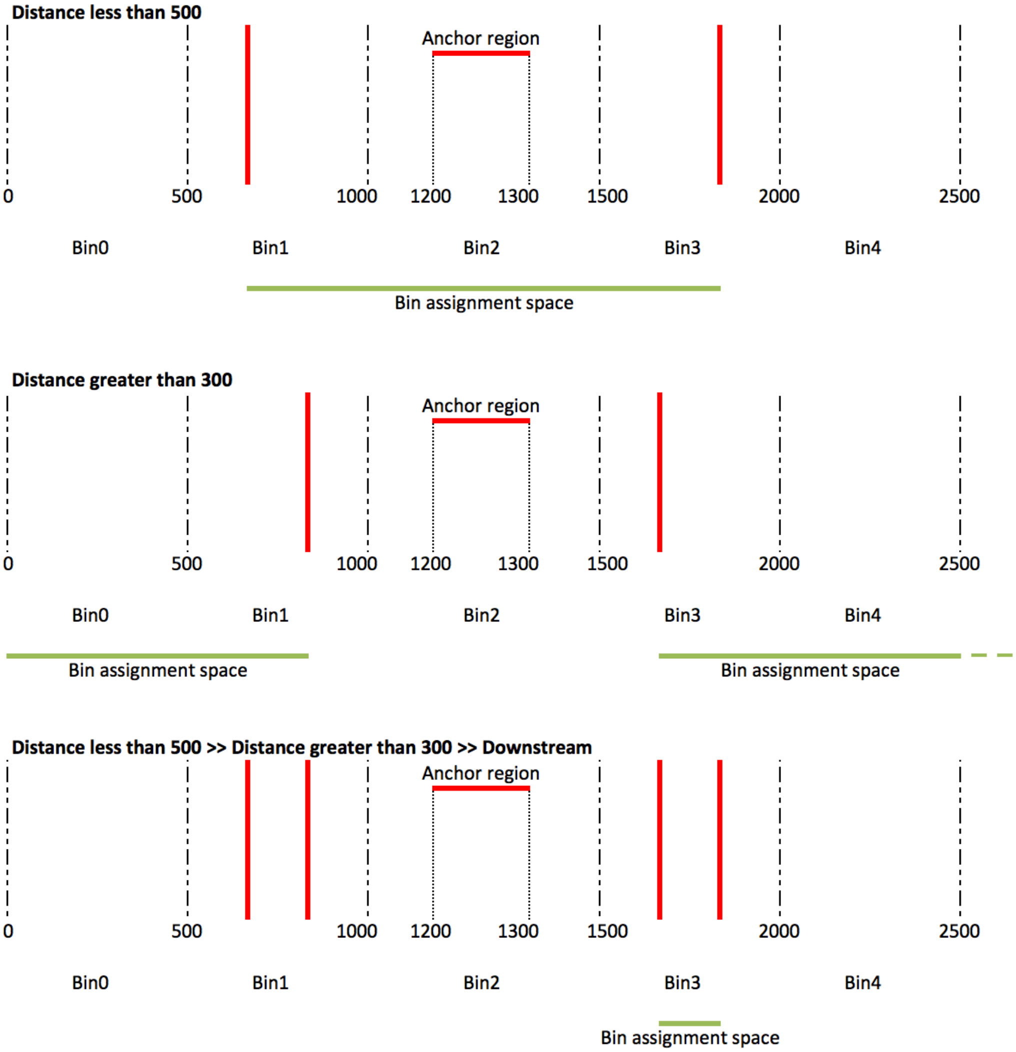


Figure 7.2.2: Search spaces for three distal clauses, Step 1. The dashed lines show the borders of bin size 500 bases, the red lines are the search space borders, the green line shows the search space confirming to the query, and the anchor region is the reference region (size 100 bases) that we are calculating the search space for. Each line is an example and has its query on the top left corner.

7.2.4 Evaluation of Distal Clauses in Step 1

This construction allows a parallel evaluation of join predicates. In particular, the following theorem holds due to the way in which search spaces are constructed:

Theorem 1. *The join predicate between an anchor region and any experiment region falling outside of its search space is false.*

In addition, we would like to evaluate the Step 1 join predicate between given regions of the anchor and experiment in a given bin only, so as to generate the corresponding result region only once, avoiding duplicates. The following theorem provides a solution of this problem.

Theorem 2. *If the Step 1 predicate between an anchor region and an experiment region is true, it can be tested in a given bin, denoted as* **testing bin**.

Proof. We build the proof by considering four cases which exhaustively cover the relationships between anchor and experiment regions, and defining the testing bin for each of them.

- Assume that *the experiment is at the left of the anchor*, i.e. the experiment's left end is strictly less than the anchor's left end and the experiment's right end is less than or equal to the anchor's right end. Then, the testing bin is the experiment bin with greatest number (the one at the smallest distance from the anchor); the predicate can be true only if the portion of experiment region within the testing bin intersects with the search space. Some examples are shown in Fig. 7.2.3, where the testing bin is denoted by a thicker trait. The predicate can be true in case (a) (when the testing bin falls within the search space) and is false in case (b) (as the region is too close to the anchor) and (c) (as the region is too distant from the anchor).

- The case when *the experiment is at the right of the anchor* is symmetric; in such case, the experiment's right end is strictly greater than the anchor's right end and the experiment's left end is greater than or equal to the anchor's left end. Then, the testing bin is the experiment bin with the smallest number; also in such case, the predicate can be true only if the portion of experiment region within the testing bin intersects with the search space.

- Assume that *the experiment is included within the anchor*. Recall that by construction the search space either properly includes the anchor region or does not overlap with it. Thus, the experiment can satisfy the join predicate only if it intersects with the search space in anyone of its bins; conventionally, we may use as testing bin the experiment bin with the smallest number. This case is illustrated in Fig. 7.2.4.

- Finally, assume that *the anchor is included in the experiment*. Then, the anchor is at negative distance from the experiment, and again the search space either properly includes the anchor

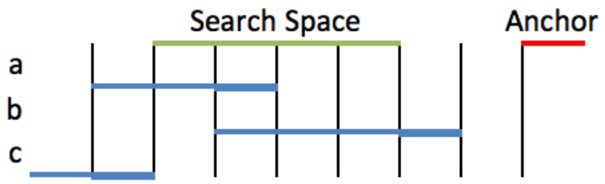


Figure 7.2.3: Experiment regions at the left of the search space. Every bar is one base unit.

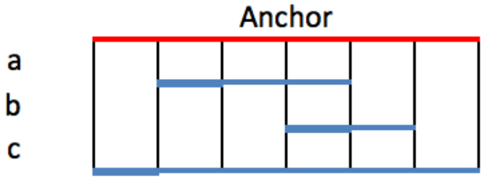


Figure 7.2.4: Experiment regions enclosed within the anchor region. Every bar is one base unit.

region or does not overlap with it; it follows that the join predicate between the region and the anchor can be true only if the search space includes the anchor. Conventionally, we may use as testing bin the anchor bin with the smallest number. This case is illustrated in Fig. 7.2.5.

□

Thanks to Theorem 2, at each bin B we evaluate Step 1 conditions just for those pairs of experiment and anchor regions such that B is their testing bin; thus, we either discard the pair of regions, or produce the resulting regions exactly once. This result is used by the parallel execution strategy which is next discussed.

7.2.5 Join Execution Strategy in Flink and Spark

Fig. 7.2.6 illustrates the flow of Flink and Spark operators for implementing a join operation. We recall that joins require first to select the pairs of samples that need to be joined, using a metadata predicate, and then to compute the result regions, using a genometric predicate. The operation applies to two datasets, respectively called *anchor* and *experiment*; as a running example we consider the join with:

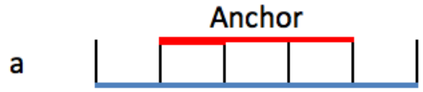


Figure 7.2.5: Anchor region enclosed within the experiment region. Every bar is one base unit.

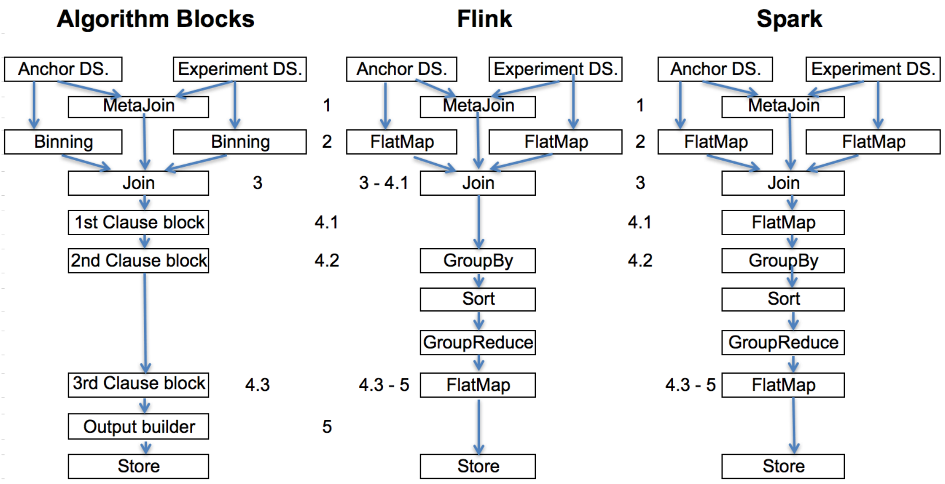


Figure 7.2.6: Operators for encoding the Join algorithm in Flink and Spark

```
Step 1: DGE(140), DLE(500)
Step 2: MD(1)
Step 3: DOWN
```

computed on:

```
Anchor: Id, chromosome, start, stop
1 C1 150 160
1 C1 285 390
Experiment: Id, chromosome, start, stop
2 C1 10  20
2 C1 430 550
2 C1 750 780
```

Throughout the examples of this section, we do not consider strands; in reality, join predicates evaluation is defined only between regions with compatible strand[8]. We also do not consider region values, they are carried along with each region and concatenated in the result[9].

- Block 1 (Metajoin) produces in output, for each anchor sample, the *join list* of the experiment samples that must be joined to it.

  Example. The join list of sample 1 is [2].

[8] Positive and negative strands are not compatible, and they are both compatible with undefined strands.

[9] With big value sizes, it is convenient to project the values prior to Block 1 and then join them to resulting regions within Block 5.

- Block 2 (FlatMap) is responsible of copying regions to the bins:
  - For every anchor region and bin b intersecting with the search space, it generates a copy of the anchor region for every bin b of the search space, by adding to it the attribute `Bin` (b) and the attribute `SBin` (the bin where the anchor region starts.)
  - For every sample of the join list and for every bin b intersecting with each experiment region, it generates a copy of the experiment region, by adding to it the attribute `Bin` (b) and the attributes `SBin` (the bin where the anchor region starts) and `EBin` (the bin where the anchor region ends.)

  Note that anchor regions are replicated at the bins of their search space, computed in this block, and experiment regions are replicated at the bins which intersect with them. The added attributes allow to test with a simple predicate if the current bin b is the testing bin of a given pair of anchor and experiment regions, based on the four cases of Theorem 2.

  Example. With a bin size $B = 100$, the first anchor region is copied to the bins $0, 2 - 6$, the second anchor region is copied to the bins $0 - 8$; the experiment regions is copied to the bins 0, $4 - 5$, and 7.

- Block 3 (Join) joins the anchor and experiments by `chrom` and `bin`. In this way, for any pair of anchor and experiment samples to be joined and for any of their anchor and experiment regions, all the relevant data are available at all bins, hence also at their testing bin. This operation is the most expensive, as it may join millions of regions to millions of regions; it is effectively computed by the `Join` operator, available in both frameworks. Performance depends on the bin size, as smaller bin size increases both replication and parallelism, therefore we study its optimal tuning as a function of data sizes and query parameters.

  Example. The following pairs are produced:

Bin	Chr	Id1	SB1	L1	R1	Id2	L2	R2
B0	C1	1	B1	150	160	2	10	20
B4	C1	1	B1	150	160	2	430	550
B5	C1	1	B1	150	160	2	430	550
B7	C1	1	B1	150	160	2	750	780
B0	C1	1	B3	285	390	2	10	20
B4	C1	1	B3	285	390	2	430	550
B5	C1	1	B3	285	390	2	430	550
B7	C1	1	B3	285	390	2	750	780

- Block 4.1 (Join in Flink, FlatMap in Spark) performs Step 1, by computing the distance between the regions in each row and then selecting only the rows of the testing bins where the distal conditions hold; testing bins are determined as indicated in the four cases of the proof of Theorem 2. This step is computed in parallel in each bin, in Flink is included in the `Join` of step 3, in Spark is a `FlatMap`.

  Example. The following pairs are produced:

```
Bin Chr  Id1 SB1 L1 R1   Id2 SB2 EB2 L2 R2 D
B4  C1   1 B1 150 160    2 B4 B5 430 550 270
B0  C1   1 B3 285 390    2 B0 B0 10 20 265
B7  C1   1 Bin3 285 390  2 B7 B7 750 780 360
```

- Block 4.2 (GroupBy, Sort, GroupReduce) performs Step 2, by selecting experiment regions based upon their minimal distance from anchor regions; it is implemented by the `GroupBy`, `Sort` and `GroupReduce` operators, but it requires data shuffling for collecting the experiment regions at nodes where sorting by distance and $top - k$ selection can be performed. We can reduce data shuffling with an alternative implementation, which adds a sort operation at each bin, producing at each bin the $top - k$ regions; these needs to be moved, while all other regions can be discarded. We discuss pros and cons of this alternative implementation in Section 9.1.3.

  Example. The following pairs are produced:

```
Bin Chr  Id1 SB1 L1 R1   Id2 SB2 EB2 L2 R2 D
B4  C1   1 B1 150 160    2 B4 B5 430 550 270
B0  C1   1 B3 285 390    2 B0 B0 10 20 265
```

- Block 4.3 (FlatMap) performs Step 3, by further reducing the filtered regions according to the distal conditions of Step 3. It uses the `FlatMap` operator.

  Example. In the example, the condition `DOWN` filters one pair, producing:

```
Bin Chr  Id1 SB1 L1 R1   Id2 SB2 EB2 L2 R2 D
B4  C1   1 B1 150 160    2 B4 B5 430 550 270
```

- Block 5 (FlatMap) is responsible of outputing the resulting pairs, by computing their sample identifier and their region coordinates according to the coordinate composition option and is executed together with block 4.3

Example. We finally obtain the following result, where a new sample identifier is generated as a hash function of the identifiers of the two operands, and the resulting region is obtained by concatenating the operand regions:

```
Id        Chr  Start Stop
Hash(1,2) Chr1 150   550
```

7.3 Map

MAP is a binary operation over two datasets, respectively called **reference** and **experiment**. Let us consider one reference sample, with a set of reference regions; the operation computes, for each sample in the experiment, new values produced by aggregation functions over the values of the experiment regions that intersect with each reference region; we say that *experiment regions are mapped to reference regions*. The operation produces a regular structure, called **genomic space**, where each experiment sample is associated with a row, each reference region with a column, and each matrix entry is a single value[10]. Thus, a MAP operation allows a quantitative reading of experiments with respect to the reference regions; when the biological function of the reference regions is not known, MAP helps in extracting the most interesting regions out of many candidates.

Example. When the input consists of one reference sample and three experiment samples, the output consists of three samples with the same regions as the reference sample, whose features corresponds to the number of mutations which intersect with those regions. The result can be interpreted as a (3×3) genome space.

The encoding of the Map operation as a sequence of operations for Spark and Flink is shown in Fig. 7.3.1. The algorithm requires to bin the two datasets, to group them by sample pair, chromosome and binning, to compute intersections within the bins, to compute aggregate functions, and output the results for each sample pair. The complexity of this problem grows quadratically with the sizes of the reference and experiment dataset. In the example, we count the experiment regions intersecting with reference regions; we consider:

```
Anchor: Id, chromosome, start, stop
1 C1 150 235
Experiment: Id, chromosome, start, stop
2 C1 10  230
```

[10] Biologists typically consider the transposed matrix, because there are fewer experiments (on columns) than regions (on rows). Such matrix can be observed using heat maps, and its rows and/or columns can be clustered to show patterns.

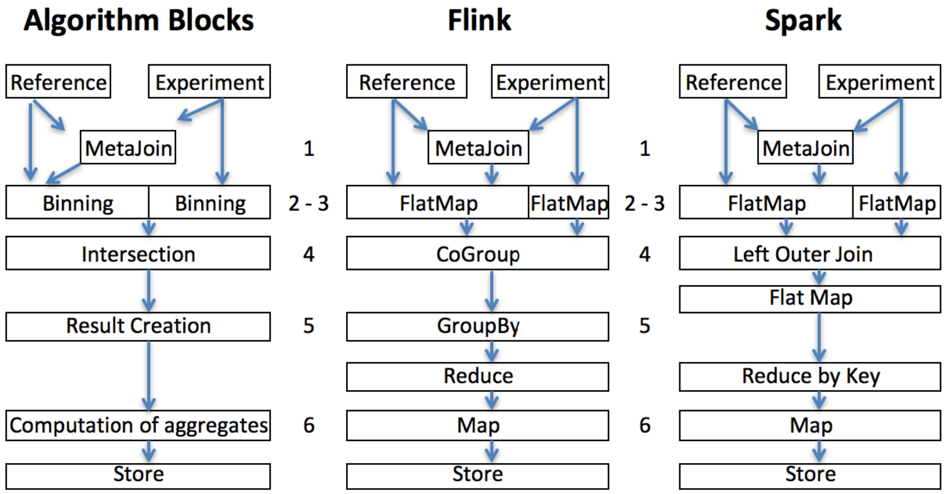


Figure 7.3.1: Operators for encoding the Map algorithm in Flink and Spark

- Block 1 (Metajoin) produces in output, for each reference sample, the *map list* of the experiment samples that must be mapped to it.

- Block 2 (Experiment Binning) is responsible of copying experiment regions to the bins. For every experiment region and bin b intersecting with the experiment, it generates a copy of the region for every bin b; only the attributes which are used by aggregate functions are copied.

  Example. With bins of size 100, the following copies are generated:

Id	Chr	Bin	Start	Stop
2	c1	0	10	230
2	c1	1	10	230
2	c1	2	10	230

  Note that a list of attribute values is generated, but no attribute value is needed for computing the `COUNT`.

- Block 3 (Reference Binning) is responsible of copying reference regions to the bins. For every reference region of a given sample, for every bin b intersecting with the reference, and for every experiment samples in its map list, a copy of the reference region is built, having as attributes the concatenation of `Id, Chr, Bin, Start, Stop` of the reference with the `Eid` of the experiment and with a new attribute `H` obtained by hashing all the attributes except the bin; this attribute is later used for assembling all copies relative to the same reference and experiment regions.

  Example. The following copies are generated:

```
Id Chr Bin Start Stop Eid H
1  c1  1 150 235 2 567
1  c1  2 150 235 2 567
```

- Block 4 (LeftJoin) is responsible of computing a partial map within each bin. It joins references and experiment by Eid, Chr and Bin; if the join succeeds, it further selects resulting tuples by considering only the bins where either the reference region or the experiment region start (note that this bin exists and is unique by construction). At each selected pair, a portion of the aggregate function is computed. A new region is built, having as attributes the concatenation of Chr, Bin with Rid, Start, Stop, H of the reference and EId, EStart, EStop, V of the experiment; V stores the experiment values to be used by the aggregate functions (in the case of Count, it stores 1.) If the join fails, because of the left join constructor, all the reference information is stored to the result, with null values stored for the experiment; in this way, all reference regions are correctly accounted.

  Example. The following copies are generated, and the second one is then filtered:

  ```
  Chr Bin RId Start Stop H EId EB EStart Estop V
  c1  1   1 150 235 567    2 c1 1 10 230 [1]
  c1  2   1 150 235 567    2 c1 2 10 230 [1]
  ```

- Block 5 (Assembling) is responsible of assembling all copies corresponding to the same reference and experiment at one node, through data shuffling; the operation is performed thanks to a reduce phase which uses the Hash attribute. Partial sums are performed for computing COUNT, and lists of attribute values are concatenated within a bag.

  Example. In the example, the two regions are reduced to one, as they have the same hash attribute. The following region is generated:

  ```
  Rid Chr Start Stop Val
  567 chr1 150 235 1
  ```

- Block 6 (Aggregating) is responsible of computing aggregate functions, by applying them to the bag of values built at block 5. This step does not apply to the running example.

7.4 Cover

The COVER operation applies to a single dataset and computes a single sample from several input samples by taking into account region intersections. In the basic COVER operation, each resulting region r is the contiguous intersection of at least minAcc and at most maxAcc regions r_i in the input samples; minAcc and maxAcc are called **accumulation indexes**[11].

Resulting regions may have new attributes Ar, calculated by means of aggregate expressions over the attributes of the contributing regions. Jaccard Indexes[12] are standard measures of similarity of the contributing regions r_i, added as default attributes. Three variants of the basic COVER are biologically relevant:

- The HISTOGRAM variant returns the nonoverlapping regions contributing to the cover, each with its accumulation index value, which is assigned to the **AccIndex** region attribute.

- The FLAT variant returns the union of all the regions which contribute to the COVER (more precisely, it returns the contiguous region that starts from the first end and stops at the last end of the regions which would contribute to each region of the COVER).

- The SUMMIT variant returns only those portions of the result regions of the COVER where the maximum number of regions intersect (more precisely, it returns regions that start from a position where the number of intersecting regions is not increasing afterwards and stops at a position where either the number of intersecting regions decreases, or it violates the max accumulation index).

We discuss the computation of the Histogram, i.e. of the accumulation index, as discussed in Section 3.5.1; all other properties of the Cover are easily derived from that index. A sequential algorithm for solving this problem consists of scanning the genome from left to right and maintain the accumulation count. At every start of a region the count is incremented, and at every stop is decremented; the result is given by every consecutive pairs of region ends with a positive counter. In the following, we propose a parallel version of this algorithm which relies on partitioning the genome into bins. The operation flow is shown in Fig. 7.4.1.

Example. We show only three regions, with the peculiarity that the first region stops where the second region starts and that the third region intersects with two bins, whose size is set to 500.

```
Id, Chr, Start, Stop
1 chr1 154 237
1 chr1 237 450
1 chr2 460 600
```

[11] The keyword ANY can be used as maxAcc, and in this case no maximum is set (it is equivalent to omitting the maxAcc option); the keyword ALL stands for the number of samples of the operand, and can be used both for minAcc and maxAcc; these can also be expressed as arithmetic expressions built by using ALL (e.g., ALL-3, ALL+2, ALL/2); cases when maxAcc is greater than ALL are relevant when the input samples include overlapping regions.

[12] The JaccardIntersect index is calculated as the ratio between the lengths of the intersection and of the union of the contributing regions; the JaccardResult index is calculated as the ratio between the lengths of the result and of the union of the contributing regions.

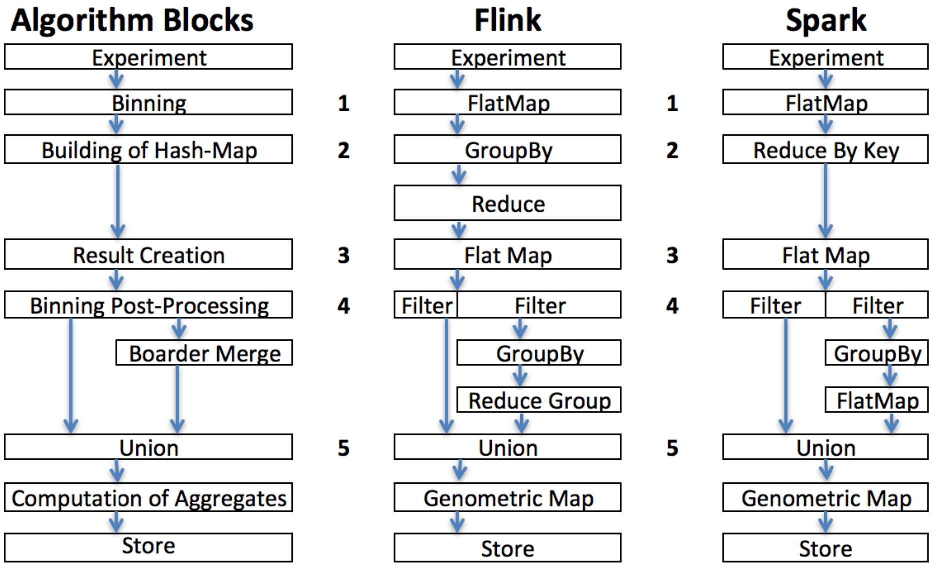


Figure 7.4.1: Operators for encoding the Cover algorithm in Flink and Spark.

```
1 chr2 580 700
```

- Block 1 (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).

```
Chr , 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}
chr2 1 {580->+1, 700->-1}
```

- Block 2 (GroupBy, Reduce) 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.

```
Chr, Bin, HashMap[Int,Int]
```

```
chr1 0 {154->+1, 237->0, 450->-1}
chr2 0 {460->+1, 500->-1}
chr2 1 {500->+1, 580->+1, 600->-1, 700->-1}
```

- Block 3 (FlatMap) returns the list of produced regions, along with their accumulation value, with each region placed within a bin, thus creating a raw histogram:

```
Chr, Start, Stop, Count
chr1 154  450  1
chr2 460  500  1
chr2 500  580  1
chr2 580  600  2
chr2 600  700  1
```

- Block 4 (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:

```
Chr, Start, Stop, Count
chr2 460 580 1
```

- Finally, Block 5 (Union) performs the union of the regions separately produced, Block 6 computes the aggregate (if any) using a Genometric Map operation and then (DataSink) writes them to the disk; it generates:

```
Chr, Start, Stop, Count
chr1 154 450 1

chr2 460 580 1
chr2 580 600 2
chr2 600 700 1
```

Part III

EVALUATION: Comparing Different GMQL Implementations

Performance is your reality. Forget everything else.

Harold S Geneen

8 Experimental Evaluation of GMQL Engine V1

In this chapter, we present the performance of the GMQL V1 system in comparison with the state-of-the-art; we also show that the system has a linear scale-up. Finally, we present a full example with biological interpretation. We measured performances both on Amazon Web Service cloud, taking advantage of multiple nodes, and on our server, an Intel® Xeon® Processor with CPU E5-2650 at 2.00 GHz, six cores, RAM of 128 GB, hard disk of 4x2 TB, and the engines Apache Hadoop 2.6.2, Apache Pig 0.15.0 and Apache Lucene 5.3.1.

8.1 GMQL V1 Implementation Optimization - Parallelism in the Generated Code

Several aspects of the translation contribute to the generation of high-performance Apache Pig code (which makes the translator a sort of syntax-directed optimizer):

- Use of **by-pair parallelism**, generated when an operations can be split into independent computations over pairs of samples.
- Use of **by-chrom parallelism**, which is generated by partitioning the `GROUP` and `CROSS` Apache Pig operations by chromosome. This is a classic *distributed join*; as result, regions are only produced from input regions with matching chromosomes[1].

[1] This parallelism is produced by changing, in Fig. 6.1.2, lines 11 and 12, where `GROUP` has to be applied by using also

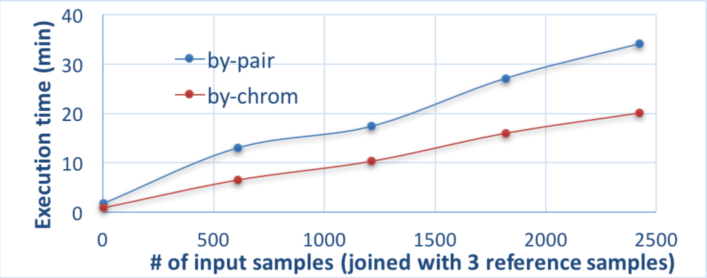


Figure 8.1.1: Performance of by-chrom parallelism added to by-pair parallelism.

- Use of suitable **parallelism directives** in Apache Pig operations, so as to improve the performance (e.g., setting the number of reducers as a function of the size of the input bags).

We instead delegate basic Apache Pig optimizations (such as dead-code deletion, filter pushing and so on) to the Apache Pig compiler, which is called upon the generated code.

8.1.1 Effect of By-Chrom Parallelism

In Fig. 8.1.1 we show the effect of adding by-chrom parallelism to the by-pair parallelism of the JOIN operation of three reference samples over an increasing number of experiment samples. The figure shows an important reduction of processing time with the addition of by-chrom parallelism, which depends both on the increased parallelism and reduced data sizes of operands (we also observed much smaller intermediate data sizes); moreover, all algorithms of join and map in GMQL V1 need the ordering of regions along the genome, and the time of ordering is also reduced with smaller data sizes. We found similar results in other operations and hence generally adopted the by-chrom parallelism together with the by-pair parallelism.

8.1.2 Size-Specific Tuning

In our experiments we used Apache Pig version 0.15.0; this version performs an automatic setting of internal parameters based on the size of the input of each Pig operation, controlling parallelization aspects such as the number of reducers. Our compiler allows overruling of the setting of the reducer threshold, limiting the amount of data that should be managed by each reducer; we noted better performance by increasing the standard number of reducers by a factor 4, dividing the input in chunks of 0.25 GB rather than 1 GB.

We then considered the behavior of Apache Pig operations with small input sizes. We noted that operations over metadata are less demanding and operations over regions are more demanding. Thus, after several experiments, we empirically produced a simple rule: if the size of inputs of an

the chromosome, and lines 13 and 14, where CROSS is turned into a join on the chromosome.

Apache Pig operation is above 2 GB, we simply set the reduce threshold to 0.25 GB; if it is below 2 GB, we assign to the Apache Pig operation a fixed number of reducers, equal to 1 if the operation applies to metadata and to 8 if the operation applies to regions, where 8 is the number of reduce slots available on our server. We tested this rule on many complex queries and obtained a performance gain **between 10% and 20%**[2].

8.2 Comparison with the State of the Art

No cloud computing system operates on region-based processed data, but BEDTools [84] and BEDOPS [83] are popular biologists' tools for scripting region-based computations which perform set-oriented operations upon regions; hence, we consider them the closest tools for a state-of-the-art comparison. They provide single machine code that uses multi-threading for some computationally expensive operations; they do not support implicit iteration over data samples or metadata management.

As BEDTools and BEDOPS are not cloud computing tools, they have excellent performance when applied to one pair of samples; however, these tools scale with great difficulty, both for what concerns programmability and performance. Since they do not support implicit iteration, for a comparison we coded a `read` function, which iteratively reads input files, and then scripted programs with explicit iteration. For instance, the GMQL operation:

```
RES = MAP(COUNT) GENE EXP;
```

is encoded by the following BEDOPS program:[3]

```
sort-bed ~/gene.bed > ~/file1_sorted.bed;
i = 0
while read NAME
do
 i = $((i+1))
 sort-bed $NAME > ~/file2_sorted.bed;
 bedmap  --ec --count --echo ~/file1_sorted.bed
  ~/file2_sorted.bed > "~/$i.bedOpsRes";
 echo "$i.$NAME";
done < ~/inputfiles.txt
```

[2] In Example 3.2 in [39], which includes a cascade of 4 GMQL joins and is translated into 32 Apache Pig operations that use reducers, we obtained a reduction of execution time of 17%, from 633 to 525 seconds.

[3] For brevity, we do not show the encoding in BEDTools and for distal join both in BEDTools and BEDOPS; neither BEDTools nor BEDOPS have metadata, hence we omitted from the translation of GMQL the Apache Pig code which loads and builds metadata.

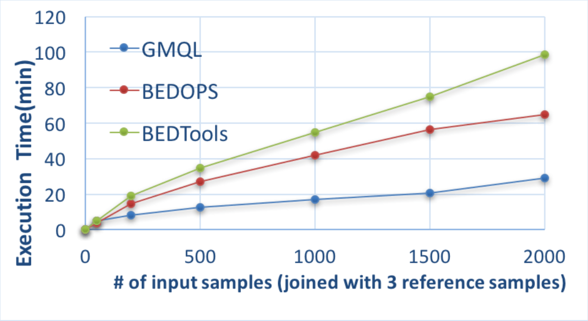


Figure 8.2.1: Performance of the `DISTANCE JOIN` operation with increasing number of samples; GMQL vs. BEDOPS vs. BEDTools.

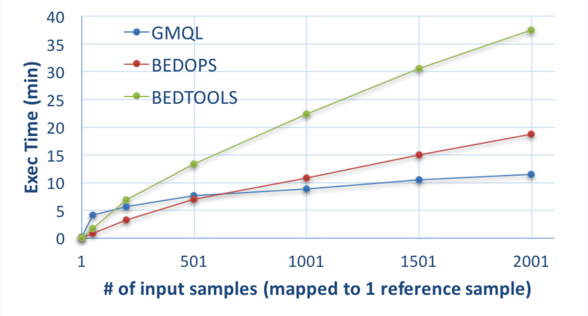


Figure 8.2.2: Performance of the `MAP` operation with increasing number of samples; GMQL vs. BEDOPS vs. BEDTools.

Fig. 8.2.1 shows comparative performances of the `DISTANCE JOIN` operation between three reference samples of about 45K regions each and an increasing number of experiment samples with an average of 50K regions and 7.5 MB size each. GMQL performs worse than BEDOPS and BEDTools when experiment samples are less than 50, but it outperforms them above such threshold.

Fig. 8.2.2 shows comparative performances of the `MAP` operation with a `count` aggregate function in the same experimental setting, but with a single reference sample; in this case, GMQL performs worse when experiment samples are less than 500, but it outperforms both BEDOPS and BEDTools above such threshold. Furthermore, GMQL time does not depend on the complexity of operation (e.g., number of computed aggregates), but rather to the need of distributing sample files, partitioned over the number of chromosomes, to the computing nodes. Instead, BEDOPS requires an increase of 30% of execution time for computing two aggregates, and BEDTools does not support multiple aggregates.

We conclude that BEDOPS has comparatively better performance than BEDTools, as independently reported in [99], and that beyond given thresholds GMQL performs faster on big data; moreover, GMQL provides a cloud computing solution, whose performance will increase with better operating system, better computing infrastructures and larger clouds.

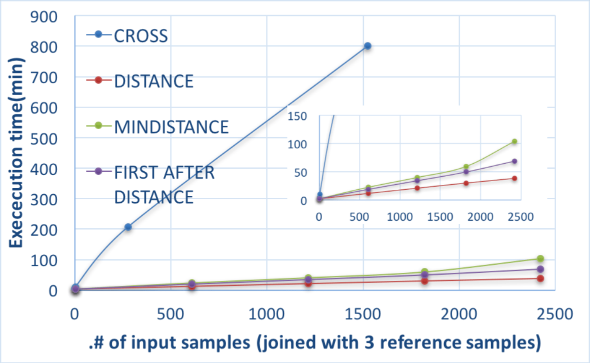


Figure 8.3.1: `JOIN` performance over big data.

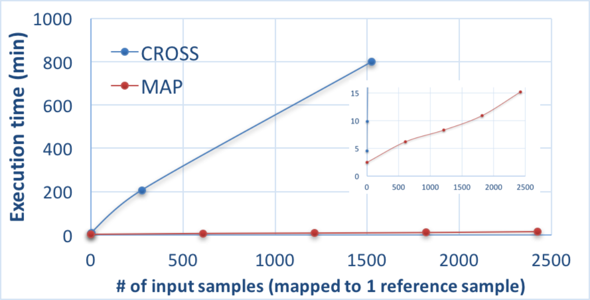


Figure 8.3.2: `MAP` performance over big data.

8.3 GMQL V1 Scaling with Big Datasets

Fig. 8.3.1 illustrates the performance of the three kinds of `JOIN` (`DISTANCE`, `MINDISTANCE` and `FIRST AFTER DISTANCE`), when executed with 3 samples of about 45K regions each as fixed references and a growing number of samples (up to 2.5K) as experiment samples; these samples have a variable number of regions and sizes, with an average of 50K regions and 7.5 MB size each. The diagram shows almost linear scaling; it also shows that the encoding of the `JOIN` as crossproduct (`CROSS`) has much worse performance.

`MAP` is a simple case of `DISTANCE JOIN`; hence, its performance curves are similar. Fig. 8.3.2 shows the `MAP` performance when the set of all known human genes (both protein coding and not) is used as single reference sample; note the linear scale up to only 15 minutes with 2500 experiment samples (77,778,000 regions) and 45K genes.

We also used the Hadoop framework provided by Amazon Web Services[4] with m3.2xLarge model, 8 CPU, 30 GB RAM, 2x80 GB storage SSD to test the `MAP` operation with over than 4000 samples, our largest dataset (about 31.1 GB); parallelism was set to 1 master node and 5, 10, and 15 slave nodes, respectively. Table 8.3.1 shows a significant reduction in execution time with the increase

[4] http://aws.amazon.com/

Table 8.3.1: Scalability of MAP execution time by increasing parallelism in the Amazon Web Service cloud

Master Nodes	Slave Nodes	Processing Time
1	5	29 min 14 sec
1	10	19 min 30 sec
1	15	16 min 30 sec

of the number of nodes, although scalability is lower when going from 10 to 15 nodes, most likely due to higher communication overhead.

8.4 Use Case Example

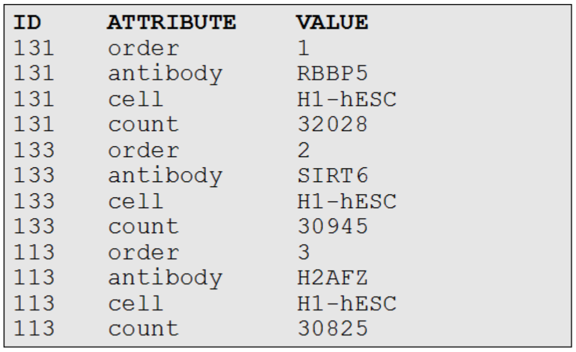
This example uses a MAP operation to count the peak regions in each ENCODE ChIP-seq sample that intersect with a gene promoter (i.e., proximal regulatory region); then, in each sample it projects over (i.e., filters) the promoters with at least one intersecting peak, and counts these promoters. Finally, it extracts the top 3 samples with the highest number of such promoters.

```
HM_TF = SELECT(dataType == 'ChipSeq') ENCODE;
PROM = SELECT(annotation == 'promoter') ANN;
PROM1 = MAP(peak_count AS COUNT) PROM HM_TF;
PROM2 = PROJECT(peak_count >= 1) PROM1;
PROM3 = AGGREGATE(prom_count AS COUNT) PROM2;
RES = ORDER(DESC prom_count; TOP 3) PROM3;
```

The query was executed over 2,423 samples including a total of 83,899,526 peaks, which first were mapped to 131,780 promoters within the ANN annotation dataset, producing as result 29 GB of data; next, promoters with intersecting peaks were counted, and the 3 samples with more of such promoters were selected, having between 30K and 32K promoters each. Processing required 11 minutes and 50 seconds.

The RES result variable includes both regions and metadata; the former ones indicate interesting promoter regions (that can be further inspected using viewers, e.g., genome browsers), the latter ones allow tracing provenance of resulting samples. Fig. 8.4.1 shows 4 metadata attributes of the resulting samples: the `order` of the sample, the `antibody` and `cell` type considered in the ChIP-seq experiment, and the promoter region `count`.

Further biological use case examples were thoroughly illustrated and discussed previously in [39].



ID	ATTRIBUTE	VALUE
131	order	1
131	antibody	RBBP5
131	cell	H1-hESC
131	count	32028
133	order	2
133	antibody	SIRT6
133	cell	H1-hESC
133	count	30945
113	order	3
113	antibody	H2AFZ
113	cell	H1-hESC
113	count	30825

Figure 8.4.1: Metadata excerpt of the resulting samples.

Don't lower your expectations to meet your performance. Raise your level of performance to meet your expectations. Expect the best of yourself, and then do what is necessary to make it a reality.

Ralph Marston

9

Experimental Evaluation of GMQL Engine V2

In this chapter, we evaluate the performance of the GMQL V2 system in comparison with V1; we also show the performance of domain specific operations and the scalability of the algorithms with the increase of the data size and number of nodes in the cluster. Testing has been done on a local server that has 16 core and 125 GB of RAM. GMQL V1 is executed on Apache Pig over Hadoop Yarn, while we have chosen the Spark implementation for V2 over Hadoop Yarn [51]. We noticed that the cluster resources reservation was very high in Pig execution (over 90% of the resources) while it was moderated for Apache Spark execution (less than half the resources dedicated for this processing). The data used for this test is a real data from Encode [22].

9.1 Join Operation Benchmark

Join operation has several options as described in chapter 3. The test includes five queries; the first query contains only a distance predicate, with no restriction on streams, thus it can be solved by using the first step of the join execution strategy discussed in Chapter 7. The second query contains in addition a *mindistance* predicate, therefore it requires two steps of the execution strategy. The third query contains a sequence of predicates: it includes *distance greater than*, then *up stream*, and finally a *distance less than* function; for this query, all the steps of the join execution strategy are needed. The fourth query, without a *mindistance* predicate, can be solved by executing only the first step; the fifth query is similar to the second one except that the *distance less than* condition is less restrictive,

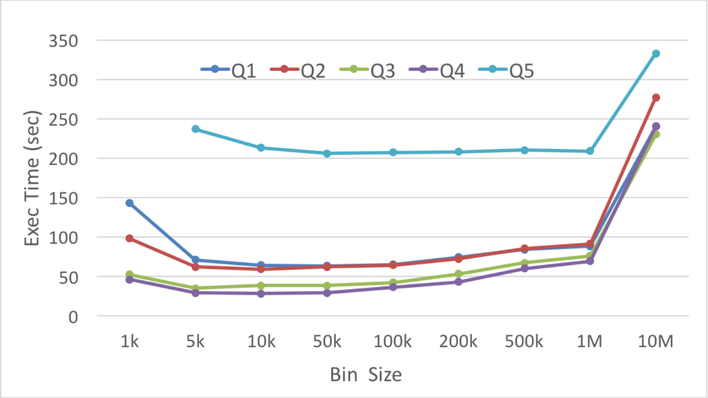


Figure 9.1.1: Execution times of 5 join queries with different bin sizes.

yielding to a larger result.

```
Q 1 = DistLess(100000)
Q 2 = DistLess(50000), MinDistance(2)
Q 3 = DistGreater(50000), Upstream(), DistLess(100000), MinDistance(1)
Q 4 = DistGreater(50000), Upstream(), DistLess(100000)
Q 5 = DistLess(200000), MinDistance(1), Upstream()
```

Fig.9.1.1 shows the execution times of the five join queries with different bin sizes. The test shows that the best bin size for all the join queries is the same and it does not change much, even though different queries require different steps of execution. The best performance in Fig.9.1.1 is associated with the *fourth query* since the fourth query limits the search space to up stream and has a smaller distance interval.

9.1.1 Optimal Bin Size

As discussed before, the rationale of binning is to reduce the number of regions to be considered within each bin, instead of computing chromosome-wide cross product; however, regions that cross the binning borders must be replicated. Large bins reduce replication of regions, but they lead to producing and matching many pairs of regions within each bin; conversely, short bins increase replication and therefore the generation of matching regions that should not be produced in output.

The choice of a good bin size depends on many factors:

- Physical characteristics of the cluster: if the executors have a large amount of memory, then larger bins can be used, as the cost of computing larger cross products in-memory is generally less than the cost of shuffling regions.

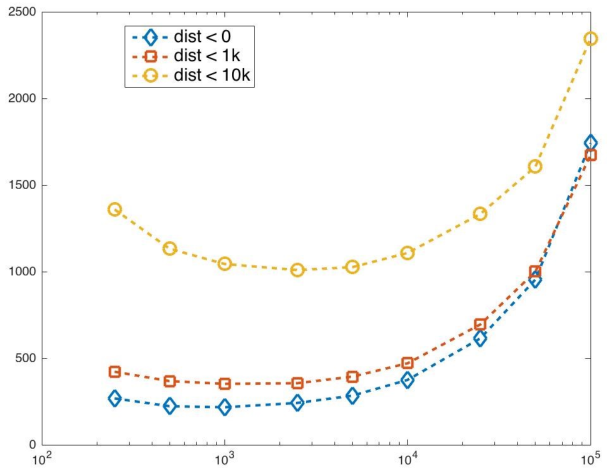


Figure 9.1.2: Execution time of Join as a function of bin size in logarithmic scale for Flink

- Total number of regions: when regions increase in number, smaller bins are needed in order to avoid huge cross products;
- Average region length: when regions are longer than bins, they are going to produce many replicates, thus increasing the cost of data shuffling and the number of useless tests.

Figure 9.1.2 shows the execution time of the join using the Flink engine, for different choices of the distal predicate (note that if the distance is less than 10K bases many more resulting regions are produced w.r.t. distances of 1K or of zero bases, yielding to longer execution times). In these cases, bin size between 1K and 10k bases are optimal. Figure 9.1.3 shows the same experiment using the Spark engine; in these cases, bin size close between $0.5 \times 10K$ and $5 \times 10K$ are optimal.

Figure 9.1.4 shows the execution time of the cover using the Flink engine, for different choices of the minimum and maximum accumulation indexes; note that the performance does not depend on accumulation indexes: once the histogram is computed (Block 3), then the extraction of result regions (Blocks 4-6) has very similar costs for any choice of accumulation indexes.

9.1.2 Performance comparison with different Bin Sizes

Join execution times of Flink and Spark are compared in Fig. 9.1.5. For small bin sizes and restrictive clauses (less matching regions) Flink has better performance, whereas for large bin sizes and more permissive clauses (more matching regions) Spark has better performance. In the Cover, pipeline parallelism results in faster execution times for Flink; Fig. 9.1.6 shows that the cover execution times are rather similar for an optimal choice of the bin size, but Flink outperforms Spark for

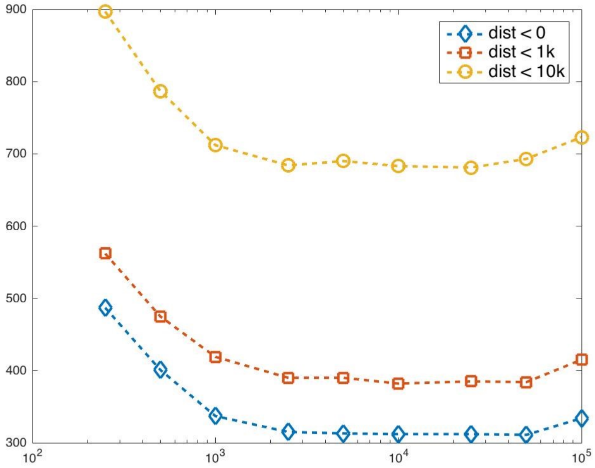


Figure 9.1.3: Execution time of Join as a function of bin size in logarithmic scale for Spark

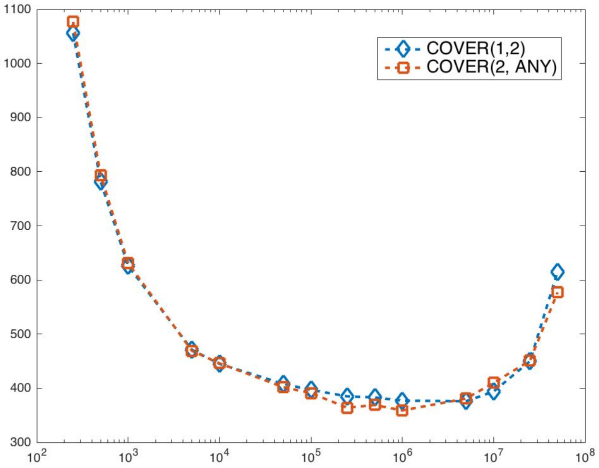


Figure 9.1.4: Execution time of Cover for Flink

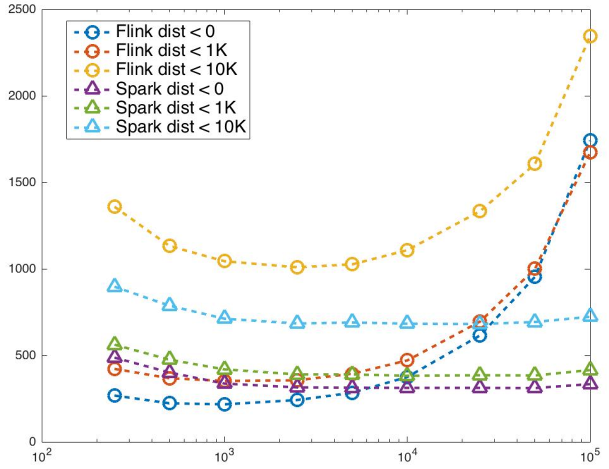


Figure 9.1.5: Comparison of Join execution times as a function of bin size and join clause in logarithmic scale for Flink and Spark

either small or large bins. These results are coherent with our findings described in the next Chapter.

9.1.3 Data Shuffling and Ordering with MinDistance

Each reference region is replicated to all the bins whose distance from the anchor region is less than the query constant; this is implemented by Block 2 of Section 7.2.5, and may generate a large number of regions satisfying the join condition at each bin, expecially when the query constant is set to Max, the maximum biological region length. This in turn may cause a lot of data shuffling for extracting the top-k regions of a minimal distance join; to reduce overhead, we suggested an alternative implementation which adds an intermediate sorting step (see Block 4.2 of Section 7.2.5). Figure 9.1.7 shows the network statistics (retrieved using Ganglia[1]); the area in the first rectangle corresponds to the standard execution and shows heavy data shuffling, while the area in the second rectangle refers to the alternative implementation which includes the intermediate sorting step and shows a much lighter data shuffling.

9.2 Map Operation Benchmark

Small Narrow peaks samples are used in tests that fit in memory, thus we can use algorithms without using binning in this test. The test has been done with the normal implementation of V1 and an implementation of GMQL V2 that does not have a binning step, since we did the test on Narrow

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

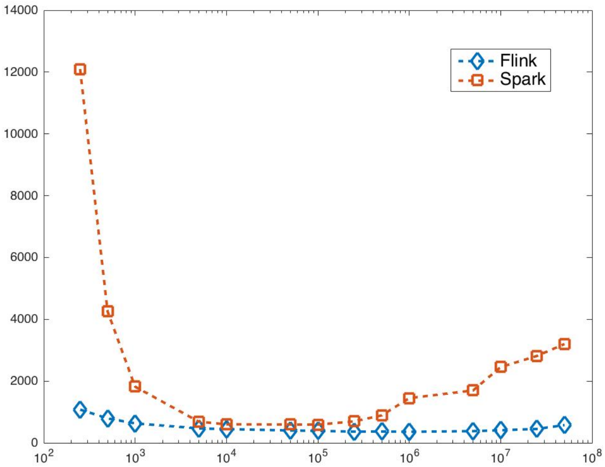


Figure 9.1.6: Comparison of Cover execution times as a function of bin size in logarithmic scale for Flink and Spark

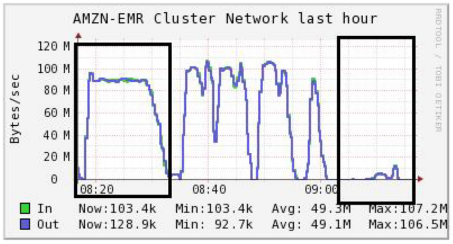


Figure 9.1.7: Comparison of data shuffling strategies

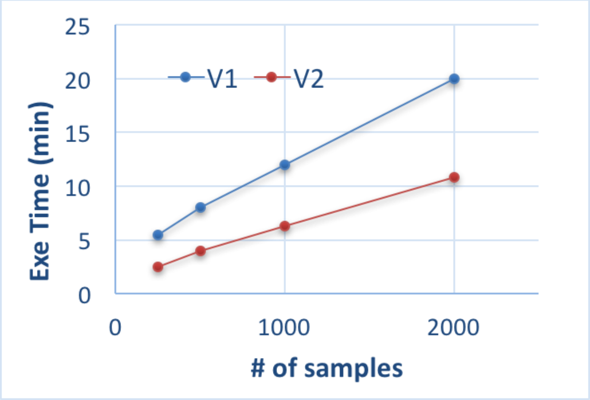


Figure 9.2.1: Map Operation with a single reference.

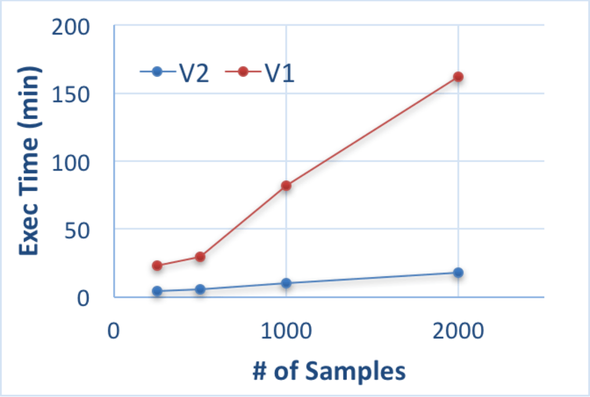


Figure 9.2.2: Map Operation with a multiple reference samples (11 samples).

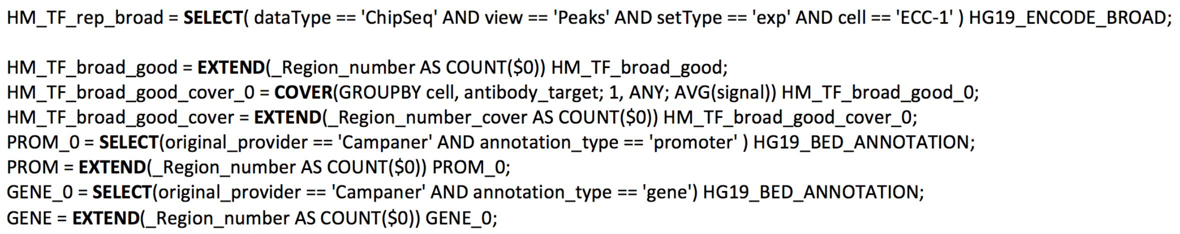
peaks data [22]. Narrow peaks data can fit in memory. The V2 implementation of the Map operation is implemented in Apache Spark. The test uses two different reference sets; the first set contains only one sample (genes regions) and another reference dataset contains 11 annotation sample.

The tests in Fig.9.2.1 shows Map operation performance with the increase of data size between V1 and V2. V2 shows linear speed up over V1 with the increase of the number of samples. Fig.9.2.2 shows that V2 is less sensitive to the increase of the size of the reference dataset in comparison to V1.

9.3 GMQL V1 Versus V2 Comparison

We used a real life application to show the power of GMQL and the difference of profiling of the application in V1 and V2. The application is composed of 4 phases:

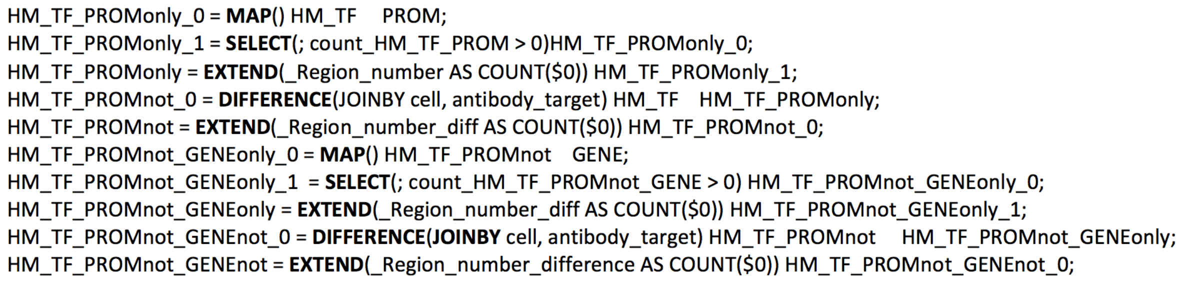
- Preparation: pre-process in data by selecting the data needed from the repository. Data selected are promoters, genes, and Transcription factors (TF) from broad Peaks, and narrow peaks (from Encode). The pre-processing includes data section, counting the number of re-



```
HM_TF_rep_broad = SELECT( dataType == 'ChipSeq' AND view == 'Peaks' AND setType == 'exp' AND cell == 'ECC-1' ) HG19_ENCODE_BROAD;

HM_TF_broad_good = EXTEND(_Region_number AS COUNT($0)) HM_TF_broad_good;
HM_TF_broad_good_cover_0 = COVER(GROUPBY cell, antibody_target; 1, ANY; AVG(signal)) HM_TF_broad_good_0;
HM_TF_broad_good_cover = EXTEND(_Region_number_cover AS COUNT($0)) HM_TF_broad_good_cover_0;
PROM_0 = SELECT(original_provider == 'Campaner' AND annotation_type == 'promoter' ) HG19_BED_ANNOTATION;
PROM = EXTEND(_Region_number AS COUNT($0)) PROM_0;
GENE_0 = SELECT(original_provider == 'Campaner' AND annotation_type == 'gene') HG19_BED_ANNOTATION;
GENE = EXTEND(_Region_number AS COUNT($0)) GENE_0;
```

Figure 9.3.1: Preparation phase of the application



```
HM_TF_PROMonly_0 = MAP() HM_TF    PROM;
HM_TF_PROMonly_1 = SELECT(; count_HM_TF_PROM > 0)HM_TF_PROMonly_0;
HM_TF_PROMonly = EXTEND(_Region_number AS COUNT($0)) HM_TF_PROMonly_1;
HM_TF_PROMnot_0 = DIFFERENCE(JOINBY cell, antibody_target) HM_TF    HM_TF_PROMonly;
HM_TF_PROMnot = EXTEND(_Region_number_diff AS COUNT($0)) HM_TF_PROMnot_0;
HM_TF_PROMnot_GENEonly_0 = MAP() HM_TF_PROMnot    GENE;
HM_TF_PROMnot_GENEonly_1  = SELECT(; count_HM_TF_PROMnot_GENE > 0) HM_TF_PROMnot_GENEonly_0;
HM_TF_PROMnot_GENEonly = EXTEND(_Region_number_diff AS COUNT($0)) HM_TF_PROMnot_GENEonly_1;
HM_TF_PROMnot_GENEnot_0 = DIFFERENCE(JOINBY cell, antibody_target) HM_TF_PROMnot    HM_TF_PROMnot_GENEonly;
HM_TF_PROMnot_GENEnot = EXTEND(_Region_number_difference AS COUNT($0)) HM_TF_PROMnot_GENEnot_0;
```

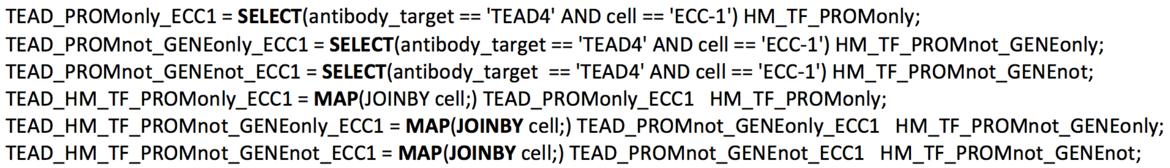
Figure 9.3.2: Annotating the transcription factors to Genes, Promoters, Not Genes or Promoters, and both.

gions in each sample (add it to its meta data), and materialize the result. This phase contains three `Selection` operations, and four `Extend` operations, and one `Cover` operation (cover is used from merging replicas), see Fig.9.3.1.

- Annotation: Transcription Factors produced by preparation step are annotated as Promoters and not promoters (extending the metadata of the promoters region number), and as Transcription factors that serves as genes or not. This operation contains two `Map` operations, two `Difference` operations, four `Extend` operations and two additional `Selection` operations on regions; in total 10 GMQL operations without the initial selection and the materialization operations. the final result of this step is a classification (annotation) of promoters, shown in Fig.9.3.2.

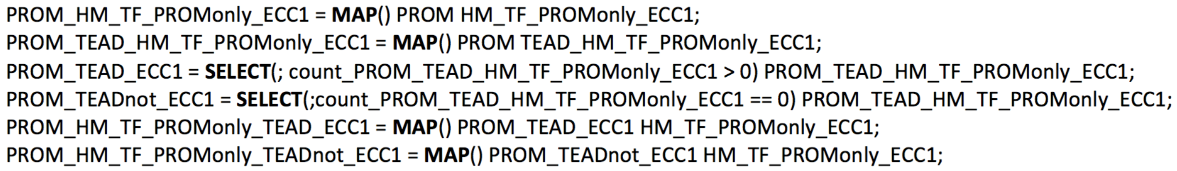
- Extraction: The cell line and antibody target needed for the study are extracted. This step contains three `Map` operations and three additional `Selection` operations; in total 6 GMQL operations shown in Fig.9.3.3.

- BiCData: Bicdata maps the antibody targets regions to the promotorial ones. This is done to know which and how many HM (Histon Modifications) and TF (Transcription Factor) regions are present in every promoter. To do so, we select the PROM sample prepared in Preparation, the *HM_TF_PROMonly* dataset annotated in Annotation, and then the sample selecting from *HM_TF_PROMonly* only the one of the antibody target TEAD4. This step



```
TEAD_PROMonly_ECC1 = SELECT(antibody_target == 'TEAD4' AND cell == 'ECC-1') HM_TF_PROMonly;
TEAD_PROMnot_GENEonly_ECC1 = SELECT(antibody_target == 'TEAD4' AND cell == 'ECC-1') HM_TF_PROMnot_GENEonly;
TEAD_PROMnot_GENEnot_ECC1 = SELECT(antibody_target  == 'TEAD4' AND cell == 'ECC-1') HM_TF_PROMnot_GENEnot;
TEAD_HM_TF_PROMonly_ECC1 = MAP(JOINBY cell;) TEAD_PROMonly_ECC1   HM_TF_PROMonly;
TEAD_HM_TF_PROMnot_GENEonly_ECC1 = MAP(JOINBY cell;) TEAD_PROMnot_GENEonly_ECC1   HM_TF_PROMnot_GENEonly;
TEAD_HM_TF_PROMnot_GENEnot_ECC1 = MAP(JOINBY cell;) TEAD_PROMnot_GENEnot_ECC1   HM_TF_PROMnot_GENEnot;
```

Figure 9.3.3: Extraction.



```
PROM_HM_TF_PROMonly_ECC1 = MAP() PROM HM_TF_PROMonly_ECC1;
PROM_TEAD_HM_TF_PROMonly_ECC1 = MAP() PROM TEAD_HM_TF_PROMonly_ECC1;
PROM_TEAD_ECC1 = SELECT(; count_PROM_TEAD_HM_TF_PROMonly_ECC1 > 0) PROM_TEAD_HM_TF_PROMonly_ECC1;
PROM_TEADnot_ECC1 = SELECT(;count_PROM_TEAD_HM_TF_PROMonly_ECC1 == 0) PROM_TEAD_HM_TF_PROMonly_ECC1;
PROM_HM_TF_PROMonly_TEAD_ECC1 = MAP() PROM_TEAD_ECC1 HM_TF_PROMonly_ECC1;
PROM_HM_TF_PROMonly_TEADnot_ECC1 = MAP() PROM_TEADnot_ECC1 HM_TF_PROMonly_ECC1;
```

Figure 9.3.4: BICDATA phase.

contains four Map operations in addition to three Select operations; in total 7 GMQL operations, shown in Fig.9.3.4.

The profiling of the four stages of this application is discussed in the same order of execution:

(A) The profiles of the application for both GMQL V1 execution and for GMQL V2 execution, shown in Fig. 9.3.5 and Fig.9.3.6, give insight of the improvement of GMQL from V1 to V2 and how it is distributed to the different operations. We can see from the profiling that the preparation phase is almost constant as portion of the execution time. **V2 execution is 4.6X faster than V1** in the preparation phase, as shown in Fig.9.3.8.

(B) From the profile shown in Fig. 9.3.5, we note that the annotation phase consumes the largest portion of the execution time of the application; this is due to the high number of Map reduce jobs generated to perform the operations of GMQL. The increment in the number of

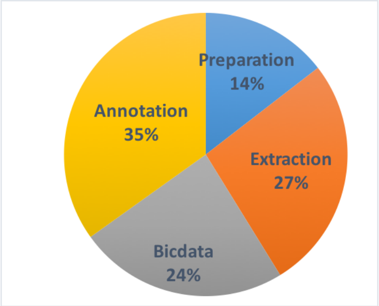


Figure 9.3.5: V1 application profiling.

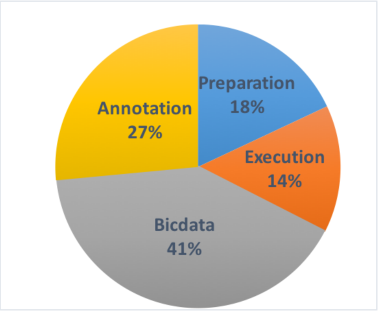


Figure 9.3.6: V2 application profiling.

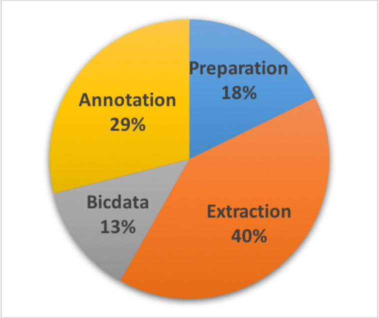


Figure 9.3.7: Speed Up of the application from V1 to V2.

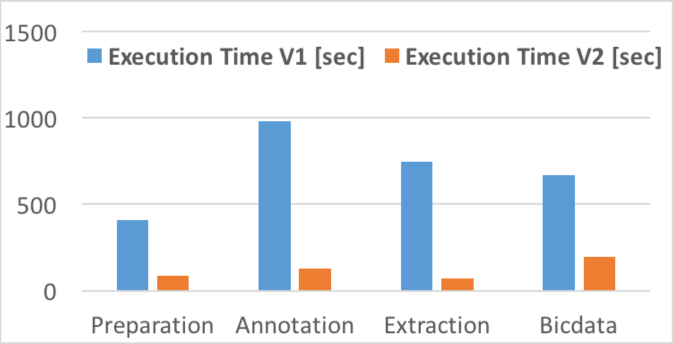


Figure 9.3.8: Performance comparison of GMQL V1 versus V2 for real case application.

Map Reduce jobs increases the number of read/write operations on the hard disk, since every MapReduce job materializes the result on hard disk and cannot stream it to the next map reduce job. In addition to that, the translation of GMQL to Pig Latin does not optimize the processing when if finds duplicate operation. For these reasons, **V2 execution is 7.6X faster than V1** in the annotation phase, as shown in Fig.9.3.8.

(C) The extraction phase contains a `JOIN`, which is the heaviest and most optimized in V2. operation in this phase which is the most optimized operation. For these reasons, **V2 execution is 10.6X faster than V1** in the extraction phase, as shown in Fig.9.3.8.

(D) Fig. 9.3.6, shows that the Bicdata phase takes the largest portion of the execution time in V2, while Fig. 9.3.7, shows that this phase has the lowest speed up in the applications. By looking to the operations in this phase, we notice that the most present operation is Map. Map Operations in V2 uses the binning mechanism which, as we have seen in chapter 7, adds extra overhead of the binning and the replication of the regions that crosses the bins. Samples of this case fit in memory and V1 uses sweep line algorithm on the pair of samples that it maps. Thus, **V2 execution is only 3.3X faster than V1**.

The total speed up of GMQL V2 execution over GMQL V1 execution for this application is **5.6X**. The application execution speed up details in regards to the phases is shown in Fig.9.3.7.

As a side note, in V1 the `Map` operation uses parallelism by sample and by chromosome, thus, the Map operation works well on samples that have small chromosomes that fit in memory. In case of very large samples, when a single chromosome from reference and experiment samples can not fit in memory, the GMQL V1 Map operation fails. This draw back of V1 was solved in V2 by the binning strategies mentioned in Chapter 6.

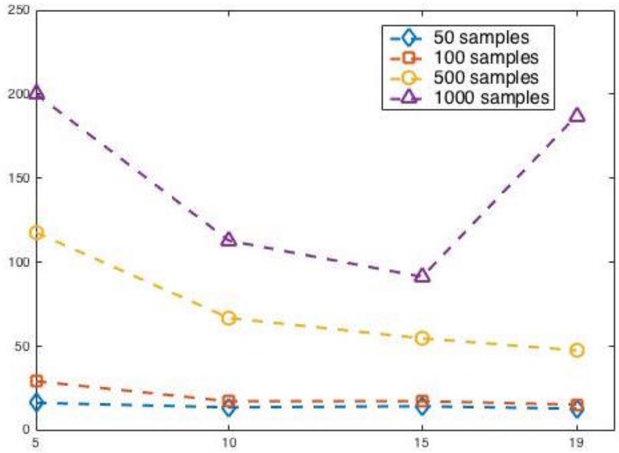


Figure 9.4.1: Scaling of execution time by increasing the number of AWS nodes

9.4 Performance Scaling with more AWS Nodes

We considered the execution of several joins with the same reference and an increasing number of samples, using the Flink execution engine, and scaling the size of the AWS network from 5 to 10, 15, and 19 nodes[2] Fig. 9.4.1 shows that the performance improves with largest networks with up to 500 samples, but in the case of 1000 samples the performance decreases; this is due to an excess of communication overhead with the addition of nodes.

Table 9.4.1 shows the query cost using AWS, which charges a fixed price per node and time unit[3]. We note that the cost of query execution increases with the growth of the network size, but this is compensated by a decreased execution time (see Fig. 9.4.1). However, with 1000 samples and going from 15 to 19 slave nodes, we note both an increase of cost and of execution time. An *elastic system* could benefit of constant monitoring of execution times, by dynamically shutting down nodes when such behavior occurs[4].

[2] Our AWS grant allows for configurations of at most 20 nodes.

[3] The hour cost of an Elastic Map Reduce (EMR) instance is the sum of the cost of nodes and the cost of EMR service. In our case we used M3.XLarge instances which cost 0,266$/h plus 0,070$/h for EMR service, yielding a total of 0,336$/h; therefore, a cluster of 6 nodes (5 slaves + 1 coordinator) of EMR costs 2,016$/h. Instances are paid hourly but we run batches of several experiments, thus we can redistribute execution costs to each query weighted by the execution time in seconds. For example, if a query takes 20 seconds to execute, then the cost of that query is 0,0112$.

[4] However, such query-specific elastic system cannot be easily developed in AWS, as configuration switching is time expensive.

# of samples	6 nodes	11 nodes	16 nodes	20 nodes
50 samples	0,0112	0,0205	0,0299	0,0373
100 samples	0.0168	0,0236	0,0403	0,0467
500 samples	0,0644	0,0770	0,0896	0,0933
1000 samples	0,1120	0,1232	0,1419	0,3640

Table 9.4.1: Cost of query execution (in US Dollars) with different samples and cluster sizes

You can have data without information, but you cannot have information without data.

Daniel Keys Moran

10 Comparative evaluation of Flink and Spark

In this chapter, we present a benchmark of the Flink and Spark engines on genomic abstractions which are used within the GMQL implementation. The benchmark is an indication of the ability of the two engines to deal with the requirements of big genomic data processing. We performed our experiments on the Amazon Cloud, in most cases with a small cluster of one master and five slaves (m3.xlarge); we used Flink-0.9.1[1] and Spark 1.52 [2]. We concentrate on Join and Cover (Map is very similar to Join). The datasets are: `TSS` for references (one sample of 131780 short regions from UCSC transcription start sites), and `NARROW` for experiments (1999 samples from Encode Narrow Peak Dataset, which has a total of 143 million regions and an average of 71915 regions per sample).

10.1 Framework Comparison

Flink and Spark are both general-purpose data processing platforms and top level projects of the Apache Software Foundation (ASF). They have a wide field of applications and are usable for dozens of big data scenarios. They support several extensions, e.g. to SQL-like queries (Spark: Spark SQL, Flink: MRQL), graph processing (Spark: GraphX, Flink: Spargel (base) and Gelly(library)), machine learning (Spark: MLlib, Flink: Flink ML) and stream processing (Spark Streaming, Flink

[1] flink-0.9.1/bin/yarn-session.sh -n 5 -jm 768 -tm 10752 -s 4; yarn.heap-cutoff-ratio = 0.15.

[2] default EMR parameters. spark-submit –master yarn –deploy-mode client –num-executors 20 –executor-memory 5G.

Streaming). Both are capable of running in standalone mode, yet most usage occurs on top of Hadoop (YARN, HDFS). For what concerns their differences[3]:

- Flink uses dataset variables and is optimized for cyclic or iterative processes by using iterative transformations on collections. This is achieved by an optimization of join algorithms, operator chaining and reusing of partitioning and sorting. However, Flink is also a strong tool for batch processing. Flink streaming processes data streams as true streams, i.e. data elements are immediately ”pipelined” though a streaming program as soon as they arrive. This allows to perform flexible window operations on streams.

- Spark is based on resilient distributed datasets (RDDs), (mostly) in-memory data structures giving to Spark the power of functional programming paradigms. Spark is capable of big batch calculations by binning memory; Spark streaming wraps data streams into mini-batches, i.e. it collects all data that arrives within a certain period of time and then runs a regular batch program on the collected data. While the batch program is running, the data for the next mini-batch is collected.

10.2 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.

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 the genome has been introduced within the UCSC Genome Browser [86] 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. 10.2.1 on the left.

[3] A thorough comparison of Flink and Spark can be found in http://stackoverflow.com/questions/28082581/what-is-the-differences-between-apache-spark-and-apache-flink.

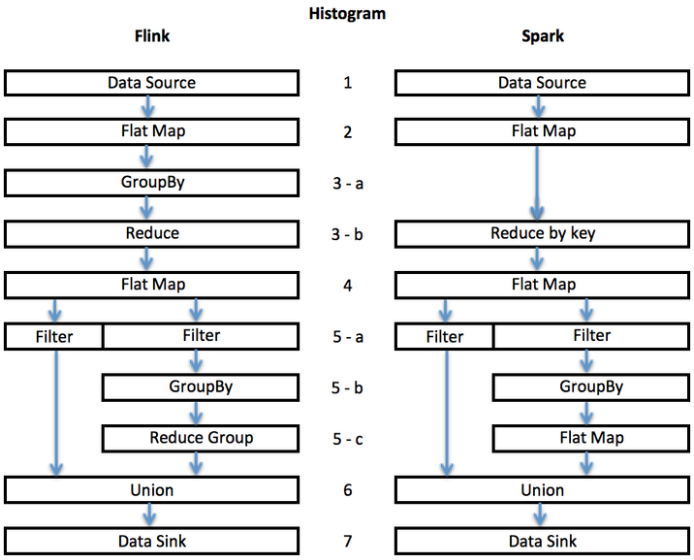


Figure 10.2.1: Operators for encoding the Histogram algorithm in Flink and Spark.

- 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
```

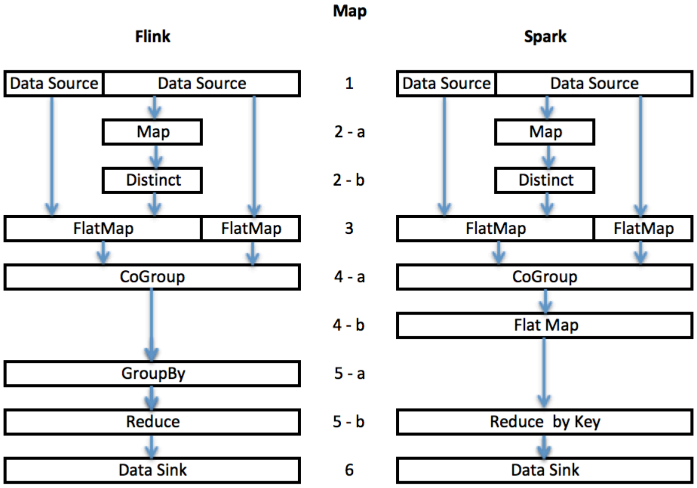


Figure 10.3.1: Operators for mapping experiments to references in Flink and Spark.

```
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. 10.2.1.)

10.3 Mapping to a Reference

The encoding of this problem as a sequence of operations for Spark and Flink is shown in Fig. 10.3.1. 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.

10.4 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.

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))
             }
    }
```

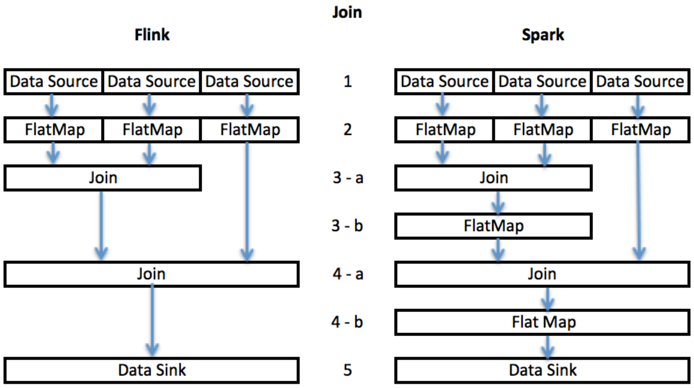


Figure 10.4.1: Operators for encoding the join algorithm in Flink and Spark

```
}
```

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. 10.4.1.

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. 10.4.1. 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. 10.4.1. 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.

10.5 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.

10.5.1 Histogram execution

We start by comparing how the Flink and Spark engine manage the blocks of operations discussed in Section 10.2. 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. 10.5.1.
- Spark groups the blocks within three stages, that are sequentially executed. In particular, Blocks 1-3 belong to Stage 1, Blocks 4-5 belong to Stage 2, and Blocks 5a-6-7 belong to Stage 3, as illustrated in Fig. 10.5.2.

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

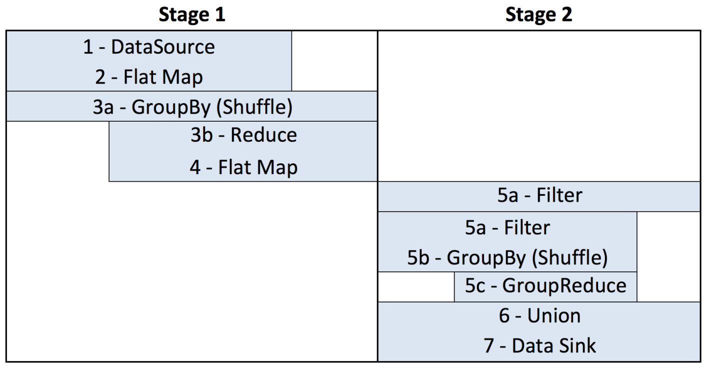


Figure 10.5.1: Stages of histogram computation in Flink.

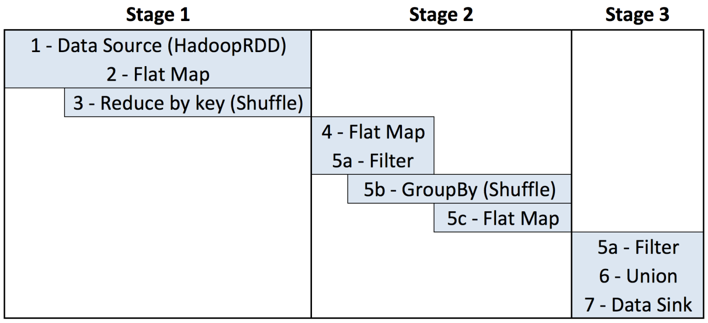


Figure 10.5.2: Stages of histogram computation in Spark.

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 10.5.1: 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 10.5.1. Regions are extracted from samples of Encode repository [21] 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 10.5.2: Execution times (in seconds) for the Histogram application.

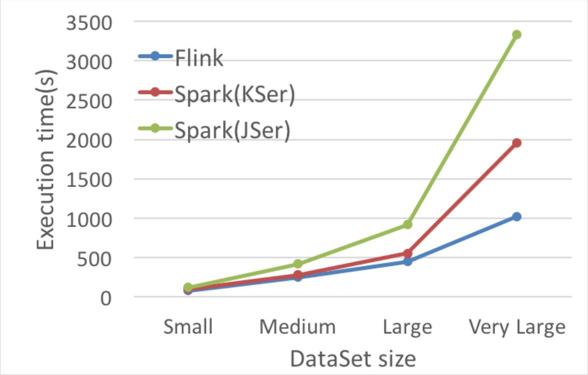


Figure 10.5.3: Execution time of the Histogram application in Flink and Spark, with 15 slave nodes and increasing sizes of input.

Execution times of the application in Flink and Spark with 15 slave nodes are reported in Table 10.5.2, and graphically compared in Fig. 10.5.3. 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 the default Java serializer and the Kryo general serializer[5]; performance was best with the latter choice, as shown in Fig. 10.5.3; in the *very large* setting, Flink outperforms Spark with Kryo serializer by a factor 2, and the Java serializer by a factor 3.

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

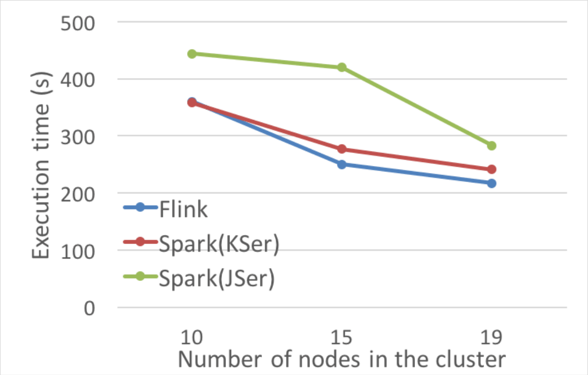


Figure 10.5.4: Execution time of the Histogram application in Flink and Spark, medium setting, with increasing nodes.

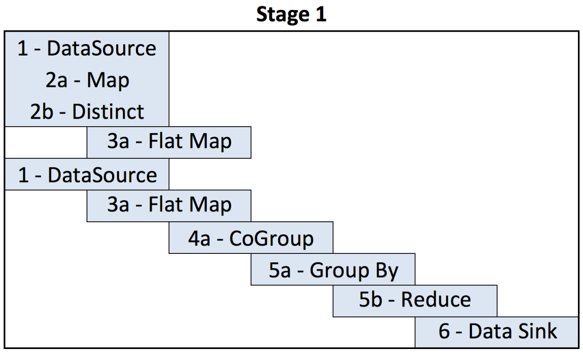


Figure 10.5.5: Stages of Map computation in Flink.

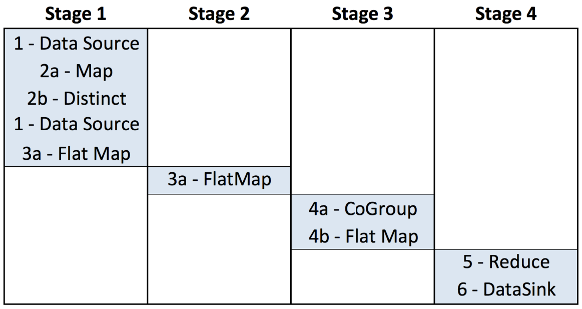


Figure 10.5.6: Stages of Map computation in Spark.

We next considered the *medium* setting and considered different cluster sizes, ranging from 10 to 19[6], see Fig. 10.5.4. In this setting, the two frameworks have very similar performance when Spark uses the Kryo serializer.

10.5.2 Map execution

We compare how the Flink and Spark engine manage the blocks of operations discussed in Section 10.3. We observe that Flink groups all the blocks within one stage (see Fig. 10.5.5), while Spark requires four stages (see Fig. 10.5.6). This is again an advantage for Flink in terms of less need for synchronization and greater parallelism.

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 10.5.3, and graphically compared in Fig. 10.5.7. In this application Flink outperforms Spark, showing

generic serializers.

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

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

Table 10.5.3: Execution times (in seconds) for the Map application.

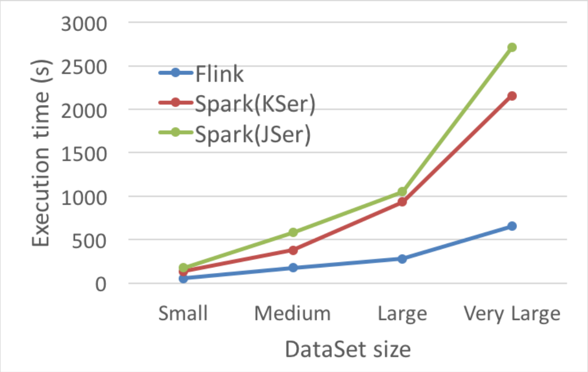


Figure 10.5.7: Execution time of the map application in Flink and Spark, with 15 slave nodes and increasing sizes of input.

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. 10.5.8. Note that the difference in performances further increases with 10 nodes.

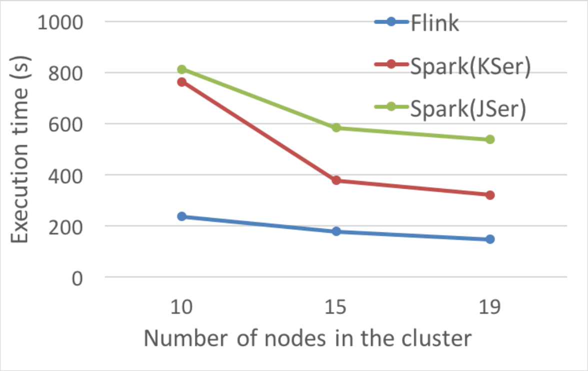


Figure 10.5.8: Execution time of the map application in Flink and Spark, medium setting, with increasing nodes.

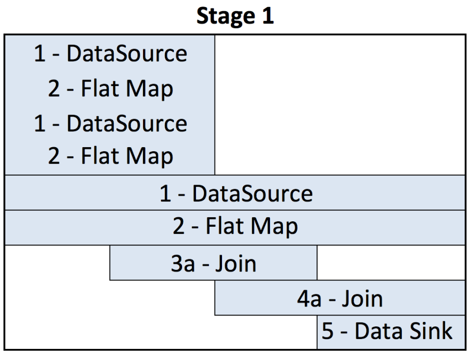


Figure 10.5.9: Stages of join computation in Flink.

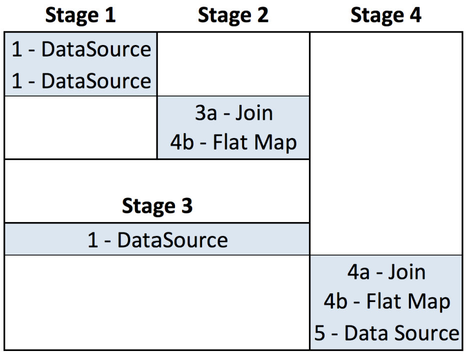


Figure 10.5.10: Stages of join computation in Spark.

10.5.3 Join execution

We finally consider the Join application of Section 10.4; we observe that Flink groups all the blocks within one stage (see Fig. 10.5.9), 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. 10.5.10).

For what concerns data sizes, we designed three cases, respectively named *small*, *medium*, and *large*, whose dimensions are summarized in Table 10.5.4; we generated three tables with very close numbers of regions. Regions have an attribute SCORE, used for the selection condition.

Execution times of the application in Flink and Spark with 15 slave nodes are reported in Table

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 10.5.4: Features of the datasets used in the Join application.

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

Table 10.5.5: Execution times (in seconds) for the Join application.

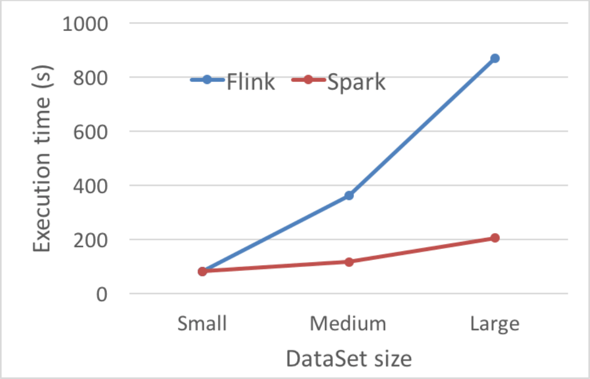


Figure 10.5.11: Execution time of the Join application in Flink and Spark, with 15 slave nodes and increasing sizes of input.

10.5.5, and graphically compared in Fig. 10.5.11; 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. 10.5.12).

Based on these experiments, we currently adopt a bin size of 10K bases for map and join and of 1M bases for cover both in Spark and Flink; we will also continue the development of GMQL on both systems, as the experiments do not demonstrate a clear winner between the two engines.

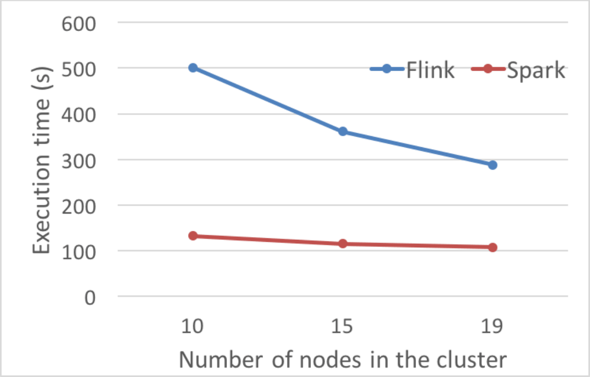


Figure 10.5.12: Execution time of the Join application in Flink and Spark, medium setting, with increasing nodes.

It is an immutable law in business that words are words, explanations are explanations, promises are promises but only performance is reality.

Harold S Geneen

11
Comparative Evaluation of Spark and SciDB

In this chapter, we present a benchmark of the Spark and SciDB engines at work on genomic abstractions which are used within the GMQL implementations. The benchmark compares a general-purpose cloud based system with a specialized database for scientific computing. For the experiments reported in this chapter, we use synthetic data, so that we can trace performance scaling with controlled, growing data sizes; synthetic datasets are similar to Encode peak datasets [22]. Datasets have the following features:

- The schema includes just a Score attribute. Chromosomes are 22, and each chromosome has 1 million bases.
- Regions in each chromosome are 2300, and they are randomly distributed over the chromosome space; their length is randomly distributed between 20 and 500 bases.

We then generate 5 datasets with an increasing number of samples (up to 20K) and regions (up to 1 billion); see Fig. 11.0.1. In the last section we also show experiments over real genomic datasets. We performed our experiments on the Amazon Web Services (AWS) cloud, using a configuration with r3.4xlarge machines; 16 cores, 122 GB of RAM and 320GB of SDD.

Dataset	Size (MByte)	Regions (Million)	Samples
REF	2.3	0.506	1
DS_1	3.5	0.1012	2
DS_2	38	1.012	20
DS_3	375	10.120	200
DS_4	3832	101.2	2000
DS_5	38232	1012	20000

Table 11.0.1: Features of the datasets used in the filtering operation.

Test	DS_2	DS_3	DS_4	DS_5
Spark *Q1*	4.391	6.063	9.403	43.645
SciDB *Q1*	0.110	0.136	0.385	4.515
Spark *Q2*	4.640	6.447	10.299	46.049
SciDB *Q2*	0.161	0.581	5.673	58.137
Spark *Q3*	0.123	0.140	0.284	2.035
SciDB *Q3*	4.478	6.145	9.813	44.015

Table 11.1.1: Execution times (in seconds) for the filter operation.

11.1 Regions Filtering

We start comparing how Spark and SciDB execute the filtering operations discussed in Section 3.1. We consider three selection predicates:

- `Q1: chr='chr1'`
- `Q2: score>0.9`
- `Q3: (chr='chr1') and (score>0.9)`

Execution times of the operations in Spark and SciDB are reported in Table 11.1.1, and graphically compared in Fig. 11.1.1. We note that execution times for SciDB on `Q1` and `Q3` are much smaller than on `Q2`; in the former cases SciDB exploits the `between` operator and outperforms Spark. In tt Q2, instead, SciDB must read each single cell in order to apply the filtering operation, and in such case the execution time is similar to Spark, and it actually becomes worse with increasing data sizes.

11.2 Region Aggregation

Execution times of region aggregation `Q4` in Spark and SciDB are reported in Table 11.2.1, and graphically compared in Fig. 11.2.1. In this case we observe a huge difference between the two

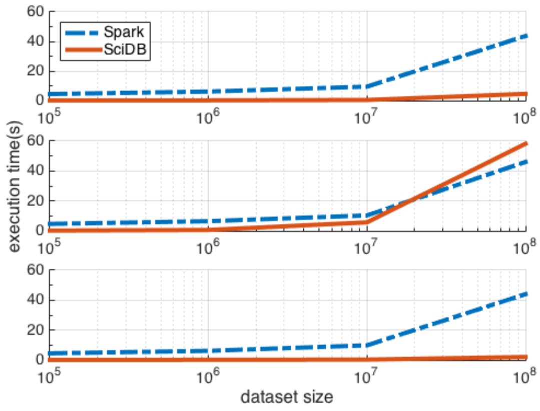


Figure 11.1.1: Execution times of region filtering in Spark and SciDB

Test	DS_2	DS_3	DS_4	DS_5
Spark Q4	10.667	18.730	29.094	133.938
SciDB Q4	0.155	0.169	0.307	1.747

Table 11.2.1: Execution times (in seconds) for the aggregation operation.

platforms performance: SciDB exploits the possibility to run in parallel the aggregation function in each chunk, and thus SciDB outperforms Spark.

11.3 Region Histogram

Execution times of region histogram Q5 in Spark and SciDB are reported in Table 11.3.1, and graphically compared in Fig. 11.3.1. Region histogram is encoded in a very similar way in SciDB and in Spark; hence, the overall perfornance in the two systems is quite similar, and there is no clear winner.

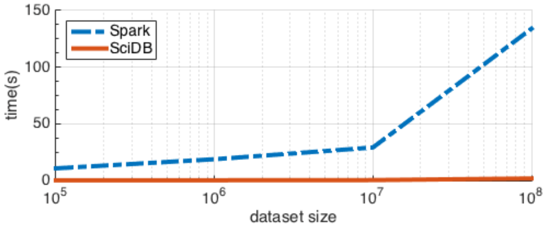


Figure 11.2.1: Execution times of region aggregation in Spark and SciDB

Test	DS_2	DS_3	DS_4	DS_5
Spark Q5	8.005	32.234	85.841	260.332
SciDB Q5	1.520	8.667	69.275	222.767

Table 11.3.1: Execution times (in seconds) for the aggregation operation

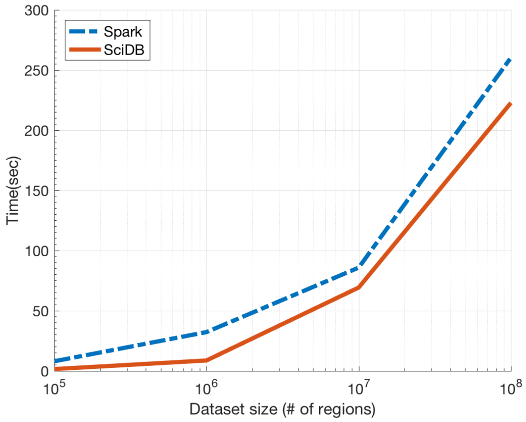


Figure 11.3.1: Execution times of region histogram in Spark and SciDB

11.4 Region Mapping

Execution times of region mapping Q6 in Spark and SciDB are reported in Table 11.4.1, and graphically compared in Fig. 11.4.1. Region mapping is an operation of **quadratic complexity**, similar to the join; hence, execution times are much higher (expressed in minutes). In this case, we note that Spark outperforms SciDB, whose performance rises to about 1.5 hrs when comparing .5 million regions of the reference with 101 million regions of 2000 experiments (note that this is a *very big data operation*, as it potentially requires 50 trillion comparisons). For this reason, and given the limitations of SciDB binning algorithms discussed in Section 3.D, we decided to focus on a new method for genome binning, that better adapts to the computational model of SciDB, discussed in the next section.

Test	DS_2	DS_3	DS_4
Spark Q6	0.12	0.57	3.82
SciDB Q6	0.28	3.29	95.33

Table 11.4.1: Execution times (in minutes) for the mapping operation.

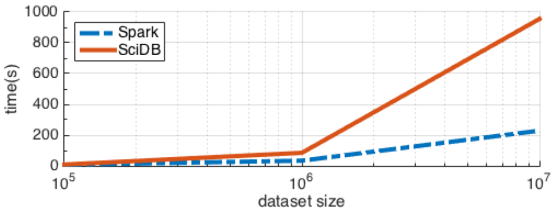


Figure 11.4.1: Excution times of region mapping in Spark and SciDB

11.4.1 Single Dimensional space binning

One of the most critical operation using a region based data model is the *interval intersection*. This procedure is frequently used in genomic analysis and its computation represent a complex problem working on big data. This section describes the optimization adopted by *Paradigm4* in order to improve this procedure.

Figure 11.4.2 shows how *Paradigm4* computes the region intersection by applying a binning strategy (they call the bins ”buckets”). Without presenting in deep the used code, we will summarize the strategy.

- *Step 1* - Compute the maximum length of the regions. This value will define the lower bound for the bins size. In this way we can be sure that each region falls at maximum into two different bins.

- *Step 2* - Each region is then duplicated joining the datasets with a synthetic 2 cells array. Then bin ids are assigned. To the first copy is selected the bin where falls the left end of the region, and for the second copy the bin where falls the right end. If the two values are the same, the second copy is dropped. At the end each region, duplicated if required, is marked with a single bin id.

- *Step 3* - Execute a `cross_join` between the two prepared datasets on same chromosome and same bin id. This will reduce the original cross product checking just the regions that fall in the same bin.

- *Step 4* - The intersection condition is then applied on the cross join result, selecting just the intersecting regions for each bin.

- *Step 5* - It's required a clean up step to avoid duplicates in the result. Looking at figure 11.4.2, the couple `(R2,E2)` is evaluated both for bin 1 and bin 2. To remove duplicates the procedure drops the pairs composed by two regions not starting in the current bin.

- *Step 6* - Finally, the result is produced simply merging the partial ones.

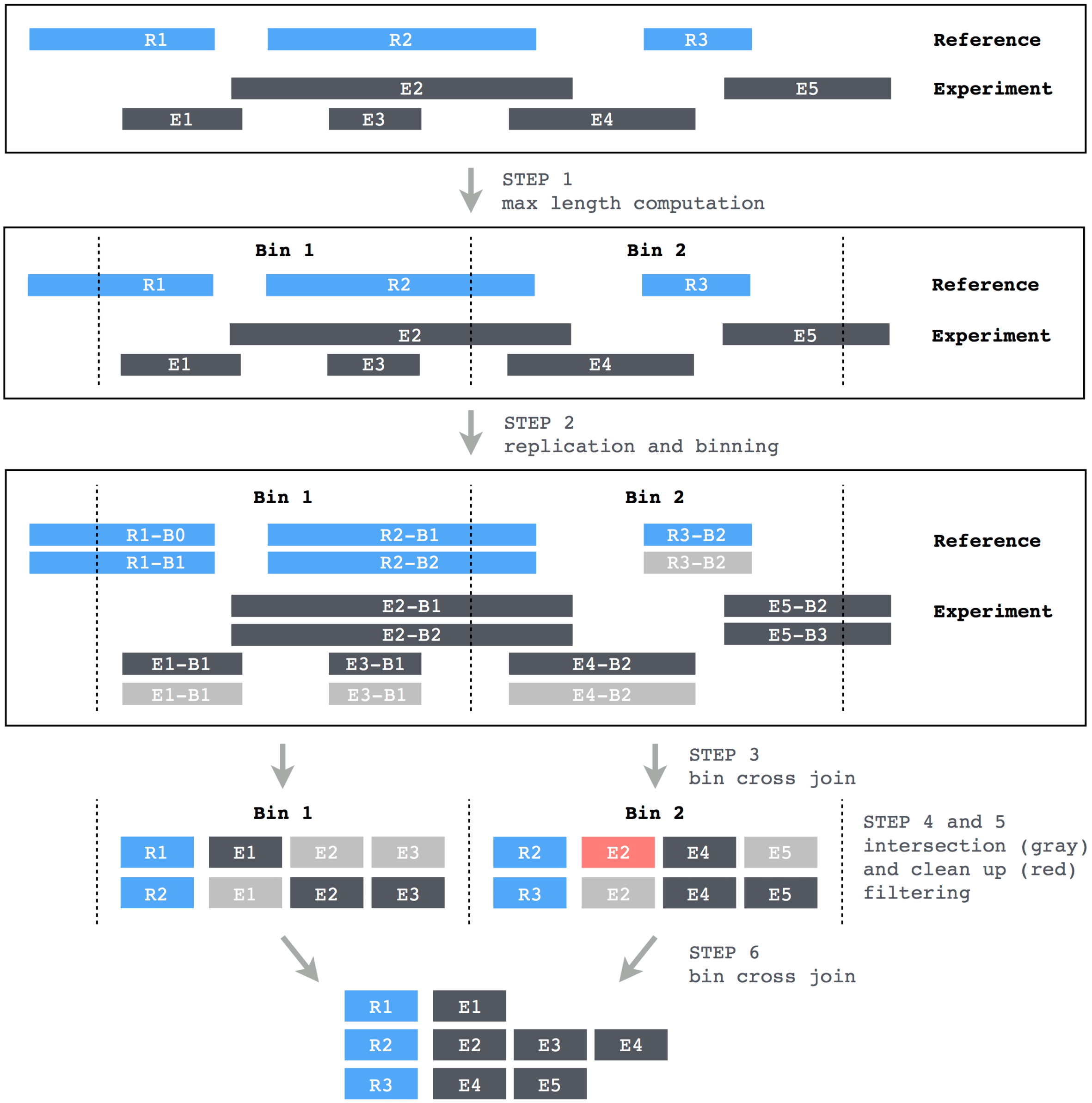


Figure 11.4.2: Binning strategy adopted by *Paradigm4* in the genomic add-on.

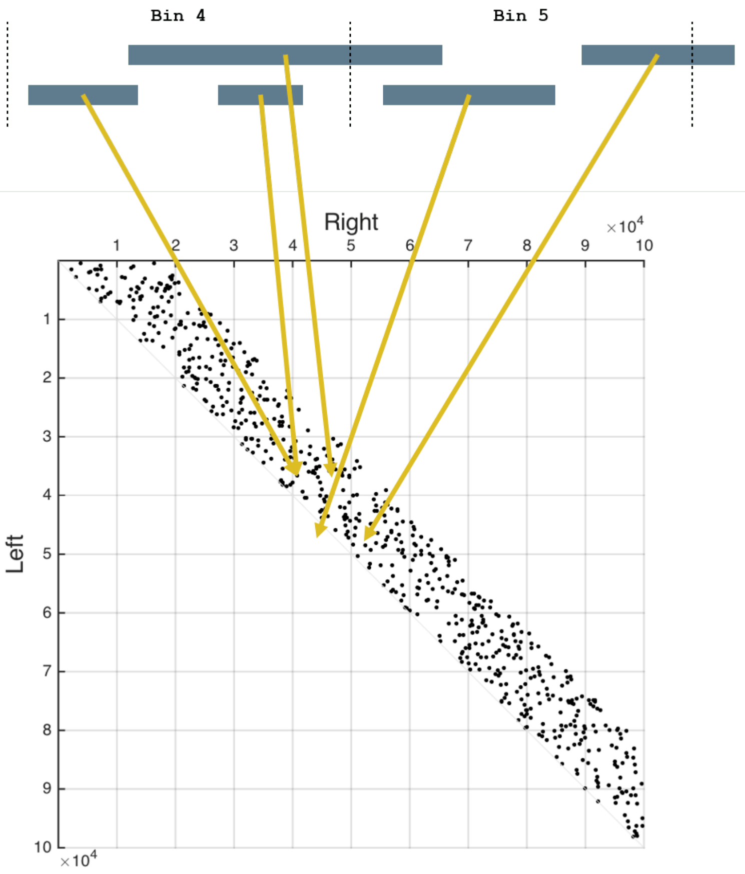


Figure 11.4.3: Region assignment to bins with bi-dimensional binning

Using the presented procedure, really distant regions are not evaluated, reducing the computation complexity of the operator. Nevertheless this method has two drawbacks: all data have to be replicated a fixed number of times, and the method is sensitive to the presence of really big regions that, however few, increase the bin size reducing the parallelism.

11.4.2 Bi-dimensional Binning

We propose Bi-dimensional binning as an alternative strategy to mono-dimensional binning, that exploits the array storage management of SciDB. As in mono-dimensional binning, the aim is to test for overlap just few pairs of regions, which could possibly intersect. In this approach, each region R is assigned to a bin defined by a pair of identifiers:

$$\texttt{bin(R)} = \left(\left\lfloor \frac{R_{start}}{\texttt{bin\_size}} \right\rfloor , \left\lfloor \frac{R_{stop}}{\texttt{bin\_size}} \right\rfloor \right)$$

A region is assigned to the (n, m) bin when it starts in the *n-th* bin and ends in the *m-th* bin (see Fig. 11.4.3); note that the genome is partitioned into a bi-dimensional grid rather than a mono-dimensional vector, and that every region is mapped to a point in such grid; since each region is constrained to have `start < stop`, a region can be assigned only to a cell either in the primary diagonal or above the diagonal of such space, and each region is assigned to exactly one bin. Note that in most genomic applications points tend to cluster either in the diagonal or in the cells immediately above the diagonal.

Consider now the mapping between a `Reference` and an `Experiment` dataset. For each bin in the reference we can divide the experiment regions into three groups: (a) regions that for sure intersect all the reference region in the bin, (b) regions that can potentially intersect them, and (c) regions that do not intersect them. By exploiting this partitioning, we can reduce the search space for each reference bin to a specific window in the experiment space, that includes just the regions that actually can intersect the regions of that bin.

Consider the bin $(2, 3)$ of the reference, i.e., regions that start in bin 2 and end in bin 3. Fig. 11.4.4 shows the regions of the experiment that certainly intersect with them, while Fig. 11.4.5 show regions of the experiment that could possibly intersect with them; their composition (taking into account that no region falls below the diagonal) generates a rectangular *target space* for the bin $(2, 3)$ of the reference, shown in Fig. 11.4.6.

The bi-dimensional binning strategy for region mapping is illustrated in Fig. 11.4.7. In this approach, several independent queries are executed, one for each non-empty bin of the reference (the figure shows theee such queries). For a given bin of the reference, an AFL query computes range intersections with all the regions of the experiment which belong to the bin's target space; the result is an aggregate value, associated to the regions of the reference bin.

A block diagram sketch for the algorithm is shown in Fig. 11.4.8. Initially, the query builds the bin of the reference and the target space of the experiments using the `between` operator, which is efficiently applied to the regions' dimensions. Then, execution continues in a similar way as discussed in Section 3.D; thus, a `cross_join` is performed, intersecting regions are extracted and grouped, and aggregate functions are computed.

11.5 Comparative Evaluation of D1, D2 Binning Strategies with Spark binning

As first evaluation of bidimensional binning, we consider again query `Q6` of Section 4.D; execution times are reported in Table 11.5.1, and graphically compared in Fig. 11.5.1; note that, when executed over the dataset `DS_4`, bidimensional binning improves of about 3X over monodimensional

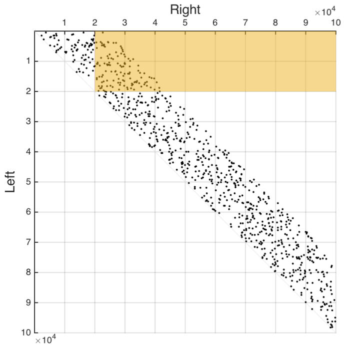


Figure 11.4.4: Experiment regions that intersect with regions in bin (2,3)

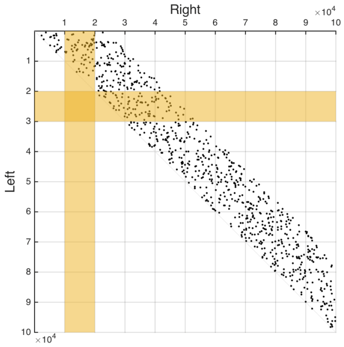


Figure 11.4.5: Experiment regions that can intersect with regions in bin (2,3)

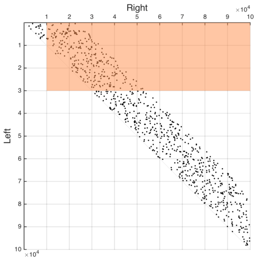


Figure 11.4.6: Target space for the regions of bin (2,3)

Test	DS_2	DS_3	DS_4
Spark *Q6*	0.12	0.57	3.82
SciDB_D2 *Q6*	0.43	2.10	28.49
SciDB_D1 *Q6*	0.28	3.29	95.33

Table 11.5.1: Execution times (in minutes) for the mapping operation.

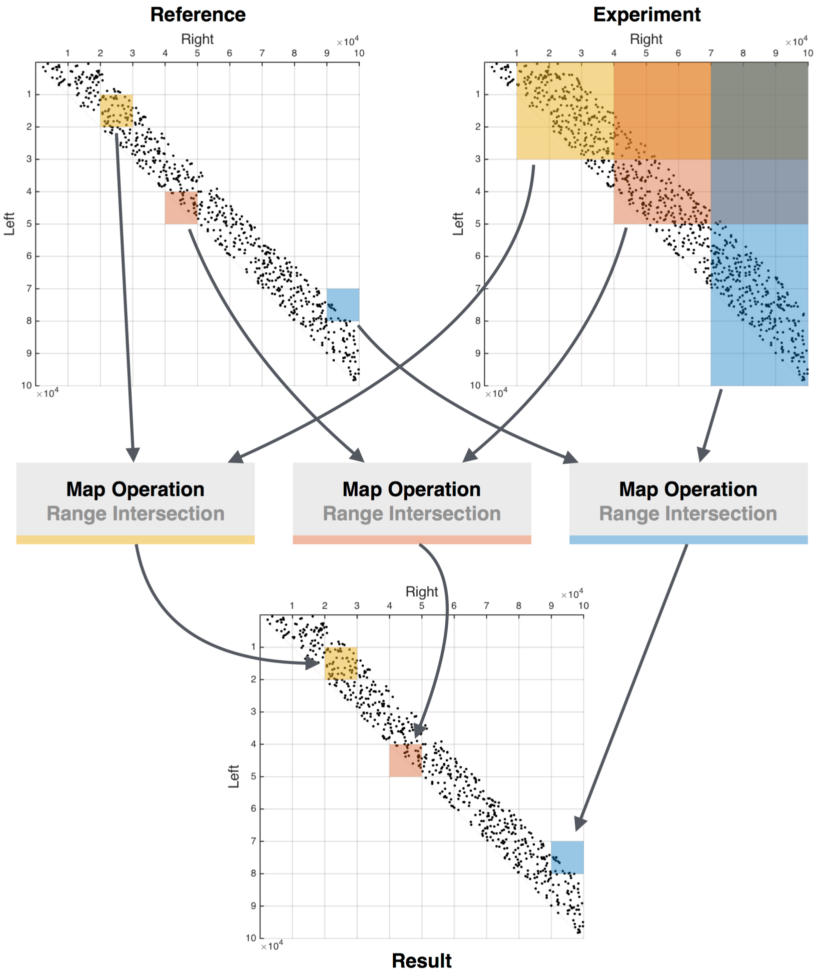
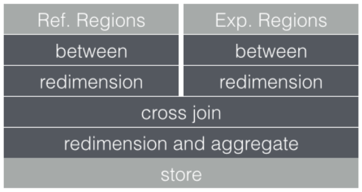


Figure 11.4.7: Bi-dimensional binning strategy



Ref. Regions | Exp. Regions

between | between

redimension | redimension

cross join

redimension and aggregate

store

Figure 11.4.8: Block representation of region mapping using bi-dimensional binning in SciDB

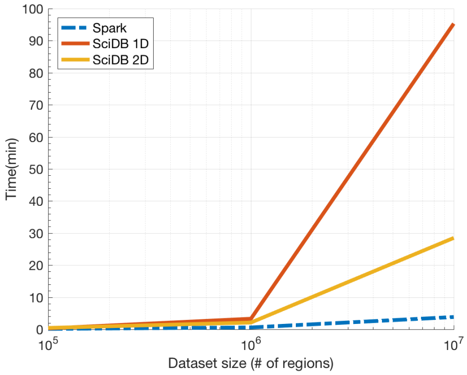


Figure 11.5.1: Comparison of Spark and SciDB with two binning strategies

Dataset	Size (MByte)	Regions (Thousands)	Samples
GENES	0.7	23.033	1
PROMOTERS	2.3	49.052	1
NP_1	17	363.537	2
NP_2	41	938.753	4
NP_3	57	1264.764	8
NP_4	108	22300.698	16

Table 11.5.2: Features of the real datasets used in the map operation.

binning, covering part of the difference in performance between SciDB and Spark.

In order to better evaluate bi-dimensional binning, we then considered real datasets. For the references, we considered two different types of regions:

- **Genes** are heterogeneous regions, as their maximum length is 24187702, their minimum length is 19, their average length is 60680, and their median length is 20102. This length variability could negatively affect mono-dimensional binning.

- **Promoters** are small homogeneous regions, each of size 2999, artificially built around a specific genomic position, the *transcription start site* [1].This lenght regularity could instead favor mono-dimensional binning.

The experiments datasets are collected from encode Narrow Peaks (NP) with different sizes, as shown in 11.5.2.

Performance comparisons are shown in Fig. 11.5.2 and 11.5.3. Note that in all cases Spark outperforms SciDB, but the difference between Spark and SciDB with bidimensional binning is much

[1] These regions are biologically relevant as they contain genomic information that is most relevant to RNA transcription.

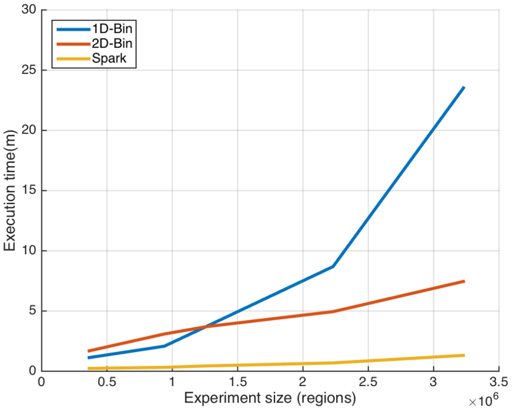


Figure 11.5.2: Performance comparison using genes as reference

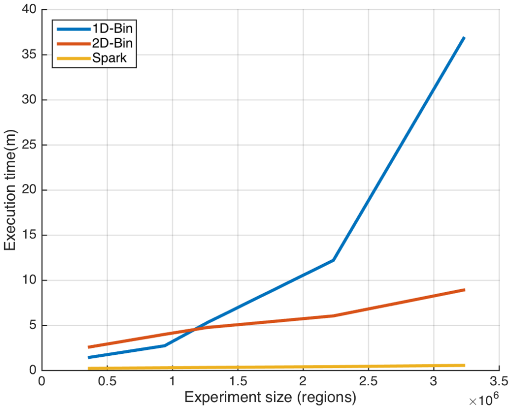


Figure 11.5.3: Performance comparison using promoters as reference

reduced, and that Spark and SciDB curves scale in a similar way. Note as well that bidimensional bidding always outperforms monodimensinal binning; the two curves are however find with different replication factors for monodimensional binning, as replication is set to 8 for genes and to 2 for promoters. This choice of the best replication factor was obtained after several experiments and shows that, with suitable tuning, monodimensional binning can be adapted to differences in reference regions.

Although a large number of benchmarks exist for comparing general purpose cloud-based engines such as Spark and Flink, including academic articles ([61]) and posts[2], we are not aware of benchmarks comparing these engines with array-based scientific databases, such as SciDB or Vertica. We shows that this benchmark has no clear winner; as expected, SciDB performs better when it ben-

[2] http://sparkbigdata.com/102-spark-blog-slim-baltagi/14-results-of-a-benchmark-between-apache-flink-and-apache-spark.

efits from the array-based database organization (hence, on region filtering and aggregation), while Spark performs better on massive region mapping operations (similar to joins). The histogram operation, that does not fall in either categories, has very similar performances in SciDB and Spark. We also present an original binning strategy, called bi-dimensional binning, and showed that such strategy outperforms the conventional mono-dimensional binning strategy used by SciDB, reducing the gap in performance between SciDB and Spark. Bi-dimensional binning can be used for parallelizing any interval-based computation and therefore has wide applicability, which goes beyond genomics.

In literature and in life we ultimately pursue, not conclusions, but beginnings.

Sam Tanenhaus

12
Conclusion and Future work

We described GMQL as a new paradigm for data-Centric genomic computing; we described the system architecture of GMQL and its evolution from V1 to V2, and we discussed the advantages of V2 over V1, mostly due to its modular architecture. We recently deployed our system in a public network at Cineca[1]. We described domain-specific operations of GMQL; we introduced binning as a general approach to the parallelization of genomic operations. We identified the trade-offs due to the bin dimensions; we proved that, although multiple bins may carry the result of region comparison, the result can be safely extracted from a single bin; and we explained how the binning logic is implemented by using Flink, Spark and SciDB. Experiments demonstrate that the bin size is a critical parameter for the overall performance of domain-specific operations.

We also showed the scaling of performance in a multi-node cloud computing network; adding computing power increments the performance but at an increasing cost per sample; moreover, above a given threshold, an increase in the number of nodes may cause a loss in performance, given the complexity of multi-node computation.

Our project's architectural choice, which includes a portable GMQL implementation to SciDB, Spark and Flink, appears well motivated by our benchmarks; we believe that supporting various implementation engines will be a key feature of our genomic data management project in the long run, as we will be able to match application requirements to the best target system and to closely follow

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

the evolution of cloud-based platforms.

The experiments demonstrate that domain-specific GMQL operations scale extremely well when challenged by very large datasets; therefore, GMQL is an ideal formalism to cope with big queries of today's and tomorrow's genomic computing. We are using GMQL in advanced biological research, for understanding how *topological domains*, i.e. recently discovered functional subdivisions of the genome, include genes which are highly expressed in either normal or tumor cells. This problem is addressed by a very simple GMQL program over the TCGA big data repository, focused on cancer; a complete pipeline iterating over 20 tissues and normal vs tumor types runs in about 2 hours.

For future work, we will develop methods for dynamic assignment of the bin size for Map, Join and Cover operations, based on the input data profiling. we will also work on the management of GMQL clusters, where each cluster contains different implementation of GMQL (Spark, Flink, SciDB), coordinated by a master GMQL application machine. GMQL master application will choose the implementation based on the profiling of GMQL code provided by the user. GDM data would be distributed on several clusters and GMQL master application will optimize the processing, at the same time by minimizing the data migration.

For longer-term, we are planning to turn GMQL into an i ncubated project within Apache, so as to provide a strong community of users and developers. We also plan to deliver custom services and to provide access to a large repository of public data, constructed by integrating and curating data from ENCODE [21], TCGA [27], 1000 Genomes Project [76] and other sources.

References

[1] Dna is a structure that encodes biological information. http://www.nature.com/scitable/topicpage/dna-is-a-structure-that-encodes-biological-6493050. Accessed: 2016-08-05.

[2] Francis HC Crick et al. The origin of the genetic code. *Journal of molecular biology*, 38(3):367–379, 1968.

[3] SA Langeveld, AD van Mansfeld, PD Baas, HS Jansz, GA Van Arkel, and PJ Weisbeek. Nucleotide sequence of the origin of replication in bacteriophage phix174 rf dna. *Nature*, 271(5644):417–420, 1978.

[4] Eric S Lander, Lauren M Linton, Bruce Birren, Chad Nusbaum, Michael C Zody, Jennifer Baldwin, Keri Devon, Ken Dewar, Michael Doyle, William FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[5] Stephan C Schuster. Next-generation sequencing transforms today's biology. *Nature methods*, 5(1):16–18, 2008.

[6] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nature biotechnology*, 26(10):1135–1145, 2008.

[7] Jorge S Reis-Filho et al. Next-generation sequencing. *Breast Cancer Res*, 11(Suppl 3):S12, 2009.

[8] Michael Eisenstein. Big data: The power of petabytes. *Nature*, 527(7576):S2–S4, 2015.

[9] Daniel Summerer. Enabling technologies of genomic-scale sequence enrichment for targeted high-throughput sequencing. *Genomics*, 94(6):363–368, 2009.

[10] Elaine R Mardis. The impact of next-generation sequencing technology on genetics. *Trends in genetics*, 24(3):133–141, 2008.

[11] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, 11(5):473–483, 2010.

[12] Yong Zhang, Tao Liu, Clifford A Meyer, Jérôme Eeckhoute, David S Johnson, Bradley E Bernstein, Chad Nusbaum, Richard M Myers, Myles Brown, Wei Li, et al. Model-based analysis of chip-seq (macs). *Genome Biol*, 9(9):R137, 2008.

[13] Zhong Wang, Mark Gerstein, and Michael Snyder. Rna-seq: a revolutionary tool for transcriptomics. *Nature Reviews Genetics*, 10(1):57–63, 2009.

[14] Lingyun Song and Gregory E Crawford. Dnase-seq: a high-resolution technique for mapping active gene regulatory elements across the genome from mammalian cells. *Cold Spring Harbor Protocols*, 2010(2):pdb–prot5384, 2010.

[15] Ron Edgar, Michael Domrachev, and Alex E Lash. Gene expression omnibus: Ncbi gene expression and hybridization array data repository. *Nucleic acids research*, 30(1):207–210, 2002.

[16] Tanya Barrett, Dennis B Troup, Stephen E Wilhite, Pierre Ledoux, Dmitry Rudnev, Carlos Evangelista, Irene F Kim, Alexandra Soboleva, Maxim Tomashevsky, and Ron Edgar. Ncbi geo: mining tens of millions of expression profiles—database and tools update. *Nucleic acids research*, 35(suppl 1):D760–D765, 2007.

[17] Gene expression omnibus. http://www.ncbi.nlm.nih.gov/geo/. Accessed: 2016-08-05.

[18] Alvis Brazma, Pascal Hingamp, John Quackenbush, Gavin Sherlock, Paul Spellman, Chris Stoeckert, John Aach, Wilhelm Ansorge, Catherine A Ball, Helen C Causton, et al. Minimum information about a microarray experiment (miame)—toward standards for microarray data. *Nature genetics*, 29(4):365–371, 2001.

[19] ENCODE Project Consortium et al. The encode (encyclopedia of dna elements) project. *Science*, 306(5696):636–640, 2004.

[20] Ewan Birney, John A Stamatoyannopoulos, Anindya Dutta, Roderic Guigó, Thomas R Gingeras, Elliott H Margulies, Zhiping Weng, Michael Snyder, Emmanouil T Dermitzakis, Robert E Thurman, et al. Identification and analysis of functional elements in 1% of the human genome by the encode pilot project. *Nature*, 447(7146):799–816, 2007.

[21] Encode project. http://www.genome.gov/encode/. Accessed: 2016-08-05.

[22] ENCODE Project Consortium et al. An integrated encyclopedia of dna elements in the human genome. *Nature*, 489(7414):57–74, 2012.

[23] 1000 genomes project website. http://www.1000genomes.org/. Accessed: 2015-08-05.

[24] Nayanah Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–256, 2008.

[25] Global alliance for genomics and health. http://genomicsandhealth.org/. Accessed: 2016-08-05.

[26] Ga4gh data working group. http://ga4gh.org/. Accessed: 2016-08-05.

[27] The cancer genome atlas website. http://cancergenome.nih.gov/. Accessed: 2016-08-05.

[28] John N Weinstein, Eric A Collisson, Gordon B Mills, Kenna R Mills Shaw, Brad A Ozenberger, Kyle Ellrott, Ilya Shmulevich, Chris Sander, Joshua M Stuart, Cancer Genome Atlas Research Network, et al. The cancer genome atlas pan-cancer analysis project. *Nature genetics*, 45(10):1113–1120, 2013.

[29] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[30] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[32] Apache flink. https://flink.apache.org/. Accessed: 2016-08-15.

[33] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[34] Apache lucene. https://lucene.apache.org/core/. Accessed: 2016-08-05.

[35] Paradigm4 Inc. SciDB MAC Storage Explained, 2015. Downloaded on April 2016.

[36] Stefano Ceri, Abdulrahman Kaitoua, Marco Masseroli, Pietro Pinoli, and Francesco Venco. Data management for heterogeneous genomic datasets. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2016.

[37] Stefano Ceri, Abdulrahman Kaitoua, Pietro Pinoli, and Marco Masseroli. Genomic data modeling for interoperability and next generation genomic data management. In *Proceedings of the 4th International Work-Conference on Bioinformatics and Biomedical Engineering*, pages 20–22. WBBIO, 2016.

[38] Stefano Ceri, Abdulrahman Kaitoua, Marco Masseroli, Pietro Pinoli, and Francesco Venco. Data management for heterogeneous genomic datasets. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2016.

[39] Marco Masseroli, Pietro Pinoli, Francesco Venco, Abdulrahman Kaitoua, Vahid Jalili, Fernando Palluzzi, Heiko Muller, and Stefano Ceri. Genometric query language: a novel approach to large-scale genomic data management. *Bioinformatics*, 31(12):1881–1888, 2015.

[40] Suhas SP Rao, Miriam H Huntley, Neva C Durand, Elena K Stamenova, Ivan D Bochkov, James T Robinson, Adrian L Sanborn, Ido Machol, Arina D Omer, Eric S Lander, et al. A 3d map of the human genome at kilobase resolution reveals principles of chromatin looping. *Cell*, 159(7):1665–1680, 2014.

[41] William A Flavahan, Yotam Drier, Brian B Liau, Shawn M Gillespie, Andrew S Venteicher, Anat O Stemmer-Rachamimov, Mario L Suvà, and Bradley E Bernstein. Insulator dysfunction and oncogene activation in idh mutant gliomas. *Nature*, 529(7584):110–114, 2016.

[42] John N Weinstein, Eric A Collisson, Gordon B Mills, Kenna R Mills Shaw, Brad A Ozenberger, Kyle Ellrott, Ilya Shmulevich, Chris Sander, Joshua M Stuart, Cancer Genome Atlas Research Network, et al. The cancer genome atlas pan-cancer analysis project. *Nature genetics*, 45(10):1113–1120, 2013.

[43] John Lonsdale, Jeffrey Thomas, Mike Salvatore, Rebecca Phillips, Edmund Lo, Saboor Shad, Richard Hasz, Gary Walters, Fernando Garcia, Nancy Young, et al. The genotype-tissue expression (gtex) project. *Nature genetics*, 45(6):580–585, 2013.

[44] Bed format. https://genome.ucsc.edu/FAQ/FAQformat. Accessed: 2016-08-15.

[45] Eleonora Cappelli Stefano Ceri Marco Masseroli Fabio Cumbo, Giulia Fiscon1 and Emanuel Weitschek. Tcga2bed: extracting, extending, integrating, and querying the cancer genome atlas. *BMC Bioinformatics (under revision)*, 2016.

[46] Apache Spark. Apache spark™-lightning-fast cluster computing, 2014.

[47] Apache knox. https://knox.apache.org/. Accessed: 2016-08-05.

[48] Livy engine. http://livy.io/index.html. Accessed: 2016-08-05.

[49] Ganglia monitoring system. http://ganglia.info/. Accessed: 2016-08-15.

[50] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[51] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[52] Apache hadoop. http://hadoop.apache.org/. Accessed: 2016-08-05.

[53] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, pages 29–43. ACM, 2003.

[54] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[55] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning in the cloud. *HotCloud*, 9:12, 2009.

[56] Shrinivas B Joshi. Apache hadoop performance-tuning methodologies and best practices. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 241–242. ACM, 2012.

[57] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1357–1369. ACM, 2015.

[58] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[59] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[60] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[61] Norman Spangenberg, Martin Roth, and Bogdan Franczyk. Evaluating new approaches of big data analytics frameworks. In *International Conference on Business Information Systems*, pages 28–37. Springer, 2015.

[62] Philippe Cudré-Mauroux, Hideaki Kimura, K-T Lim, Jennie Rogers, Roman Simakov, Emad Soroush, Pavel Velikhov, Daniel L Wang, Magdalena Balazinska, Jacek Becla, et al. A demonstration of scidb: a science-oriented dbms. *Proceedings of the VLDB Endowment*, 2(2):1534–1537, 2009.

[63] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. Scidb: A database management system for applications with complex analytics. *Computing in Science & Engineering*, 15(3):54–62, 2013.

[64] Racket translator. http://racket-lang.org/. Accessed: 2016-08-15.

[65] Frank DeRemer and Thomas Pennello. Efficient computation of lalr (1) look-ahead sets. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):615–649, 1982.

[66] Jeremy Goecks, Anton Nekrutenko, James Taylor, et al. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol*, 11(8):R86, 2010.

[67] Jelle Scholtalbers, Jasmin Rößler, Patrick Sorn, Jos de Graaf, Valesca Boisguérin, John Castle, and Ugur Sahin. Galaxy lims for next-generation sequencing. *Bioinformatics*, page btt115, 2013.

[68] General transfer format. http://www.ensembl.org/info/website/upload/gff.html. Accessed: 2016-08-15.

[69] Netty as an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers and clients. http://netty.io/. Accessed: 2016-08-05.

[70] Play framework as high velocity web framework for java and scala. https://www.playframework.com/. Accessed: 2016-08-05.

[71] Leonard Richardson and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[72] Graphviz: a graph virtualizaiton software. http://www.graphviz.org/. Accessed: 2016-08-05.

[73] Francesco Venco, Yuriy Vaskin, Arnaud Ceol, and Heiko Muller. Smith: A lims for handling next-generation sequencing workflows. *BMC bioinformatics*, 15(Suppl 14):S3, 2014.

[74] Variant call format. http://www.1000genomes.org/wiki/Analysis/variant-call-format/. Accessed: 2016-08-15.

[75] Mongodb. https://www.mongodb.com. Accessed: 2016-08-15.

[76] 1000 Genomes Project Consortium et al. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

[77] Casey E Romanoski, Christopher K Glass, Hendrik G Stunnenberg, Laurence Wilson, and Genevieve Almouzni. Epigenomics: Roadmap for regulation. *Nature*, 518(7539):314–316, 2015.

[78] W3c web services glossary.

[79] Jason H Christensen. Using restful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 627–634. ACM, 2009.

[80] Apache tomcat. http://tomcat.apache.org/. Accessed: 2016-08-05.

[81] North east italian interuniversity consortium for automatic computation. http://www.cineca.it/en. Accessed: 2016-08-05.

[82] Line sweep algorithms. https://www.topcoder.com/community/data-science/data-science-tutorials/line-sweep-algorithms/. Accessed: 2016-08-15.

[83] Shane Neph, M. Scott Kuehn, Alex P. Reynolds, Eric Haugen, Robert E. Thurman, Audra K. Johnson, Eric Rynes, Matthew T. Maurano, Jeff Vierstra, Sean Thomas, Richard Sandstrom, Richard Humbert, and John A. Stamatoyannopoulos. Bedops: High performance genomic feature operations. *Bioinformatics*, 2012.

[84] Aaron R. Quinlan and Ira M. Hall. Bedtools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, 2010.

[85] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[86] Ucsc genome browser. https://genome.ucsc.edu/index.html. Accessed: 2016-08-05.

[87] Donna Karolchik, Robert Baertsch, Mark Diekhans, Terrence S Furey, A Hinrichs, YT Lu, Krishna M Roskin, M Schwartz, Charles W Sugnet, Daryl J Thomas, et al. The ucsc genome browser database. *Nucleic acids research*, 31(1):51–54, 2003.

[88] Generic model organism database. http://www.gmod.org. Accessed: 2016-08-05.

[89] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, et al. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[90] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. *University of Maryland*, 1987.

[91] Henrik Nordberg, Karan Bhatia, Kai Wang, and Zhong Wang. Biopig: a hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, page btt528, 2013.

[92] André Schumacher, Luca Pireddu, Matti Niemenmaa, Aleksi Kallio, Eija Korpelainen, Gianluigi Zanetti, and Keijo Heljanko. Seqpig: simple and scalable scripting for large sequencing data sets in hadoop. *Bioinformatics*, 30(1):119–120, 2014.

[93] Matti Niemenmaa, Aleksi Kallio, André Schumacher, Petri Klemelä, Eija Korpelainen, and Keijo Heljanko. Hadoop-bam: directly manipulating next generation sequencing data in the cloud. *Bioinformatics*, 28(6):876–877, 2012.

[94] Marek S Wiewiórka, Antonio Messina, Alicja Pacholewska, Sergio Maffioletti, Piotr Gawrysiak, and Michał J Okoniewski. Sparkseq: fast, scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, page btu343, 2014.

[95] Christos Kozanitis, Andrew Heiberg, George Varghese, and Vineet Bafna. Using genome query language to uncover genetic variation. *Bioinformatics*, 30(1):1–8, 2014.

[96] Samir Tata, Joseph S Friedman, and Abhishek Swaroop. Declarative querying for biological sequences. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 87–87. IEEE, 2006.

[97] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, André Schumacher, Anthony D Joseph, and David A Patterson. Adam: Genomics formats and processing patterns for cloud scale computing. *University of California, Berkeley Technical Report, No. UCB/EECS-2013*, 207, 2013.

[98] Luca Pireddu, Simone Leo, and Gianluigi Zanetti. Mapreducing a genomic sequencing workflow. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 67–74. ACM, 2011.

[99] Kristian Ovaska, Lauri Lyly, Biswajyoti Sahu, Olli A Janne, and Sampsa Hautaniemi. Genomic region operation kit for flexible processing of deep sequencing data. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 10(1):200–206, 2013.