POLITECNICO DI MILANO
School of Industrial and Information Engineering
Master of Science in Computer Science and Engineering

Department of Electronics, Information and Bioengineering

A PYTHON DATA ANALYSIS LIBRARY FOR GENOMICS AND ITS APPLICATION TO BIOLOGY

Laboratory of Bioinformatics and Web Engineering

Supervisor: Prof. Stefano Ceri
Co-supervisor: Dott. Pietro Pinoli
Second supervisor (Politecnico di Torino): Elena Baralis

Master Thesis of:
Luca Nanni, Matr 850113

Anno Accademico 2016-2017

To my parents

Abstract

Genomics is the study of all the elements that compose the genetic material within an organism. The new DNA sequencing technologies (NGS) have opened new lines of research, which include the study of diseases like cancer or genetic conditions. The huge amount of data produced by these new methods makes the genomic data management one of the current biggest big data problems.

In this context, GMQL, a declarative language built on top of big data technologies, was developed by the Bioinformatics group at Politecnico di Milano.

The first aim of this thesis is to enlarge the scope of this language by designing and implementing a Python package capable of interfacing with the big data engine, to extract the results and convert them in a useful data structure and to give the user the possibility to work with GMQL in a full interactive environment. The package will be able to perform computations both on the local machine and using a remote GMQL server and will interface with the main data science and machine learning python packages.

The second focus of this work is on applying the developed package to concrete biological problems. In particular we will concentrate on the study of Topologically Associating Domains (TADs), which are genomic regions within which the physical interactions occur much more frequently than out of them. We will use the package to analyze these data, extract new knowledge about them and derive their physical properties.

Sommario

La genomica è lo studio di tutti gli elementi che compongono il materiale genetico di un organismo. Le nuove tecnologie di sequenziamento del DNA (NGS) hanno aperto, negli ultimi anni, nuove frontiere di ricerca fra cui lo studio dei tumori e delle malattie ereditarie su larga scala. La grande quantità di dati prodotti da questi nuovi metodi rendono la gestione di dati genomici uno dei problemi più grandi nel campo dei big data.

In questo contesto, presso il laboratorio di bioinformatica e web engineering del Politecnico di Milano è stato sviluppato il sistema GMQL, che consiste in un linguaggio dichiarativo il quale, utilizzando tecnologie di processamento big data, permette l'esecuzione di query genomiche su grandi moli di dati.

Lo scopo primario di questa tesi è di estendere questo linguaggio attraverso il design e l'implementazione di una libreria python che possa interfacciarsi con il sistema di processamento dati, estrarre i dati di interesse, convertirli in una struttura dati efficiente e rappresentativa e infine fornire un ambiente di sviluppo interattivo. La libreria sarà anche in grado di eseguire localmente o tramite l'utilizzo di un cluster remoto, nonché potrà interfacciarsi con le principali librerie python di data science e machine learning.

In secondo luogo, la libreria verrà testata direttamente nella risoluzione di complessi problemi biologici. In particolare verrà posta l'attenzione sullo studio dei domini topologici, i quali sono particolari regioni genomiche caratterizzate da un aumentata densità di interazioni fisiche. Durante lo studio si utilizzerà la libreria per analizzare i dati ed estrarne nuova conoscenza al fine di derivare le proprietà fisiche di queste particolari regioni.

Ringraziamenti

Naturalmente non posso che iniziare i miei ringraziamenti ringraziando il mio relatore Stefano Ceri e il mio corelatore Pietro Pinoli, che mi hanno seguito, incoraggiato e sempre sostenuto in questo percorso. Un forte ringraziamento anche a tutto il gruppo di bioinformatica presso il quale ho lavorato in questi ultimi mesi e in particolare Arif e Abdo, che mi hanno sempre aiutato in caso di bisogno. Ringrazio inoltre Michele, Andrea, Ilaria B., Ilaria R., Eirini.

Un ringraziamento particolare ai ragazzi del gruppo dell'Alta Scuola Politecnica, che mi hanno seguito e sopportato in questi due anni di lavoro: Gianluca, Paolo, Diletta, Flavio e Antonio.

Un particolare ringraziamento naturalmente va a Francesco, Riccardo, Filippo, Lorenzo, Andrea, Mirco, Giacomo, Francesca, Lisa, Ezia, senza i quali probabilmente tutto quello che ho fatto finora non avrebbe significato.

Un grazie a i miei coinquilini, che in questi due anni mi hanno sopportato e hanno riso con me: Ale, Lorenzo ed Elia.

E come nei più classici ringraziamenti, ma soprattutto perché è la verità, il più grande ringraziamento va a tutta la mia famiglia, che mi ha sostenuto in tutto e per tutto da sempre.

E (perchè no?) un grazie anche a me stesso, che ha sempre guardato avanti e cercato di migliorarsi ad ogni passo lungo questa difficile, ma bellissima esperienza.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Frankly, my dear, I don't give a damn."

Gone with the wind

1.1 Genomics and the Human Genome Project

With the term *Genomics* we mean the study of all the elements that compose the genetic material within an organism [21].

The most important chemical structure studied in genomics is the Deoxyribonucleic acid (DNA). This chemical compound contains all the information needed to guide, develop and support the life and development of all the living organisms. DNA molecules are composed of two twisting paired strands. Each strand is made of four units called *nucleotide bases* (**A**denine, **T**imine, **G**uanine, **C**ytosine): these bases pair each other on the two strands following specified rules (**A - T**, **C - G**). The information is encoded in the DNA by the order in which these bases are placed in the molecule.

A *gene* is a portion of DNA that is taught to carry the information used to build a (set of) protein(s). *Proteins* are large molecules having a lot of different functions and one of the most important molecular structure for living organisms.

The genes are packed into larger structures called *chromosomes*. These big sections of DNA are paired together: for example, in the human organism we have 23 pairs of chromosomes.

In figure 1.1 we can see the full structure of the DNA.

The study of genes and their role on the expression of human traits is one of the main problems of modern medical research. Genes and their mutations

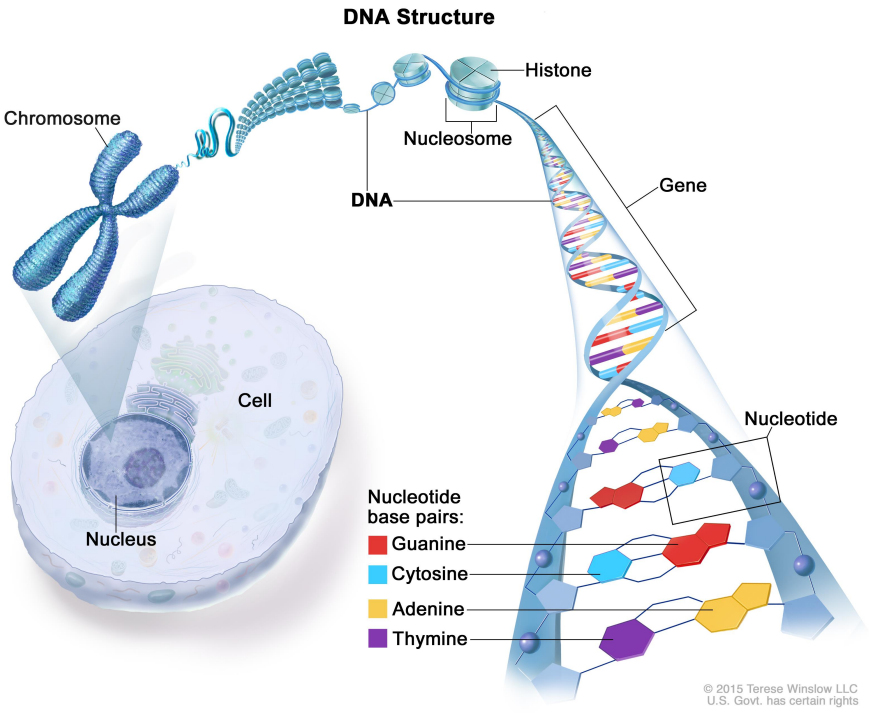


Figure 1.1: The DNA structure. Taken from [14]

have been proven to be at the base of several diseases. In particular in these years the subfield of *cancer genomics* is getting a lot of attentions. Computations biologists, geneticists and cancer researchers are looking at the genome level in order to understand the low level interactions leading to the development of tumors.

These new lines of research are now possible thanks to the new technologies for *DNA sequencing*. Sequencing the DNA means to determine the exact order of the bases in a strand of it [21]. From the 1990 to 2003 an international project called *Human Genome Project* was carried out with the aim to develop sequencing techniques for the full mapping of the human genome (see figure 1.2 for a time-line of the project).

The project was a success and in the first decade of the new millennium a new sequencing paradigm was developed, called *Next Generation Sequencing*. NGS enabled high-throughput, high-precision and low-cost sequencing making the DNA mapping a standardized process.

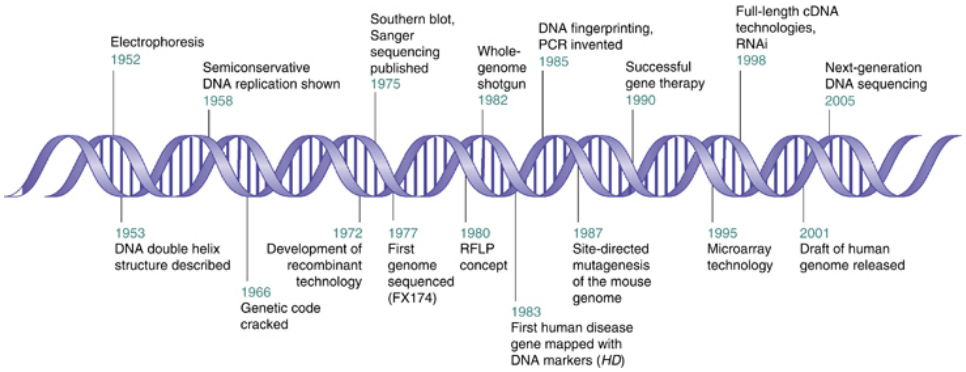


Figure 1.2: Time-line of the Human Genome Project. Taken from [4]

1.2 Big data in biology and genomics

NGS technologies and the success of the human genome project opened a new era in genomics. Never before such a great amount of data was publicly available for conducting biological research.

Of course the high volume of data produces a wide range of technical problems:

- *Storage*: data must be stored in public accessible databases
- *Heterogeneity*: data can have a potentially infinite number of formats
- *Computational power*: analyzing the data and applying algorithms to them must take into account their cardinality

These problems and the requirements that they produce configure the analysis of genomic data as one of the biggest and most important current big data problem.

The analysis of genomic data can be divided in three main categories, as explained in figure 1.3. *Primary* analysis deals with hardware generated data like sequence reads. It is the first mandatory step for extracting information from biological samples. *Secondary* analysis takes the reads of the previous step and filters, aligns and assembles them in order to find relevant features. *Tertiary* analysis works at the highest level of the pipeline and it is dedicated to the integration of heterogeneous data sources, multi-sample processing or enrichment of the data with external features.

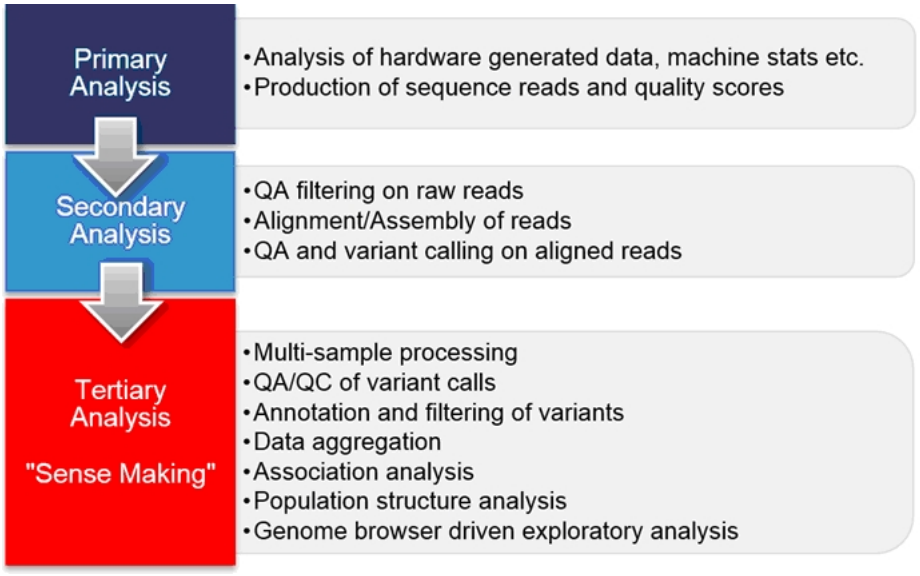


Figure 1.3: Differences between Primary, Secondary and Tertiary analysis. Taken from [20]

Figure 1.4: The role of GMQL in the genomic data analysis pipeline. Taken from [16]

1.3 Motivations and requirements

In this thesis we will address the problem of organizing, explore and analyze tertiary genomic data. In particular we will implement and use a **python package** for manipulating such data. We will design this library on the basis of the already existent **GMQL** system, which is a big genomic data engine developed at the Department of Electronics, Information and Bioengineering of the Politecnico of Milano (see figure 1.4). This python package will be validated through testing and also by applying it to a specific biological research problem. This will demonstrate the quality of the product, its usefulness and its applicability to a wide range of problems.

Biologists and researchers in genomics usually organize their work-flow in *pipelines*. A pipeline is a chain of operations applied to a set of data. Usually researchers work with files locally in their computer or in a local server and use the most common programming languages or frameworks for data analysis (Python, Perl, Matlab, ecc...). The problem of this approach is the low scalability and the implicit difficulty in the writing of a good, reliable and repeatable algorithm.

Therefore in this work we focused on the development of an integrated python environment having the following main characteristics

- *Ease of use*: biologists and researchers must be able to use this software even without solid computer science background. Therefore the complexity of the algorithms must be encapsulated into atomic and semantically rich operations.

- *Efficient processing*: The package must be able to interface to *big data* processing engines. This is a very crucial point as data used in tertiary analysis are often too big to be handled locally by the user machine. In particular, as we have said, we will interface with the GMQL system which will provide the computational power for doing genomic operations on big data.

- *Data Browsing and exploration*: this requirements comes directly from the previous one. Since we are dealing with high volumes of data, we must provide a way to explore them easily in order to understand the best pipeline.

- *Personalization*: the user must be able to define arbitrary complex pipelines using the atomic operations provided by the package.

We will apply this package to an open biological problem, which is the study of *Topologically Associated Domains* (TADs). TADs are large locations of the genomes that are thought to carry functional information. We will try to extract information about TADs by clustering them based on the expression profiles of the genes inside them.

1.4 The Python library in a nutshell

From the above requirements and through a design process that evaluated different solutions (which can be found in section 3.5) we developed a python package called PyGMQL.

The package place itself above the GMQL engine and exploits it using its Scala API or through a remote REST interface. The library therefore enables the user to operate in two different modes: in *local* mode the execution is performed in the user machine like any other python package; in *remote* mode the query is sent to the remote server and the results are downloaded, when they are ready, to the local machine at the end of execution. This creates some coherency problems that we will describe later with their solution.

Each GMQL operator has been mapped to an equivalent python function to be applied to a GMQLDataset, which represents the python abstraction of a GMQL variable.

An appropriate data structure for holding the query results has been designed. In particular this structure, the GDataframe, is very useful for performing python specific tasks or applying machine learning algorithms. A Machine Learning module is being currently developed by Anil Tuncel for his master project.

In chapter 4 we will deepen into the implementation and specifications of the python library.

1.5 Biological application

The effectiveness of the library was evaluated through its direct application to some biological problems. As first thing we applied it to some known biological problems that were already addressed by the classical GMQL system.

After this, a very complex and open biological problem was addressed: the understanding of the functional role of Topologically associating domains, which are genomic regions within which the physical interactions occur much more frequently than out of them.

We approached this problem by studying the relationship between these regions and the gene inside of them. We compared the correlation of genes in the *same* TAD with respect to genes *across* different TADs. We did this operation also considering tumor and normal expressnion profiles for the same gene.

We discovered a strong relationship between gene expression and TADs: genes on the same TAD show an higher correlation than across them. Additionally, the correlation increases when we substitute the normal expression profile with the tumor one.

We also analyzed the intersection between human and mouse TADs in order to prove their conservation across different species reaching a coverage of 60%.

Finally we performed cluster analysis of TADs by using expression profiles and ChIA-PET links (which are basically protein binding sites connections across the genome). We show how TADs can be clustered together and the characteristics of the clusters.

In chapter 6 can be found a deep explanation of these problems and the biological pipelines that were setup for solving them.

1.6 Outline of the work

This document is structured as follows:

- Chapter 2: we will give a deep background about the GMQL system. This part is fundamental in order to understand the design choices and the different architectures that were proposed and evaluated during the process.
- Chapter 3: we will deepen into the problems that making interact a python package with a Scala one. The proposed solution for exchanging data between the two languages is presented. Since GMQL is a declarative language that works in a batch execution, we will present also the problems in shifting this paradigm to an interactive and programmatic one.
- Chapter 4: here we will present the full architecture of the python module and the Scala back-end
- Chapter 5: we will deepen into the mapping between GMQL operators and python functions, also describing the defined data structures holding the results of a query
- Chapter 6: the final part of the work will explain the TADs problem and how the developed library was used to analyze it and trying to address important biological questions.
- We will conclude with an overview of the possible future enhancements and new research lines that can be opened.

Chapter 2

Background

"I'm gonna make him an offer he can't refuse."

The Godfather

In this chapter we will deepen into the features, the architecture and the design of the Genometric Query language (GMQL). This will serve as a starting point for the next sections in which we will explain the process and the solution that has been designed on top of this technology.

In particular we will explain the following concepts:

1. The Genomic Data Model: a formal framework for representing genomic data
2. The Genometric Query Language: a declarative language that gives the user the ability to perform queries on the data defined in the Genomic Data Model
3. The system architecture: the kind of services that are offered and their limitations
4. The query execution strategy: how a submitted query is executed

2.1 Genomic Data Model

We have seen that in biology and genomics there is a huge variety of data formats. The *Genomic Data Model* (GDM) tries to abstract the file format and represents genomic data as the combination of two kinds of information:

- *Region data*: the physical coordinates of the considered genome zone with the addition of specific fields having different value for each region. A set of regions is called a *sample*.

- *Metadata*: descriptive attributes of a sample (set of regions). They describe the biological, clinical and experimental properties of the sample.

It must be noticed that while for the region representation there exist several data formats, there is no agreed standard for modeling metadata. In GDM metadata are modeled as attribute-value pairs. There is total freedom in the definition of both the attributes and the values[1].

Region data, in order to be comparable between different datasets, should follow some rules defining their structure: these rules are encoded in a *schema*. Each GDM dataset has a schema and it must include at least the following information: the chromosome, the start and end positions of the region (encoded as an integer number). Optional parameter is the strand, which represents the direction of DNA reading.

2.1.1 Formal model

In [20] a formal definition of the GDM is provided. Here we will report it and deepen into the various design choices.

The atomic unity of a GDM dataset is the **genomic region**. A region is defined by a set of *coordinates*

$$c =< chr, left, right, strand >$$

and a set of *features*

$$f =< feature_1, feature_2, ..., feature_N >$$

The concatenation of these two arrays of values creates the region

$$r =< c, f >$$

The order of the coordinates and the features is fixed in all the regions of a dataset and it is dictated by the schema. In addition, these fields are all typed (for the coordinates we have respectively `string`, `integer`,`integer`,`string`).

On the other side, **metadata** are arbitrary attribute-value pairs $< a, v >$. It must be noticed that the same attribute can appear multiple times in the same sample. This is allowed due to the need of specify, in some dataset, multiple conditions for the same metadata (for example a `disease` metadata could have multiple values for the same patient).

A set of regions is called **sample** and it is structured like follows:

$$s =< id, \{r_1, r_2, ...\}, \{m_1, m_2, ...\} >$$

[1] Of course, the value must be representable in a textual form

id	chr	left	right	strand	p-value
1	2	2476	3178	*	0.00000000200
1	2	15 235	15 564	*	0.00000000052
1	5	8790	11 965	*	0.00000000009
1	5	75 980	76 342	*	0.00000000037
2	16	862	923	*	0.00000000018
2	16	1276	1409	*	0.00000000006
2	20	3852	4164	*	0.00000000031

Figure 2.1: A part of region data from a dataset having two ChIP-Seq samples

Each sample is therefore identified by an *id*, which is unique in all the dataset. A sample has multiple regions r_i and multiple attribute-value pairs of metadata m_i.

A GDM **dataset** is simply a set of samples having the same region schema and a sample *id* which is unique. This identifier provides a many-to-many connection between regions and metadata of a sample. In figure 2.1 and 2.2 we can see the relations between the metadata and the region data in the GDM.

2.2 Genometric query language

Having defined the data model we now have to explain how to query this kind of data. The language that enables us to do so is called GMQL (*GenoMetric Query Language*). It is a declarative language inspired by the classical languages for database management [20].

GMQL extends the conventional algebraic operations (selection, projection, join, etc...) with domain specific operations targeting bioinformatics applications. The main objectives of the language are:

- Provide a *simple* and *powerful* interface for biologists and bioinformaticians to huge dataset enabling them to explore and combine heterogeneous sources of informations.
- Being highly *efficient* and *scalable*
- Being as *general* as possible in order to tackle a great variety of problems and biological domains

id	attribute	value
1	antibody_target	CTCF
1	cell	HeLa-S3
1	cell_karyotype	cancer
1	cell_organism	human
1	dataType	ChipSeq
1	view	Peaks
2	antibody_target	JUN
2	cell	H1-hESC
2	cell_organism	human
2	dataType	ChipSeq
2	view	Peaks

Figure 2.2: A part of metadata from a dataset having two ChIP-Seq samples. These metadata correspond to the samples shown in figure 2.1

We can express a GMQL query as a sequence of statements crafted as follows [16]:

```
<var> = operation(<parameters>) <vars>
```

Each `<var>` is a GDM dataset and operations can be unary or binary and returns one result variable.

The resulting dataset will have inherited or new generated *id*s depending on the operation.

The declarative nature of the language implies that the user needs to specify the structure of the results and he does not care about the implementation of the operations.

2.3 GMQL operations

We can divide the set of operations in GMQL in two sets:

- *Relational* operations: these are classical operators that can be found in all the classical data management systems (`SELECT`, `PROJECT`, EXTEND, `MERGE`, `GROUP`, `SORT`, `UNION` and `DIFFERENCE`)
- *Genomic-specific* operations: designed to address the specific needs of biological applications (`COVER`, `MAP` and `JOIN`).

2.3.1 Relational operators

Select

```
<S2> = SELECT([SJ_clause ;][<pm>][; <pr>]) <S1>
```

It keeps in the result all the samples which existentially satisfy the metadata predicate `<pm>` and then selects those regions of selected samples which satisfy the region predicate `<pr>`. The result will have *id*s belonging to $S1$.

We can use *semi-join* clauses to select the samples using informations from an other dataset. They have the syntax: `<A> IN <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:

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

Semi-joins are used to connect variables, e.g. in the example below:

```
OUT = SELECT(Antibody IN EXP2) EXP1
```

which requests that the samples of `EXP1` are selected only if they have the same `Antibody` value as at least one sample of `EXP2`.

Project

```
<S2> = PROJECT([ <Am1> [AS <f1>], .., <Amn> [AS <fn>]]
               [; <Ar1> [AS <f1>], .., <Arn> [AS <fn>]]) <S1>
```

It keeps in the result only the metadata (`Am`) and region (`Ar`) attributes expressed as parameters. New attributes can be constructed as scalar expressions `fi`. 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`.

Extend

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

It creates new metadata attributes `Am` as result of aggregate functions `g`, which is applied to region attributes; aggregate functions are applied sample by sample. The supported aggregate functions include `COUNT` (with no argument), `BAG` (applicable to attributes of any type) and `SUM, MIN, MAX, AVG, MEDIAN, STD`. 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 ofsample regions, and `MinP` as the minimum `Pvalue` of the sample regions.

Group

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

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 identifer 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, iby 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
```

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 `GROUP_BY` 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 2.3.1.

Order

```
<S2> = ORDER([[DESC]<Am1>,.., [DESC]<Amn> [; TOP <k> | TOPG <k>]]
  [; [DESC]<Ar1>,.., [DESC]<Arn> [; TOP <k> | 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 $S1$ are assigned to the result $S2$. 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, TOP 5; MutationCount, 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.

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 identifers 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 concatenating the schema of the first operand with the attributes of the second operand which are not identical to any attribute of the first one; values of attributes of either operand which do not correspond to a merged attribute are set to `NULL`. For what concerns metadata, homonym attributes are prefixed with the strings `LEFT` or `RIGHT` so as to trace the dataset to which they refer.

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) EXP1 EXP2
```

extracts for every pair of samples s_1, s_2 of EXP1 and EXP2 having the same value of `Antibody` 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.

2.3.2 Domain-Specific Operations

Cover

```
<S2> = COVER[_FLAT|_SUMMIT|_HISTOGRAM]
              [( GROUPBY <Am1>, .., <Amn>)] <minAcc>, <maxAcc> ; ]
              [<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.)

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 $S1$ are assigned to the resulting single sample s in $S2$. Regions of the result sample are built from the regions of samples in $S1$ according to the following condition:

- Each resulting region r in $S2$ is the contiguous intersection of at least `minAcc` and at most `maxAcc` contributing regions r_i in the samples of $S1$ [2]; `minAcc` and `maxAcc` are called **accumulation indexes**[3].

Resulting regions may have new attributes Ar, calculated by means of aggregate expressions over the attributes of the contributing regions. `Jaccard Indexes`[4] are standard measures of similarity of the contributing regions r_i, added as default attributes. When a `GROUP_BY` 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

[2] 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.

[3] 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`. 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.

[4] 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.

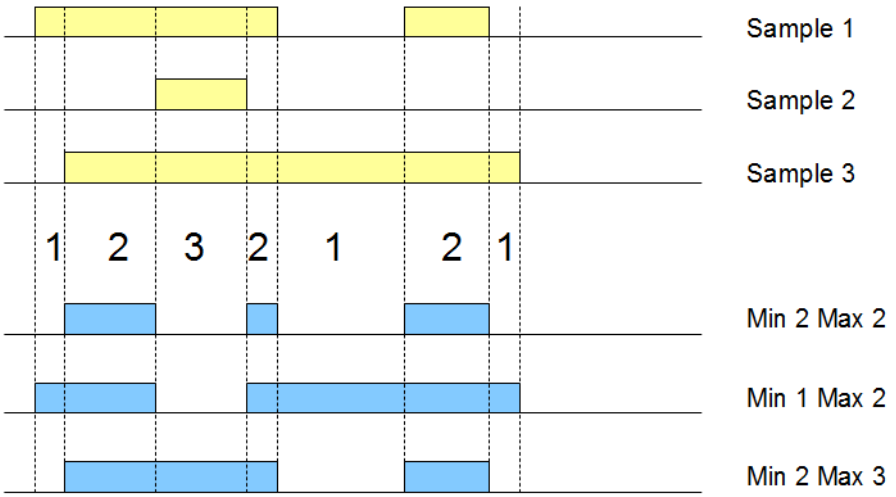


Figure 2.3: Accumulation index and `COVER` *results with three different* `minAcc` *and* `maxAcc` *values.*

operation is separately applied to each group, yielding to one sample in the result for each group, as discussed in Section 2.3.1.
For what concerns variants:

- 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).

Example. Fig. **??** shows three applications of the `COVER` operation on three samples, represented on a small portion of the genome; the figure shows the values of 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 p-Value of the overlapping regions and their Jaccard indexes; the result has one sample for each input CellLine.

```
RES = COVER(2, 3; p-Value AS MIN(p-Value)
            GROUP_BY CellLine) EXP
```

Map

```
<S3> = MAP [(JOINBY <Am1>, .., <Amn>)]
           (<Ar1> AS <g1>, .., <Arn> AS <gn>] <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 the matrix entries is a vector of numbers [5]. 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[6], 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

[5] 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.

[6] 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.

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. 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.

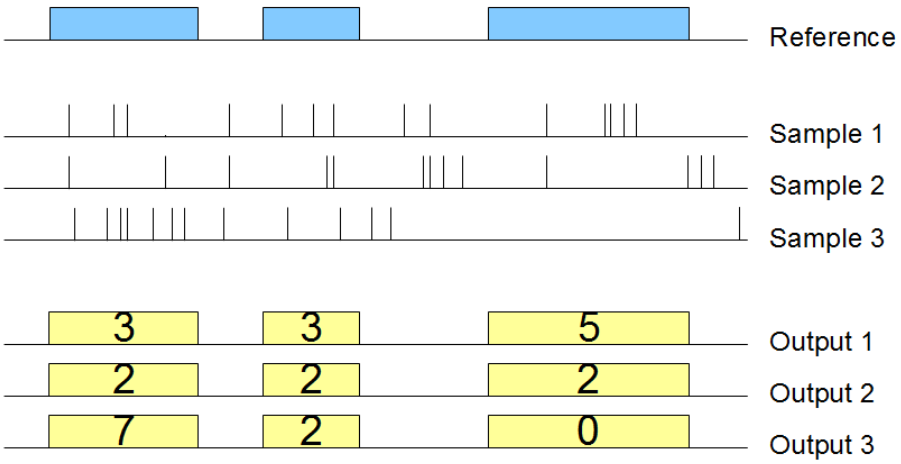


Figure 2.4: Example of map using one sample as reference and three samples as experiment, using the Count aggregate function.

Example. Fig. **??** shows the effect of this `MAP` operation on a small portion of the genome; the input consists of one reference sample and three mutation experiment samples, the output consists of three samples with the same regions as the reference sample, whose features corresponds to the mumber 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.

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

Join

```
<S3> = JOIN ([JOINBY <Am1>, .., <Amn>)] [<genometric-pred>]
```

```
<S1> <coord-gen> <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 $S1$ and s_2 of $S2$ that satisfy the join condition. If the clause is omitted, then the Cartesian product of all pairs s_1 of $S1$ and s_2 of $S2$ 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 $s1$, and then evaluating the join condition with all the regions of $s2$. 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 to 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.[7] A genometric predicate is a sequence of distal conditions, defined as follows:

- `UP/DOWN`[8] denotes the *upstream* and *downstream* directions of the genome. They are interpreted as predicates that must hold on the region s_2 of

[7] 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).

[8] Also: UPSTREAM, DOWNSTREAM.

the experiment; `UP` is true when s_2 is in the *upstream genome* of the anchor region[9]. When this clause is not present, distal conditions apply to both the directions of the genome.

- `MD(K)`[10] 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)`[11] 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[12].

- `DGE(N)`[13] 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:

```
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.

[9] *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. For the *negative strand*, ends and disequations are exchanged.

[10] Also: MINDIST, MINDISTANCE.

[11] Also: DIST <= N, DISTANCE <= N.

[12] 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.

[13] Also: DIST >= N, DISTANCE >= N.

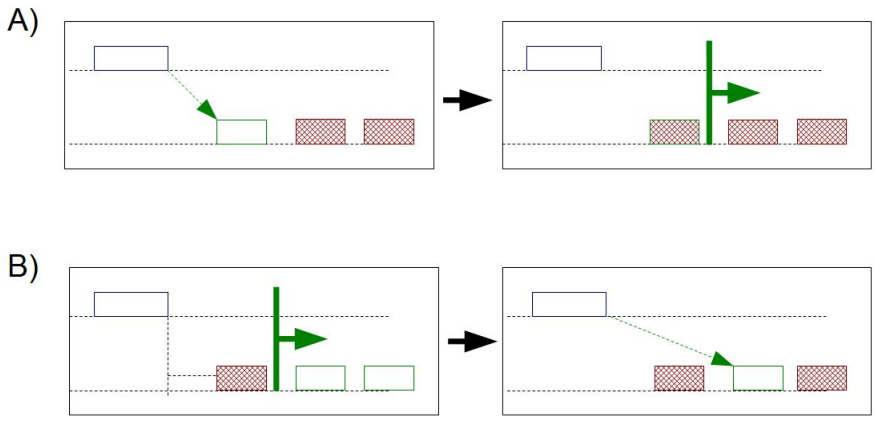


Figure 2.5: Different semantics of genometric clauses due to the ordering of distal conditions; excluded regions are gray.

Examples. In Fig. **??** 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 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[14].

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 2.3.1. The coordinates c_{ij} are generated according to the `coord-gen` clause, which has four options [15]:

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

[14]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.*

[15]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.

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[16]. Note that the result uses the coordinates of the experiment.

```
RES = JOIN(MD(1), DLE(12000); RIGHT) TSS HM;
```

2.3.3 Materialization of the results

Having performed a series of operations on a set of variables, at a certain point the user will have to declare that he wants to compute the resulting dataset w.r.t. a particular variable. This can be done using the `MATERIALIZE` operator.

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

The result is saved in a file.

It must be noticed that all the dataset defined in GMQL are *temporary* in the sense that their content is not stored in memory if the corresponding variable is not materialized.

All the declarations before a `MATERIALIZE` operation are merely a *declaration of intent*: no operation is performed until a materialization. This is a very important point and it will be crucial in the definition of our solution.

2.3.4 Some examples

In the following we show a set of examples demonstrating the power of the language:

Find somatic mutations in exons

We have a set of samples representing mutations in human breast cancer cases. We want to quantify how many mutation there are for each exon[17]

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

[17] An *exon* is a portion of a gene which will encode part of the RNA codified by that gene

and select the ones that have at least one mutation. As a final step we also want to order the result, given as a set of samples, by the number of exons present in each sample.

```
Mut = SELECT(manually_curated|dataType == 'dnaseq' AND
 clinical_patient|tumor_tissue_site == 'breast') HG19_TCGA_dnaseq;
Exon = SELECT(annotation_type == 'exons' AND
      original_provider == 'RefSeq') HG19_BED_ANNOTATION;
Exon1 = MAP() Exon Mut;
Exon2 = SELECT(count_Exon_Mut >= 1) Exon1;
Exon3 = EXTEND(exon_count AS COUNT()) Exon2;
Exon_res = ORDER(exon_count DESC) Exon3;
MATERIALIZE Exon_res INTO Exon_res
```

The first operation retrieves the mutations in which we are interested: it uses the dataset `HG19_TCGA_dnaseq` that provides DNA sequencing data in various cancer types and selects the ones about breast cancer. The second operation works on `HG19_BED_ANNOTATION` which represents the full genome annotated in significant parts and gets all the coordinates of the exons. Both these dataset are aligned to the hg19 reference genome[18].

A `MAP` operation follows which counts, for every exon, how many mutations happen to be on it. This is the most important operation in the query as it contains basically all the main logic of the program. Follows a simple selection to filter out exons having no mutations.

The `EXTEND` operation is used to put in the metadata of each sample how many exons are present in it. Follows the ordering which uses the metadata `exon_count` just created and puts the samples in descending order.

The query ends with a typical materialization.

Find distal bindings in transcription regulatory regions

We want to Find all enriched regions (peaks) in CTCF transcription factor (TF) ChIP-seq samples from different human cell lines which are the nearest regions farther than 100 kb from a transcription start site (TSS). For the same cell lines, find also all peaks for the H3K4me1 histone modifications (HM) which are also the nearest regions farther than 100 kb from a TSS. Then, out of the TF and HM peaks found in the same cell line, return all TF peaks that overlap with both HM peaks and known enhancer (EN) regions.

[18] A *reference genome* is a DNA sequence database that is used by scientists to align their experiments to a common point of reference. There are several versions for different species, based on the recency and accuracy

```
TF = SELECT(dataType == 'ChipSeq' AND view == 'Peaks'
    AND antibody_target == 'CTCF') HG19_ENCODE_NARROW;
HM = SELECT(dataType == 'ChipSeq' AND view == 'Peaks'
    AND antibody_target == 'H3K4me1') HG19_ENCODE_BROAD;
TSS = SELECT(annotation_type == 'TSS'
             AND provider == 'UCSC') HG19_BED_ANNOTATION;
EN = SELECT(annotation_type == 'enhancer'
AND provider == 'UCSC') HG19_BED_ANNOTATION;
TF1 = JOIN(distance > 100000, mindistance(1); output: right) TSS TF;
HM1 = JOIN(distance > 100000, mindistance(1); output: right) TSS HM;
HM2 = JOIN(distance < 0; output: int) EN HM1;
TF_res_0 = MAP(joinby: cell) TF1 HM2;
TF_res = SELECT(count_HM2_TF1 > 0) TF_res_0;
MATERIALIZE TF_res INTO TF_res;
```

The first four operations prepare the variables of the relative dataset for the later computation; in particular:

- We load from `HG19_ENCODE_NARROW` into `TF` all the samples representing *peaks* relative to the CTCF transcription factor.
- We load from `HG19_ENCODE_BROAD` into `HM` all the samples representing *peaks* relative to the H3K4me1 histone modification
- Using `HG19_BED_ANNOTATION` find all the coordinates relative to enhancers (`EN`) and transcription starting site (`TSS`)

The first thing we do is to join the transcription starting sites to the CTCF transcription factor. We select only the CTCF which are farther than 100 kb from a TSS. Same process is done for the histone modification.

Now we take only the `HM1` regions that intersect with an enhancer `EN` and we take the intersection of the resulting regions into `HM2`.

Finally, for each cell line, we count how many regions in `HM2` intersect with the transcription factor regions in `TF1` and take only the transcription factors having at least one `HM2`.

2.4 Architecture of the system

The architecture of the system underwent a series of revisions during time.

In the initial release (see figure 2.6) [16] the GMQL queries were translated to PIG [1]. Basically a *translator* between GMQL and *Pig Latin* [25]

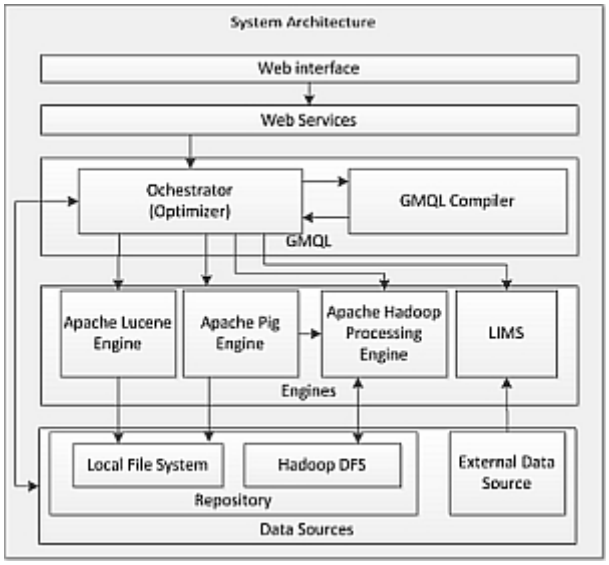


Figure 2.6: First version of GMQL. The architecture.

was developed. Pig Latin is a language designed to act as a bridge between the declarative world of SQL and the low-level performance-driven world of Map-Reduce [10].

The second version of the system completely revised the architecture and was designed to have:

- an execution layer independent from the storage one
- the possibility to have *multiple* engine implementations and to choose the desired one
- the possibility to have *multiple* storage systems
- a set of API enabling external applications or languages to interface the engines

In figure 2.7 we can see a graphical representation of an high level architecture of GMQL V2 [19].

We can therefore divide the implementation of the system in the following layers:

- *Access layer*: the highest level of the architecture. Enables external users to interact with the system. This can happen through a web interface, a command line interface and a lower level API in Scala.
- *Engine abstraction*: at this level the system present a series of modules that manage the engines from an high level point of view. It contains

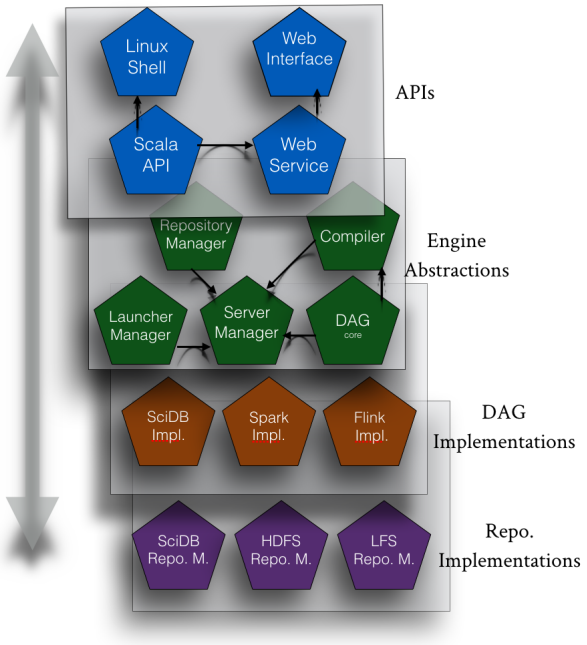


Figure 2.7: Current architecture of GMQL. Taken from [19]

the language compiler and the repository manager, which interact with a DAG scheduler, whose objective is to represent the tree structure of a query. The engines are managed by a server manager and the submission of a query is managed through a launcher manager.

- *Implementations*: at this level there are the operator implementations in the various engines. Currently GMQL offers an implementation in SciDB, in Spark and Flink. The Spark implementation is the most updated and stable one.

- *Repository implementations*: the last level manages the datasets and currently GMQL enables the user to choose between a local file system, the Hadoop HDF and the SciDB repository manager.

2.4.1 User interfaces

As we have said, GMQL currently provides the following interfaces for the final user:

- A *web interface* (see figure 2.9): the user can specify a textual query using both its private dataset or public ones. These dataset are stored

```
GMQL-Submit  [-username USER] [-exec FLINK|SPARK] [-binsize BIN_SIZE]
 [-jobid JOB_ID] [-verbose true|false] [-outputFormat GTF|TAB]
 -scriptpath /where/gmql/script/is

Description:
[-username USER]
    The default user is the user the application is running on $USER.
[-exec FLINK|SPARK]
    The execution type, Currently Spark and Flink engines are supported
    as platforms for executing GMQL Script.
[-binsize BIN_SIZE]
    BIN_SIZE is a Long value set for Genometric Map and Genometric
    Join operations. Dense data needs smaller bin size. Default is 5000.
[-jobid JOBID]
    The default JobID is the username concatenated with a time stamp
    and the script file name.
[-verbose true|false]
    The default will print only the INFO tags. -verbose is used
    to show Debug mode.
[-outputFormat GTF|TAB]
    The default output format is TAB: tab delimited files in the
    format of BED files.
-scriptpath /where/gmql/script/is/located
    Manditory parameter, select the GMQL script to execute
```

Figure 2.8: Command line interface for sending GMQL queries.

using the repository manager. There are also utilities for building interactively the queries and for exploring the metadata of the datasets.

- A *command line* (see figure 2.8): a very simple interface for sending directly the GMQL scripts to execution.

2.4.2 Scripting interfaces

At a lower level of interaction we find a Scala API that the user can use to build query *programmatically*. This routines enable the creation of the *Directed Acyclic Graph* (DAG), which is an abstract representation of a GMQL query.

Figure 2.9: Example of web interface usage for GMQL

The execution of the query is naturally performed locally in this case and requires the user to provide all the computational environment to execute it. For example in the following example the user is required to instantiate a Spark context to support the execution of the GMQL engine.

```
import it.polimi.genomics.GMQLServer.GmqlServer
import it.polimi.genomics.core.DataStructures
import it.polimi.genomics.spark.implementation._
import it.polimi.genomics.spark.implementation.loaders.BedParser._
import org.apache.spark.{SparkConf, SparkContext}

val conf = new SparkConf()
     .setAppName("GMQL V2.1 Spark")
     .setMaster("local[*]")
     .set("spark.serializer",
           "org.apache.spark.serializer.KryoSerializer")
val sc:SparkContext =new SparkContext(conf)

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

val expInput = Array("/home/V2Spark_TestFiles/Samples/exp/")
val refInput = Array("/home/V2Spark_TestFiles/Samples/ref/")
val output = "/home/V2Spark_TestFiles/Samples/out/"

//Set data parsers
val DS1: IRVariable = server READ ex_data_path USING BedParser
val DS2 = server READ REF_DS_Path USING BedScoreParser2

//Selection
```

```
val ds1S: IRVariable = DS1.SELECT(
    Predicate("antibody", META_OP.EQ, "CTCF"))
val DSREF = DS2.SELECT(Predicate("cell", META_OP.EQ, "genes"))

//Cover Operation
val cover: IRVariable = ds1S.COVER(CoverFlag.COVER,
               N(1), N(2), List(), None)

//Store Data
server setOutputPath output_path MATERIALIZE cover

server.run()
```

We can easily see from this example that this mode of usage of the system is not as easy and user-friendly as the others.

2.4.3 Engine abstractions

A GMQL query can be represented as a graph. If we simply look at the single GMQL statements in the language we could say that a node of this graph represent one statement.

Obviously, each GMQL operation can be split in different subsections (sub-nodes) and this increase the granularity of our graph representation. The first division is surely between *region nodes* and *metadata nodes*. An additional category of nodes regards *meta-grouping* and *meta-joining*: these operations are needed to group or join samples from one or more datasets.

Metadata nodes

- `ReadMD`: reader of metadata files
- `ReadMEMRD`: reader of metadata from memory
- `StoreMD`: materialization of metadata
- `SelectMD`: filtering of metadata
- `PurgeMD`: deletion of an empty dataset
- `SemiJoinMD`: the semi-join condition in `SELECT`
- `ProjectMD`: projection of metadata
- `AggregateRD`: produce a new metadata based on the aggregation of region data

- `GroupMD`: partitions the input data into groups and creates a new metadata for each sample indicating the belonging group
- `OrderMD`: orders the samples according to a metadata value
- `UnionMD`: union of samples of two input metadata nodes
- `CombineMD`: combines two metadata nodes and produces as output the union of each pair of metadata samples
- `MergeMD`: union of the samples of the input node

Region nodes

- `ReadRD`: reader of region files
- `StoreRD`: materialization of region data
- `PurgeRD`: filters out the regions from the input that have an *id* that is not in a metadata node
- `SelectRD`: filtering of region data
- `ProjectRD`: projection on region data
- `GroupRD`: partitions the regions in disjunctive sets and only one region for each partition is returned
- `OrderRD`: orders the regions based on one of their attributes
- `RegionCover`: applies the Genometric Cover to a node
- `UnionRD`: puts together all the regions from the input node
- `DifferenceRD`: difference of two datasets. Returns only the regions of the left dataset that do not intersect with the ones of the right
- `GenometricJoin`: applies the Genometric Join to two nodes
- `GenometricMap`: applies the Genometric Map to two nodes
- `MergeRD`: computes the union of the region nodes and assigns new *id*s for avoiding collisions

Meta-grouping and Meta-joining

- `GroupBy`: partitions the dataset based on metadata
- `JoinBy`: applies the join condition to the cross product of the input metadata nodes

Example query

For a better understanding of the DAG structure let's see how the following query would be implemented as a tree structure:

```
GENES = SELECT(<pm1>) ANNOTATION;
PEAKS1 = SELECT(<pm2>) BED_DATA;
PEAKS2 = SELECT(<pm3>) BED_DATA;
FILTERED = JOIN( DLE(0) ;
                antibody == antibody,
                cell == cell) PEAKS1 LEFT PEAKS2;
FILTERED2 = GROUP(chr,start,stop,strand) FILTERED;
MAPPED = MAP(COUNT) GENES FILTERED2;
MATERIALIZE MAPPED;
```

In figure 2.10 we can see the corresponding DAG implementation.

2.4.4 Implementations

Currently the most updated and stable implementation of the DAG operators uses Spark [2]. Each DAG node has a corresponding implementation.

Since Spark itself builds a directed acyclic graph for optimize the computation we result is a second (larger) DAG after the translation of the DAG operations into Spark ones.

In this work we will only focus on the Spark implementation of GMQL and our Python API will exploit only this computational engine.

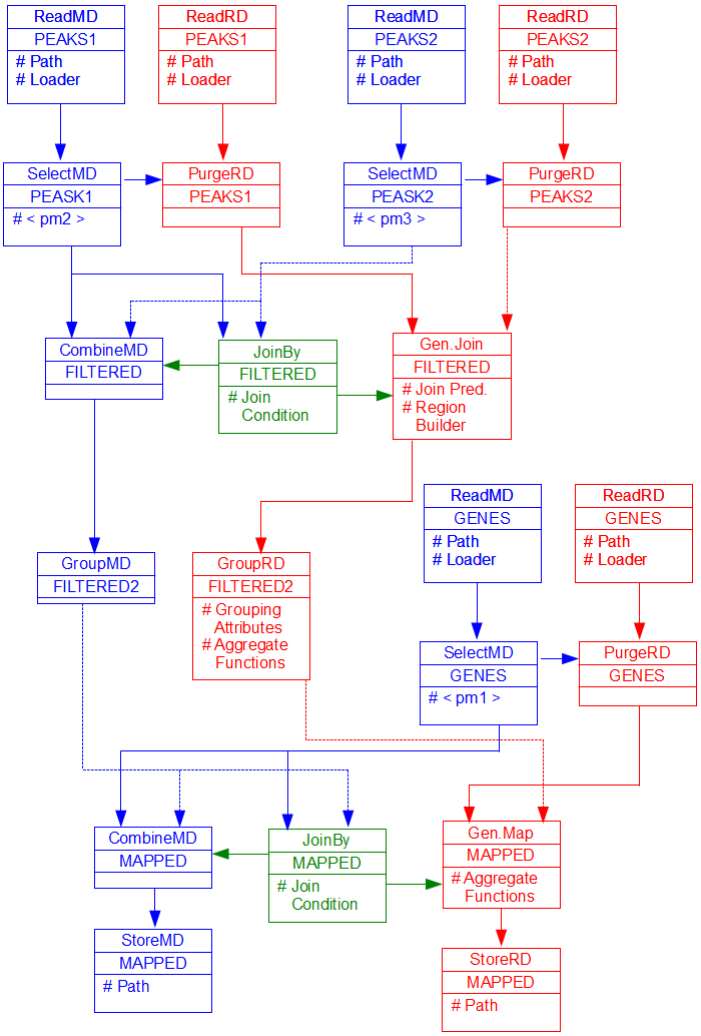


Figure 2.10: Complete abstract DAG for the query.

Chapter 3

Interoperability issues and design of the library

"I love the smell of napalm in the morning."

Apocalypse now

Before getting into the description and design of the Python library that should integrate the GMQL environment with the Python programming model we need to explore and compare these two worlds.

GMQL is written in the *Scala* language and its main implementation exploit the Spark big data engine for doing computations. We need therefore to understand how to integrate this language and this computing engine with Python.

In this chapter we will cover the following aspects:

1. *Scala*: we will describe the Scala language with a particular attention on its differences with respect to Java.

2. *Python*: a description of the Python programming language will follow. Also the main users and usages of this language will be explained.

3. *Interoperability*: we will show how make a Python and a Scala program interact. This step is fundamental for the design of the PyGMQL library.

4. *Interactive computation*: we will show the main issues that arise when we try to make a big data computation system *interactive* and how this requirement was satisfied in the design of PyGMQL

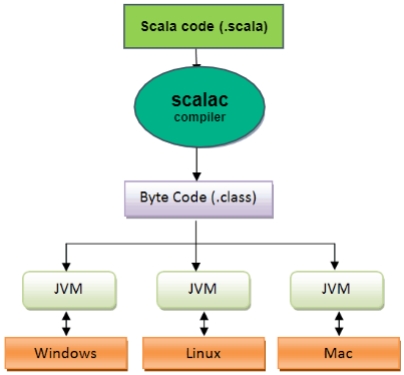


Figure 3.1: Process of compilation of a Scala program

5. *Alternative architectures*: the last part of the chapter is dedicated to the description of the various alternative architectures that were considered for PyGMQL. Advantages and disadvantages of each of them will be presented.

3.1 Scala language

Scala is a multi-paradigm programming language: it is object-oriented, imperative and *functional*. Its design started in 2001 at the École Polytechnique Fédérale de Lausanne (EPFL) by Martin Odersky and it was the continuation of the work on *Funnel*, a programming language mixing functional programming with Petri nets [22]. It was publicly released in 2004 on the Java platform.

3.1.1 Compatibility with Java

Scala is implemented over the Java virtual machine (JVM) and therefore is compatible with existing Java programs [23]. The Scala compiler (`scalac`) generates a byte code very similar to the one generated by the Java compiler. From the point of view of the JVM the Scala byte code and the Java one are indistinguishable. In figure 3.1 we can see the process of Scala compilation.

3.1.2 Extensions with respect to Java

Scala introduces a lot of syntactic and model differences with respect to Java. Follows a brief list of the main differences:

- In Scala *everything* is an object, including numbers and functions. This

is different from Java, where primitive types (boolean, integers, floats, etc...) are distinguished from reference types (the classes). This means that also numbers can have methods for example.

Functions are objects and this is what makes Scala a *functional programming* language. For example in the following code, that shows a timer function that must perform *some action* every second:

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

we can see that the function `oncePerSecond` gets a call-back function as argument. The type of the argument can be read as "all functions which take no arguments and return nothing" (`Unit` is the null return value, similar to the `void` of the C language).

- In Scala it is possible to define *anonymous functions* which are basically functions without a name. A revised version of the example above that uses an anonymous function is below:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

- All classes in Scala inherit from a super-class. When no super-class is specified it is implicit that the class inherits from `scala.AnyRef`

- Scala offers the concept of *case classes*, which are a special kind of classes that can be seen as plain immutable data-holding objects. The

state of a case class depends *exclusively on the constructor arguments*. This enables the Scala compiler to perform several optimizations and to add very useful features like *pattern matching*, *equality* and *comparison*.

In the code below we see an application of case classes in the context of the creation of an algebraic expression tree:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree

def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n) => env(n)
  case Const(v) => v
}
```

Notice how the function `eval` uses pattern matching to understand the type of the node.

3.1.3 Spark and Scala

The Scala language was selected by the Spark development team as the engine implementation language. This is due the functional programming style that facilitates the writing of Big Data applications and the conciseness of the language.

In the last years several API were offered to the Spark users for interfacing with the most common programming languages like Python, R and Java.

The GMQL project is currently implemented in Scala for a better compatibility with the Spark engine and for facilitating the writing of parallel genomic operations.

3.2 Python language

Python is an interpreted multi-paradigm programming language with dynamic semantics. It is designed for Rapid Application Development (RAD) [7] and for creating scripts.

The main focus of Python is on the easiness of the language, the plain learning curve and the readability. This creates applications that are very easy to maintain, develop and reuse.

One of the most powerful aspects of Python is its huge development community, which is one of the biggest in the world. Every python package is open source and can be downloaded very easily through package versioning programs like `pip` and `easyinstall`.

3.2.1 An interpreted language

Due to the interpreted programming model, Python can be use *interactively* in the sense that one can open a Python shell, digit commands and receive an immediate response without compiling the source code. In the following example we see the Python console with some commands and Python control structures:

```
$ python3.6
Python 3.6 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

This model suits very well the needs of data scientists and researchers, that most of the time explore, mine and analyze data without having a clear initial idea on what to do with them. For this reason Python has become a de facto standard in the data science community, that regularly publishes new data analysis packages with state of the art algorithm implementations.

3.2.2 A strongly dynamically typed language

Python is also strongly dynamically typed. This means that it combines two features:

- *Strongly typed language*: every value assigned to a variable has a type that cannot change unless an explicit conversion is ordered

```
>>> password = 1234
>>> encryption = 'encryption_text'
>>>
>>> type(password)
<type 'int'>
>>> type(encryption)
<type 'str'>
```

- *Dynamically typed language*: the runtime objects (variables) can hold values having different types. This is the opposite of statically typed languages, where a variable is associated directly to a type and the values assigned to that variable must be coherent with its type.

  For example in the following example the variable `employeeName` is assigned with an integer, than a string and then an other integer in three consecutive python statements.

```
employeeName = 9
employeeName = "Steve Ferg"
employeeName = 8*9+5
```

3.3 Connecting Scala and Python

One of the main issues that were addressed during this project was *how can we make a Python module communicate with a Scala program?* Since GMQL is programmed in Scala and all its data structure are implemented using Scala constructs, we need a way to communicate between our Python API and the Scala back-end.

The solution that we adopted was inspired by the work done by the Spark development team for integrating Python with their Big Data processing system. We use a python library called `Py4j`[8] that has enables the python interpreter to dynamically access Java objects contained in a JVM that is currently running a program. The interaction between the python and Scala programs follows the classical client-server paradigm. Py4J can be used in both directions: Scala as main program (client) and Python as a service (server); Python as main program (client) and Scala as a service (server).

In figure 3.2 we can see an example of interaction using Py4J.

Basically we need to instantiate a client python-side that it is called Java Gateway: this object will instantiate the communication with a Gateway Server created in the JVM. The communication can be carried on with several protocols like SSL. The server is waiting for commands on a specified port.

In figure 3.3 we can see what happens when PyGMQL is instantiated in a python program.

1. First of all we check how many other instances of PyGMQL are currently running in our system. In the `instances` file we can find the port numbers where the different back-ends are listening.

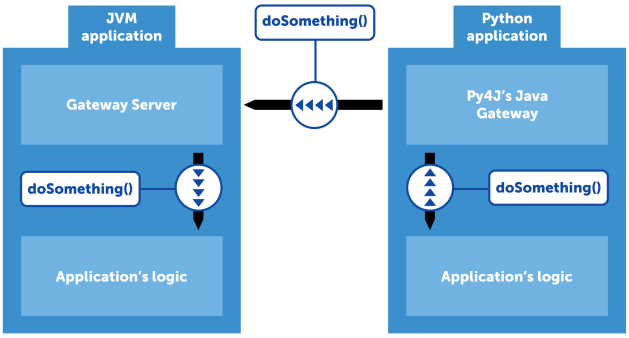


Figure 3.2: Py4J model

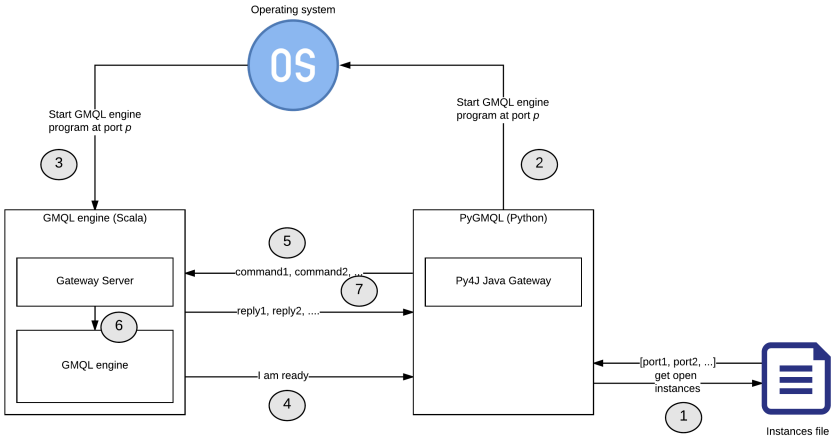


Figure 3.3: Process of creation of the Scala back-end and interaction with PyGMQL.

2. PyGMQL generates a new port number (different from the already used ones) and ask the operating system to start the Scala back-end at the specific port

3. When the back-end is ready it sends an acknowledgement signal to PyGMQL

4. Now the normal communication can begin: PyGMQL uses the Java Gateway for sending commands and the Server Gateway receives them. When the results are ready, they are sent to the caller.

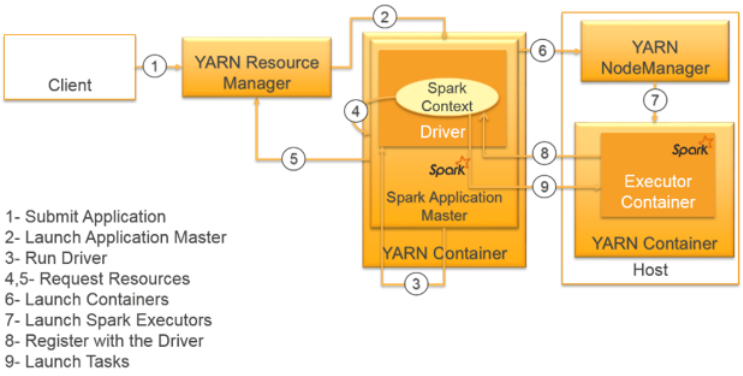


Figure 3.4: Steps in the Spark Submit mode of execution

3.4 Asynchronous big data processing and interactive computation

A great attention in the last year is given to the research of interactive ways to perform or visualize big data operations [27].

One of the main tasks of PyGMQL is to enable an interactive way of working with big genomic datasets. The problem is that GMQL is a declarative language and this paradigm is suited for batch processing: the user writes the query *entirely*, sends it to the system, waits for a response and finally downloads the result. This execution mode does not fit the requirements of the python package we have to develop.

Therefore a big effort was put into *hiding* from the user the underlying declarative implementation.

The Spark implementation of GMQL leverages on the Submit mode of execution (see figure 3.4). This means that the Scala program containing the query is packed in a JAR file and sent to the cluster at the moment of execution.

We will see that the PyGMQL library have to support both *local* and *remote* execution. This means that the computational engine must be available both on the local machine and on the remote server.

For what concerns remote execution, the only thing that we can do is to hide the submission of the query and the downloading of the results. The execution command is blocking and waits until the completion of the download of the resulting datasets.

Local execution requires the execution of Spark locally. In order to hide this from the user, the Spark engine is packaged in the Scala back-end together with the GMQL engine. The execution mode of Spark must be

changed to *local* and this means that the user machine acts both as driver and executor. The submission of the query is substituted by the direct calling of Spark actions like `collect` or `take`. The SparkContext is kept alive during the entire program session and not used only for one single query.

3.5 Analysis of Design Alternatives

Here we describe the design choices and the various alternatives that were examined during the first part of the project.

Due to the high number of stakeholders and the size of the development team a time-line was defined. The architecture proposal were evaluated by the development team of GMQL and a final structure was selected accordingly.

The design choices that led to the various architecture proposal and to the final architecture had to take in consideration both *soft* and *hard* requirements of the system. A soft requirement is related to the development team and to the project objectives as a research driven software. On the other side an hard requirement deals with the technicalities of the software, the specific technical barriers that need to be surpassed and the features that need to be added to the existing software. In table 3.1 are listed the main soft and hard requirements for this project.

Table 3.1: Table of the main soft requirements and hard requirements of the Python API for GMQL

Soft	**Hard**
The implementation of the API must *minimize the amount of modifications* to the already existent system. A conservative approach should be always considered.	The library must be *interactive* in the sense that the user can perform a series of operations, see the result and continue working with the resulting data.
The implementation must be *easy to update* due to the high rate of changes to the GMQL system, which undergoes regular redesign and extensions.	The library must change the paradigm of the GMQL language from declarative to programmatic. The underlying declarative structure of the system must be *transparent* to the user.

The function signatures must be *coherent* and similar to the ones of the GMQL operators. This is needed in order to make the learning of the software by the user simpler.	The user must be able to use all the usual python libraries for data analysis in conjunction with the GMQL API.
The library must be publicly available on the international Python repository	The installation of the library must be straightforward and similar to a typical Python library.

It must be stressed out the fact that GMQL is an open project and that it is constantly evolving. This constitutes an added challenge both from a design and an implementation point of view. It creates both external (user driven) and internal (project driven) requirements and both of them are equally important.

Other important aspect that guided the design of the library was the type of user that it addresses. We can say that the main users of it are described in table 3.2

Table 3.2: Table of the main users of the library

User	**Description**
Biologists	The principal users. Usually they are not very skilled in programming and they use languages like R having a very smooth learning curve. They mainly use computer science tools for data extraction
Bioinformaticians	Other main users. They have much more computer science experience and they use a wide variety of languages and tools.
Data scientists with focus on biological problems	As bioinformaticians they have a good knowledge of programming languages but they usually rely on more general purpose libraries
Generic researchers	They can work on a potentially infinite type of problems. We can assume that the mean expertise of this category is similar to the one of biologists

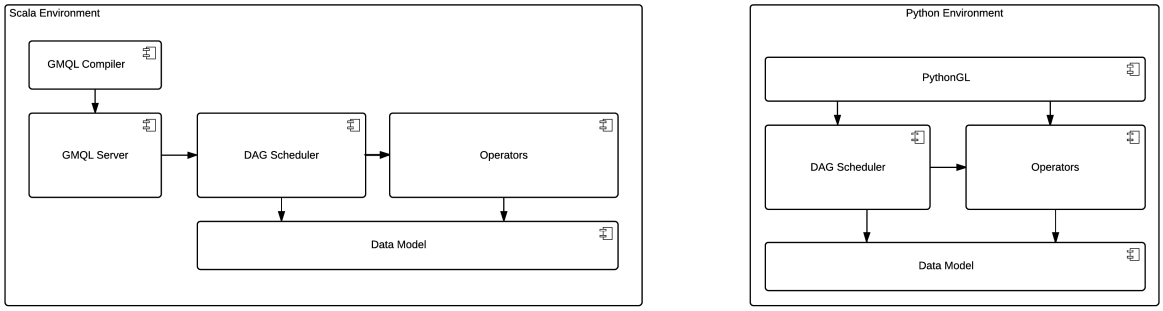


Figure 3.5: High level diagram of the first architecture proposal

3.5.1 Full-python implementation

The first architecture proposal that was considered was a complete new python implementation of the system. The focus of this brand new implementation is on a programmatic interface with the user. Therefore no compiler is needed and the user directly calls function of an API at a lower level than in the case of GMQL queries.

The data structures at the basis of the original system (basically the organization of the samples $s =< id, \{r_1, r_2, ...\}, \{m_1, m_2, ...\} >$ and the datasets), that originally are implemented as Scala objects need to be converted to Python structures (for example a dictionary of the type $\{id \rightarrow ([r_1, r_2, ...], [m_1, m_2, ...])\}$)

The computational engine also in this case is Spark and it is based on the Python wrapper called PySpark [3], which was already described in the previous chapter.

In figure 3.5 we can see a conceptual representation of this architecture. We can see that the original GMQL implementation is left untouched the new package it totally independent.

Let's see some advantages and disadvantages of this approach:

Advantage	**Disadvantage**
Full *freedom* in the implementation. We can craft the system to be very efficient for the task it is designed. The data structure can be designed to be efficient in Python computations and to be used in already existent data analysis libraries.	All the GMQL operators need to be reimplemented in Python (PySpark).

We do not rely on the original engine implementation, which is designed for a declarative system. We can exploit the *interactivity* of the Python language and use PySpark for heavy computations.	We need to implement a new DAG scheduling system that take into account the interactivity requirements.
The python data structures enable the users to use arbitrary selection/projection criteria by using *lambda functions*. In general, by relaxing the data model we can generalize some operations and enable user-defined conditions/expressions.	The autonomy from the original system makes necessary to continuously synchronize the two versions of the operators. If an operator is modified/deleted/added to the original system, the same must be done in the Python implementation.

3.5.2 Mixed approach

The second proposal tried to get the best from the Python world and the original Scala implementation. Some operators have to be re-implemented in python in order exploit the power of the language, while others can be left in their original implementation.

This approach is very complex and requires one to create a sort of transformation procedure between the data structures in Python and the original ones. It relies on Spark and PySpark and therefore a procedure for converting the Python RDDs to the Scala ones is needed.

In figure 3.6 we can see the conceptual architecture of this approach.

In the following table we summarize pros and cons of the approach.

Advantage	**Disadvantage**
We can use Python constructs for specific GMQL operations like selection and projection	We need to implement a complex data transformation module Python to/from Scala
We do not need to re-implement all the GMQL operators	Synchronization issues persist like before, because some operators are re-implemented in Python

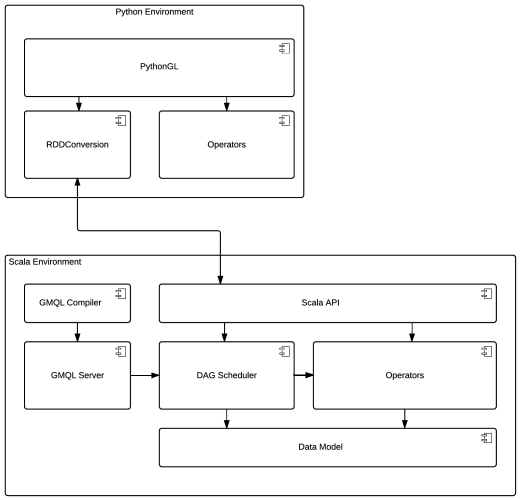


Figure 3.6: High level diagram of the second architecture proposal

3.5.3 Wrapper implementation with added functionalities

The last approach that has been considered, and that at the end was selected, completely separates the computational engine of GMQL from the Python API. The Python library uses the GMQL engine as a black box that takes a set of datasets as input and produces a set of datasets as outputs. The interaction can be both local, through a to-be-defined communication protocol, or remote, through the use of a proper REST API.

This approach clearly respects the principle of *separation of concerns* and creates a layered and modular architecture. The Python library has the role of providing efficient data structures for pre-processing and post-processing the data before and after a GMQL query, which is executed by the below engine transparently.

The main focus of this approach is on hiding the underlying declarative architecture and providing, as much as possible, an interactive interface to GMQL operations. The library, in addition, can add functionalities for genomic data processing, like clustering methods, classification and statistical analysis.

Advantages	**Disadvantages**
The architecture is fully modular. Any change at the lower level is automatically updated at the API level.	We have lost the total freedom of the first and second implementation. We have to rely on an already defined data model and computational engine.

We can use the repository service that is already provided by the original system.	Any additional feature that is added to the API cannot be propagated to the underlying system.
Remote computation is already provided by a REST API	The underlying system is designed for handling declarative programs, therefore it is not suited for interactive processing. A particular attention must be put on the design of a system for mimicking an interactive interaction.

In figure 3.7 there is the conceptual architecture.

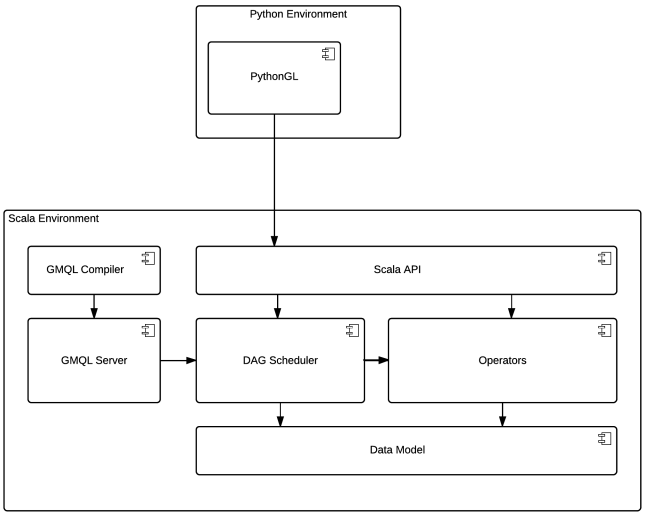


Figure 3.7: High level diagram of the third architecture proposal

Given the hard and soft requirement listed in the previous section, we selected as most promising solution this last architecture proposal.

Chapter 4

Architecture of the library

> *"I've seen things you people wouldn't believe. Attack ships on fire off the shoulder of Orion. I watched C-beams glitter in the dark near the Tannhauser Gate. All those moments will be lost in time like tears in rain. Time to die."*
>
> Blade Runner

In this section we will explain the architecture and the details of the developed python package. In particular the chapter will be organized as follows:

- We will firstly illustrate an high level view of the package, the different modules and their role.
- Then we will deepen into the design choices that were made for enabling remote and local execution and the interoperability of these two modes.
- At the conclusion of the chapter, a brief description of the packaging and publication of the library

For a deep explanation of the GMQL operators wrapped in PyGMQL, the data structure used for holding the results of a query and a complete example of usage refer to chapter 5.

4.1 General architecture

In this section we will describe the features of the PyGMQL library. We will follow a top-down approach by firstly describing the different modules as black boxes and the deepening into the details of each of them.

The python API is part of the GMQL ecosystem and offers to the user the possibility to perform GMQL computations both locally (in the user

machine) and remotely (through an cluster of server appropriately configured with the GMQL engine). In figure 4.1 we can see the relationship between PyGMQL and the GMQL ecosystem along with the use cases of both of them.

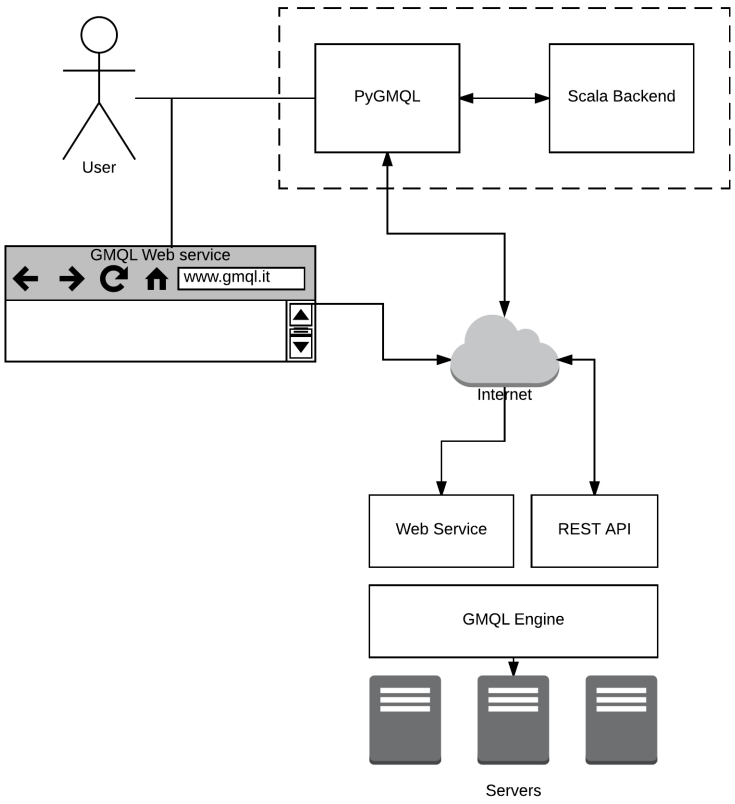


Figure 4.1: High level view of the whole GMQL system.

PyGMQL substitutes the Web interface and offers a set of functions for building GMQL queries programmatically. The python library interacts with a local Scala back-end which contains the essential computational modules of GMQL. In the Scala back-end we can find the definition of the operators, their interface and the DAG data structure for query optimization. The Python side and the Scala side dialog through a classical client-server model in which the Scala back-end acts as the server.

The remote execution is possible through a REST API offered by the GMQL server, which offers a set of functions that mirror the features of the Web interface. The user can use both local and remote datasets for building his queries and he can also decide if the computation must be performed

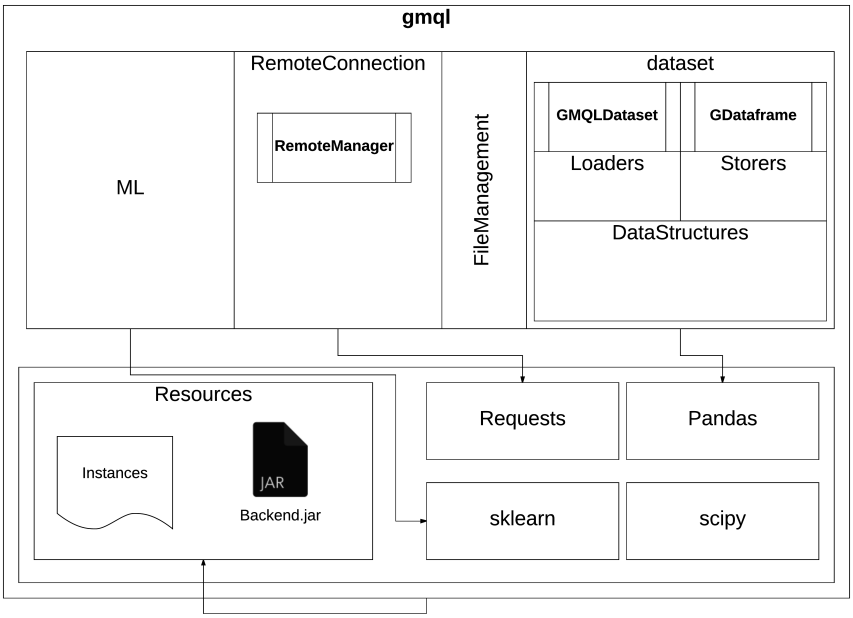


Figure 4.2: Architecture of the PyGMQL package

locally or remotely. This is a very important feature and its details will be explained later. The download or upload of datasets from/to the remote server can be done automatically or manually by the user through proper functions.

In figure 4.2 we show the main modules of PyGMQL.

We can see that the architecture can be seen as two layer of packages that interact. Starting from the bottom we have some of the dependencies of the library, that are already existent python libraries like `Requests, Pandas, sklearn, scipy`. At the same level there is a package called `Resources` that contains very important files and executables used for the management of the execution of the programs: inside this folder there is also the Scala executable running the back-end.

We can see that the library exposes the following packages to the user:

- `dataset`: it contains the main abstractions of the library. This module implements the wrapping procedures and strategies for writing GMQL queries, loading and storing of datasets, and management of the results of the queries. It is the most important package in PyGMQL

- `FileManagement`: this module manages the local and remote datasets. It can be used for creating temporary directories for storing datasets or results in general

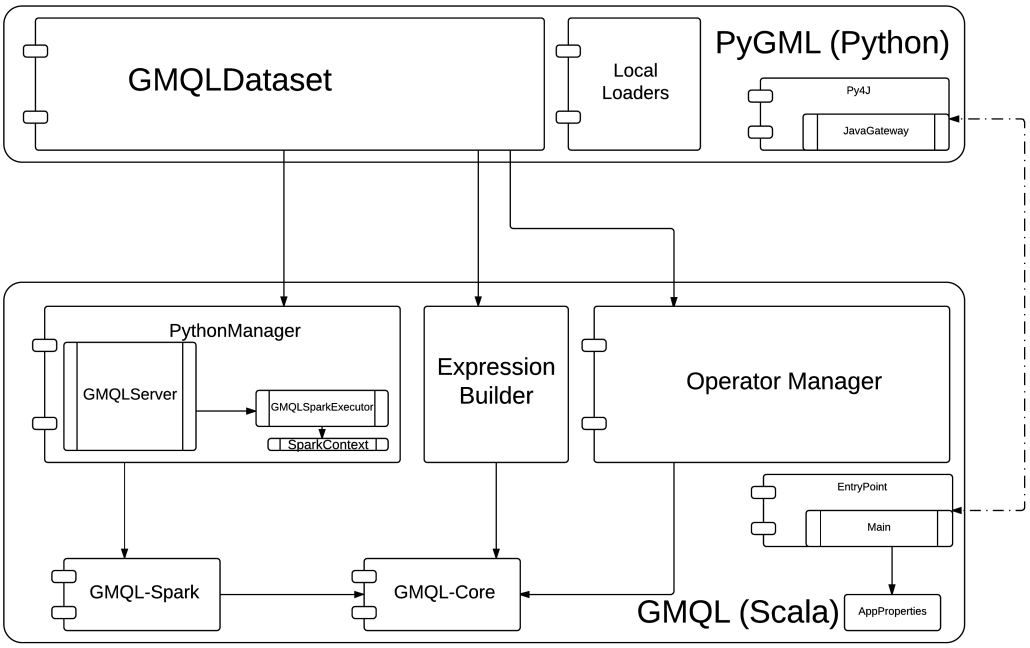


Figure 4.3: Main interactions between the Python library and its Scala back-end

- `RemoteConnection`: functions and classes for the communication with the remote server. If the user works in local mode, he does not need to use this package
- `ML`: a module implementing clustering and classification algorithms designed for genomics. It is currently developed by Anil Tuncel for his master thesis.

In particular, the `dataset` package contains the two most important data structures in the library:

- `GMQLDataset`: represents a GMQL variable. The user can apply to this variable or to a set of them all the GMQL operations. A GMQL statement is converted to a Python assignment like follows:

  ```
  new_dataset = dataset.operation(paramters, ...)
  ```

  where `parameters` can be also an other `GMQLDataset`
- `GDataframe`: The act of materializing a (set of) variable(s) triggers the execution of the query. This produces a result, which can be stored in

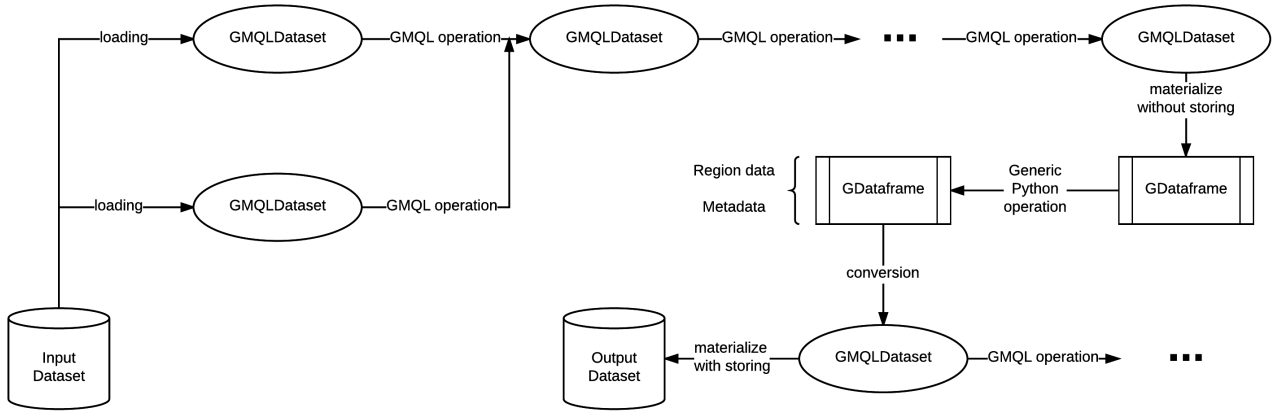


Figure 4.4: An example of programming work-flow using PyGMQL.

a `GDataframe`. This structure is fully python-based and encapsulates both region and metadata for post-query exploration or analysis.

These two data structures therefore have the role of representing genomic data in two states of their "life" inside the program. GMQLDataset can be thought as a pointer to a local or remote genomic dataset during a GMQL query. GDataframe is a python local structure holding in memory both regions and metadata and it is generated after a `materialize` operation or simply by loading directly a genomic dataset in memory. GDataframe are designed to be used for exploring, analyzing and mining the data before/after a GMQL query. In figure 4.4 we can see an example of work flow of a program that makes use of both these structures.

We can see that it is possible, once a GDataframe is materialized, to come back to a GMQLDataset and do other GMQL queries. This is an extremely powerful feature, because it enables to mix very specific operations (like, for example, a machine learning algorithm that assign to each region a class) with genomic operations in the same program.

4.2 Remote execution

In this section we will briefly explain how PyGMQL manages the remote execution of its queries. We will see that this is not a straightforward procedure due to both language and implementation requirements.

An important design choice that influenced the implementation of the remote execution feature was to allow the user to select the computation

mode (local or remote) *wherever in the program he wants*. This means that the user can change the mode from local to remote and vice versa in each moment.

We have seen that the DAG of a query is composed of several nodes representing the various operations performed on data. If a query is executed it means also that the last operation in the DAG is a materialization. Until then no operation is performed.

This bring to the conclusion that every choice related to materialization cannot be taken before the `materialize` operation. Only at that time we will be able to know what to do. At materialization time, PyGMQL has to do the following:

- *Local mode*: Every remote dataset involved in the query must be downloaded from the remote server and its name must be changed in the DAG nodes to the new local path. After that the execution can start using only local datasets.

- *Remote mode*: Every local dataset involved in the query must be uploaded to the remote server and its name must be changed in the DAG nodes to the new remote name. The DAG then must be sent to the server. After that the remote execution can start. At the end of computation, if required, PyGMQL downloads the result datasets.

In figure 4.5 we can see a graphical representation of the whole process.

4.2.1 Dag serialization

In order to send the DAG of the query to the server, this graph must be transformed into a stream of bytes. The strategy that we adopted was to *serialize* the DAG into a string encoded in Base64[1].

4.2.2 Graph renaming

As we have seen, in order to execute correctly the query we need to be able to modify some nodes in the DAG. In particular we are interested in the initial nodes, which are related to the loading of datasets from a location. In chapter 2 we have seen that these node are named `ReadMD` (for metadata) and `ReadRD` (for region data). The graph renaming process happens both at the API side (managed by PyGMQL) and at the server side (managed by

[1] *Base64* is a set of binary-to-text encoding schemes. It uses 64 different textual characters to represent bit strings

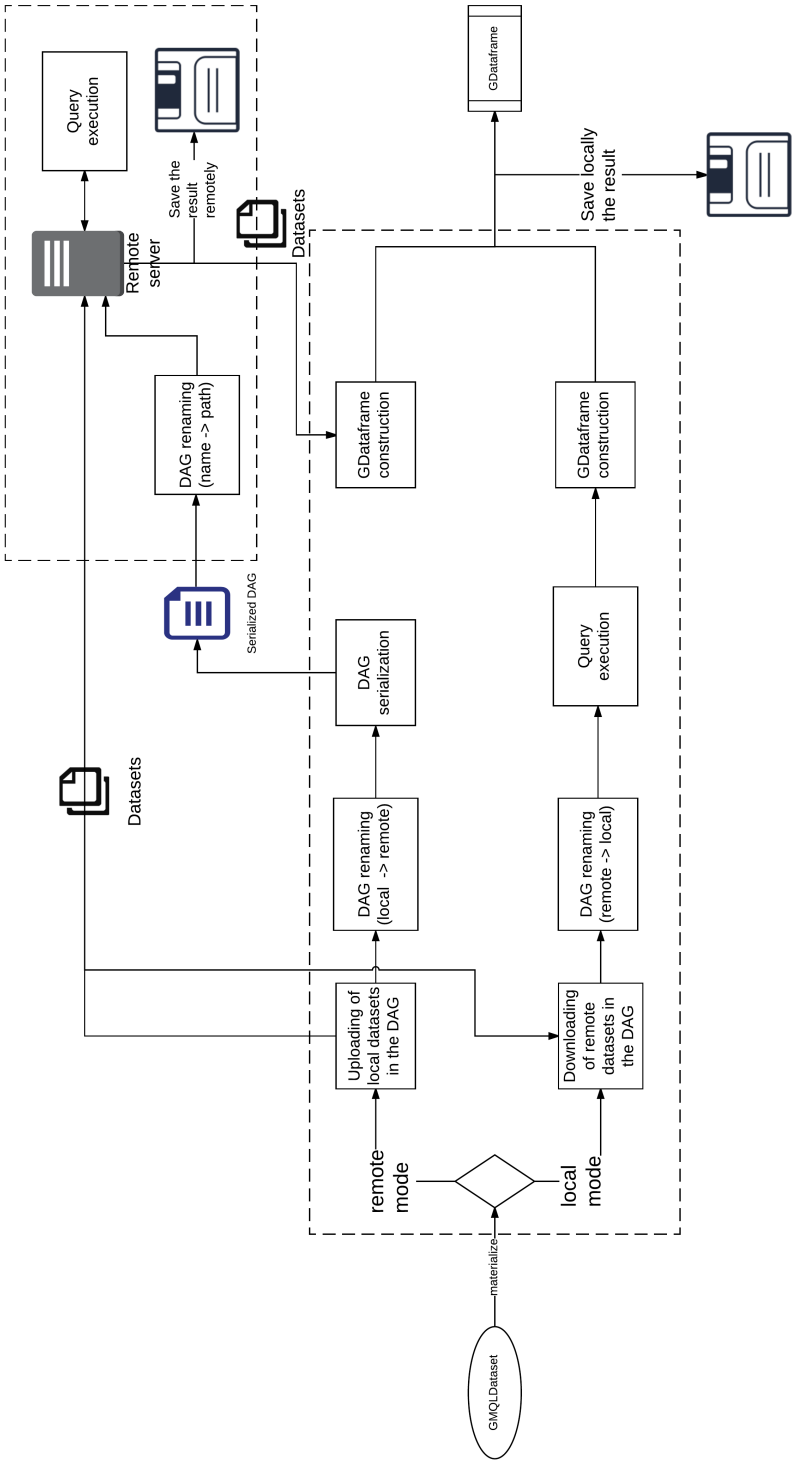


Figure 4.5: The process of materialization depending on the execution mode of PyGMQL. In the image we can see both the execution at the API level and at the server level. Each square represents a step and the arrows represent information exchange.

the remote infrastructure). In particular we need to rename some nodes in the graph in the following cases:

- *PyGMQL side*
  - If we are in local mode and we are using in our query some remote dataset, we need to change the name of the dataset to the local path (where it was downloaded).
  - If we are in remote mode and we are using in our query some local dataset, we need to change the local path of the dataset to the remote name (which were chosen when it was uploaded).
- *Remote server side*: every time the server receives a DAG to process, all the nodes related to the loading of data contain the *name* of the dataset and not its actual *location*. For example, if the server uses Hadoop as file system, then all the nodes must be changed to the actual path. The server uses the repository

The procedure for graph renaming is recursive and search in the DAG all the `ReadMD` and `ReadRD` nodes and change them depending on the situation. In figure 4.6 we can see an example.

4.3 Interfacing with the Machine Learning module

Contemporary to the development of the PyGMQL project also a Machine Learning module specialized on genomics was developed by Anil Tuncel for its master thesis. The main idea was to exploit the features of PyGMQL to build on top of it the machine learning algorithms.

The machine learning algorithms and in particular the clustering methods require a specific data structure that we call *Genomic Space*. An additional effort of the PyGMQL project was the definition of the data structure.

The genomic space is a specialization of a GDataframe and therefore can be used *only after the materialization* of a query. Therefore the methods developed by Anil work entirely on Python and do not exploit GMQL functionalities.

4.4 Deployment and publication of the library

Python packages can be distributed to the community through a service called *Python Package Index* (PyPI) [6]. PyPI is a public repository where python *packages* are stored. This enables end users to download easily the

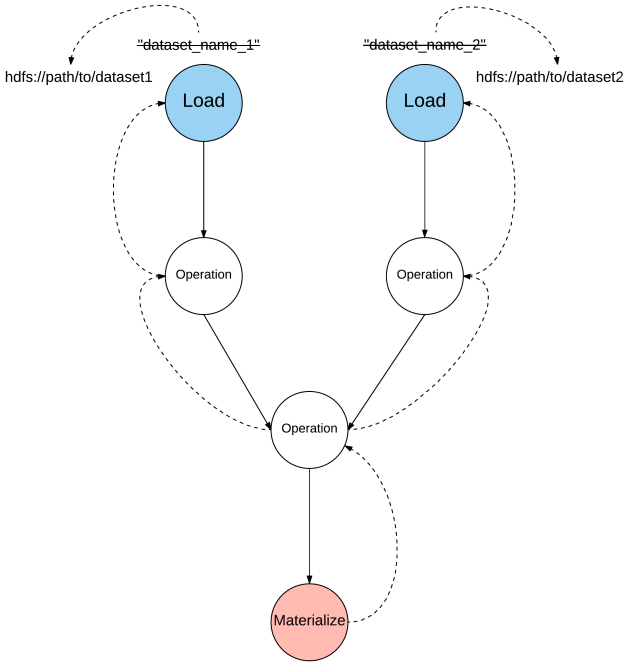


Figure 4.6: Example of DAG renaming where the server has to rename the dataset names, in all the loading nodes, to Hadoop file system paths

python packages that they want without knowing the actual location of it. For example, if a user wants to install PyGMQL in his system he only needs to execute the following command:

```
$ pip install gmql
```

In order to distribute a package like PyGMQL the following steps must be performed:

1. The "packager" of the library must *register* the name of the package to the repository. In our case we choose `gmql` as library name.

2. The python package must be prepared for being distributed. This requires the definition of a `setup.py` file which describes the project, lists the dependencies and the requirements that the user must satisfy to use the package. For example: the required version of Python, the python external libraries that were used in the development, the author name and email, and so on.

3. A *source distribution* must be done. This is basically a compressed file containing the source files of the package. In addition the package

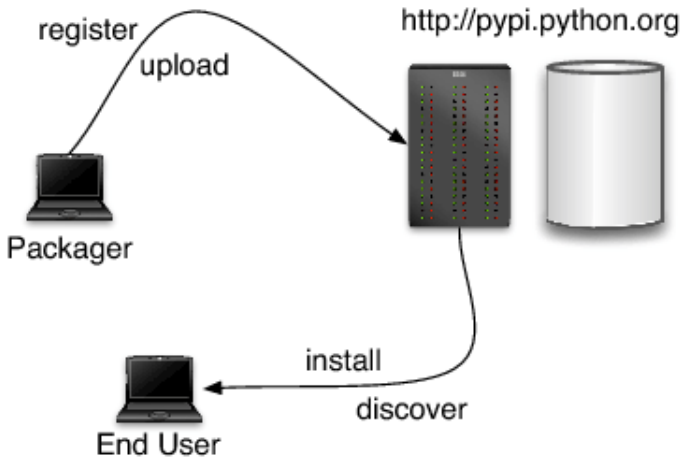


Figure 4.7: PyPi work-flow from the point of view of both the package maintainer and final user

developer can also create a *wheel*, which is a pre-build file that does not require the user to build the source code.

4. The last step is to upload the various created files (source distribution and wheels) to the PyPI repository.

In figure 4.7 we can see a schematic representation of the process.

Chapter 5

Language mapping

"Well, here's another nice mess you've gotten me into!"

Stan and Oliver - Sons of the Desert

In this chapter we continue with the description of the library. In particular we deepen into the description of the mapping of the GMQL operator to python functions. We will describe also the data structures that hold the data of a GMQL dataset before and after the materialization.

The discourse will be organized as follows:

1. We will describe the GMQLDataset structure and the rationale behind its design
2. We will proceed by reporting, for each GMQL operation, its mapping into a GMQLDataset method
3. Then we will describe the GDataframe structure, its internals and its usage
4. We will conclude by doing a commented full example of usage of PyGMQL

5.1 Query creation: the GMQLDataset

A GMQLDataset is a python object that represents a GMQL variable. All GMQL operators can be applied to a GMQLDataset and they return an other GMQLDataset.

```
new_dataset = dataset.operation(paramters, ...)
```

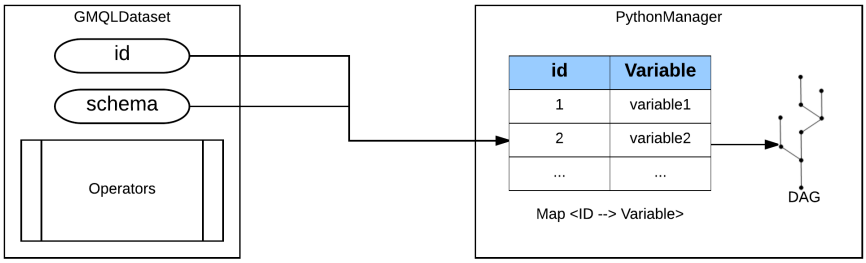


Figure 5.1: Interaction between a GMQLDataset and the Scala back-end for storing the variable information

This logic mimics a classical GMQL query and it is coherent with the concept of closed algebra of GMQL, like it has been stated in chapter 2.

This component interacts with the Scala back-end for building the DAG of the queries. In figure 4.3 some components of the Scala back-end are shown and how they interact with the python side.

The logic behind the construction of a GMQL query works like follows:

1. Each GMQLDataset has associated an *id*. This identifier is used for referring to the internal variable in the Scala back-end that represent the sub-graph of the full DAG of the query.

2. The Scala back-end holds a data structure (hash map) that associate each identifier to the variable. See figure 5.1

3. In order to apply an operation to a GMQLDataset, the python side orders to the back-end to perform a specific operation on the specified *id*.

4. The Scala back-end builds the new DAG adding the new operation at the bottom. Generates a new identifier and puts a new record in the hash map. It ends returning to the python side the new *id* for the new GMQLDataset.

5. Since the schema of the new generated dataset can be computed before the actual query execution, the GMQLDataset gets also the schema of the new variable. This is very useful for performing error checking interactively during the query construction.

We proceed by describing the signatures of the operators that can be applied to a GMQLDataset. They mirror the original GMQL operators but takes as arguments python data structures and data types.

5.1.1 Selection

Here the design choice was to *split* the selection on region data from the one on metadata. This is done to reduce the amount of arguments of the following functions.

Both functions takes a selection predicate as input. This is obtained through a boolean expression that considers region fields or metadata fields. For example

```
(dataset['cell'] == 'imr90') & (dataset['antibody_target'] == 'CTCF')
```

is a predicate on metadata that selects the samples concerning cell IMR90 and the antibody target CTCF; while

```
(dataset.start > 100000) | (dataset.stop < 500000)
```

is a predicate on region fields that selects the regions that start after position 100000 or end before position 500000.

Follows the signature for the selection on metadata

```
GMQLDataset.meta_select(self,
              predicate=None,
             semiJoinDataset=None,
             semiJoinMeta=None)
```

Follows the signature for the selection on region data

```
GMQLDataset.reg_select(self,
             predicate=None,
             semiJoinDataset=None,
             semiJoinMeta=None)
```

The `semiJoinDataset` is another GMQLDataset and has the same role of the `semijoin` clause in the GMQL `SELECT` and the `semiJoinMeta` is a list of metadata attributes.

5.1.2 Projection

Using the same rationale of selection, we split also the meta projection from the region one.

Follows the signature for the projection on metadata

```
GMQLDataset.meta_project(self,
              attr_list=None,
               all_but=None,
               new_attr_dict=None)
```

Follows the signature for the projection on regions

```
GMQLDataset.reg_project(self,
             field_list=None,
             all_but=None,
             new_field_dict=None)
```

`attr_list` and `field_list` are respectively list of metadata attributes and region fields. Same for the `all_but` parameter, which has the opposite semantics (take all the metadata attributes / region fileds that are *not* in that list).

`new_attr_dict` and `new_field_dict` are two dictionaries of the following type:

```
new_attr_dict = {
    "new_metadata_attribute" = <expression_on_metadata>,
...
}

new_field_dict = {
    "new_region_field" = <expression_on_region_fields>,
...
}
```

where `<expression_on_metadata>` is an expression built using metadata attributes from the dataset. For example `dataset['n_regions']/dataset['variance']` computes the division of the metadata `"n_regions"` and `"variance"` for each sample in the dataset; while `dataset.pValue / 2 + 25` computes, for each region, the half of the `"p_value"` field plus 25.

5.1.3 Extension

Extension uses a logic similar to projection for building new metadata from region aggregates.

```
GMQLDataset.extend(self, new_attr_dict)
```

`new_attr_dict` is a dictionary of the following type:

```
new_attr_dict = {
    "new_metadata": aggregate_function("region_field"),
    ...
}
```

where `aggregate_function` is one of the classical GMQL aggregate functions, mapped in python like in table 5.1.

GMQL aggregate	PyGMQL aggregate
COUNT	`gmql.COUNT()`
SUM	`gmql.SUM("field")`
MIN	`gmql.MIN("field")`
MAX	`gmql.MAX("field")`
AVG	`gmql.AVG("field")`
BAG	`gmql.BAG("field")`
STD	`gmql.STD("field")`
Q1	`gmql.Q1("field")`
Q2	`gmql.Q2("field")`
Q3	`gmql.Q3("field")`

Table 5.1: Mapping between the GMQL aggregate operator and their equivalent in PyGMQL

5.1.4 Genometric Cover

The signature for a generic genometric Cover is the following

```
GMQLDataset.cover(self,
           minAcc, maxAcc,
           groupBy=None,
           new_reg_fields=None,
           type="normal")
```

The variants of the GMQL Cover can be specified in the `type` field (`{"normal", "flat", "summit", "histogram"}`). Equivalently one can call directly the following functions using the same set of parameters (without `type`)

```
GMQLDataset.normal_cover(self, ...)
GMQLDataset.flat_cover(self, ...)
GMQLDataset.summit_cover(self, ...)
GMQLDataset.histogram_cover(self, ...)
```

`minAcc` and `maxAcc` play the same role as their homonyms in GMQL and are integer numbers (or the strings `"ALL"` and `"ANY"`).

The `groupBy` is a list of metadata attributes that we can use to group the results.

`new_reg_fields` is a dictionary of the type

```
new_field_dict = {
    "new_region_field" = aggregate_function("region_field"),
...
}
```

where `aggregate_function` is one of the aggregate function in table 5.1.

5.1.5 Join

The signature for a genometric Join is the following

```
GMQLDataset.join(self,
           experiment,
           genometric_predicate,
           output="LEFT",
           joinBy=None,
           refName=None,
           expName=None)
```

where `experiment` is another GMQLDataset. The semantics of this operator is that the *caller* GMQLDataset is the *reference* while the GMQLDataset used as parameter is the *experiment* of the Join.

The `genometric_predicate` is an ordered list of genometric predicates. The mapping between the GMQL genometric predicates and the ones of PyGMQL is shown in table 5.2

GMQL genometric predicate	PyGMQL genometric predicate
DLE	`gmql.DLE(value)`
DGE	`gmql.DGE(value)`
DLE	`gmql.DLE(value)`
MD	`gmql.MD(value)`
UP	`gmql.UP()`
DOWN	`gmql.DOWN()`

Table 5.2: Mapping between the genometric predicates of GMQL and the ones of PyGMQL

`output` is a string that can assume the values `{'LEFT', 'RIGHT', 'INT', 'CONTIG'}` with the same semantics of the original language.

`joinBy` is a list of metadata attributes

With `refName` and `expName` the user can also specify the name to give to the reference and experiment dataset in the result.

5.1.6 Map

The signature for a genometric Join is the following

```
GMQLDataset.map(self,
           experiment,
           new_reg_fields=None,
           joinBy=None,
           refName=None,
           expName=None)
```

As in the Join operation, the `experiment` is an other GMQLDataset and the semantics is the same.

`new_reg_fields` is a dictionary of the type

```
new_reg_fields = {
    'new_region_field': aggregate_function("region_field"),
    ...
}
```

like for the Cover operator.

The `joinBy` is a list of metadata attributes and `refName` and `expName` have the same role as in the Join operator.

5.1.7 Order

The order operation is pretty straightforward as it is very similar to the GMQL equivalent:

```
GMQLDataset.order(self,
           meta=None,
           meta_ascending=None,
           meta_top=None,
           meta_k=None,
           regs=None,
           regs_ascending=None,
           region_top=None,
           region_k=None)
```

We can order based on metadata and regions by specifying `meta` and `regs` and we can decide the ascending/descending order using le boolean lists `meta_ascending` and `regs_ascending`. If the i-th elements of these lists is 1, it means that the order for that attribute/field will be ascending; if it is 0 it will be descending. `meta_top` and `regs_top` can assume the values `{"top", "topq", "topp"}` and `meta_k`, `regs_k` are the relative values.

5.1.8 Difference

```
GMQLDataset.difference(self,
             other,
             joinBy=None,
             exact=None)
```

`other` is another GMQLDataset and `joinBy` is, as always, a list of metadata attributes.

5.1.9 Union

```
GMQLDataset.union(self,
             other,
             left_name="",
             right_name="")
```

`other` is another GMQLDataset while `left_name` and `right_name` are optional parameters that give a name to the left (caller) and right (parameter) datasets. They are used in the creation of the new attributes.

5.1.10 Merge

```
GMQLDataset.merge(self, groupBy=None)
```

`groupBy` is a list of metadata attributes.

5.1.11 Group

The group operator is still under development also in the declarative version of GMQL. Anyway the signature of the method has been defined both in PyGMQL and in GMQL.

```
GMQLDataset.group(self,
           meta=None, meta_aggregates=None,
           regs=None, regs_aggregates=None,
           meta_group_name="_group")
```

There exist also a metadata specific version of the operator

```
GMQLDataset.meta_group(self,
           meta, meta_aggregates=None)
```

and a region specific one

```
GMQLDataset.regs_group(self,
           regs, regs_aggregates=None)
```

`meta` and `regs` are lists of metadata attributes or region fields. `meta_aggregates` and `regs_aggregates` are dictionaries of aggregates functions for metadata attributes and region fields.

5.1.12 Materialization

Follows the signature for the materialization method:

```
GMQLDataset.materialize(self,
                output_path=None,
                output_name=None,
                all_load=True)
```

The materialization procedure has a different semantics depending on the mode in which PyGMQL is set:

- *Local mode*: the computation must be performed locally. If the user does not specify anything (all parameters set to `None`) the local computation engine produces the result and transfer it directly into a python GDataframe without saving it. If the user specifies `output_path` then the result is also saved in a set of files (two for each sample, one for regions and one for meta). In local mode `output_name` and `all_load` are not effective
- *Remote mode*: the computation is performed remotely. The user can specify the name that will be given to the resulting dataset in the remote server using `output_name`. If the user specifies `output_path` or `all_load` then the result is downloaded directly into a local GDataframe.

Only in local mode, the user can also ask to materialize only n samples from the result. This can be done with the following function:

```
GMQLDataset.take(self, n)
```

5.2 Results management: the GDataframe

Once a query has been materialized or the user wants to directly load in memory a GMQL dataset, we have to define an appropriate data structure

to hold all its information. For this task the GDataframe has been designed.

The GDataframe is composed of two tables:

- *Region data*: one row for each region in the dataset and one column for each region field
- *Metadata*: one row for each sample in the dataset and one column for each metadata attribute

Each row of these tables is associated with an *id*, which represents the sample of the dataset that row comes from. Formally, if $\mathcal{R}$ is the table of region data and $\mathcal{M}$ is the table of metadata, we have

$$\mathcal{R} = \{\text{sample id} \rightarrow [\mathbf{r}_1, \mathbf{r}_2, ...]\}$$

where $\mathbf{r}_i$ is a vector of values for each region ordered and typed based on the dataset schema

$$\mathcal{M} = \{\text{sample id} \rightarrow \mathbf{m}\}$$

where $\mathbf{m} = \{\text{metadata attribute} \rightarrow \mathbf{v}\}, \mathbf{v} = [\text{value}_1, \text{value}_2, ...]$. Every sample in the dataset must have the same metadata attributes but can have different values for each of them. Notice the possibility to have multiple values for the same attributes.

Of course these two data structures must be coherent in the sense that the keys (the sample ids) must be the same.

For a visual representation of the relationship between $\mathcal{R}$ and $\mathcal{M}$ take a look at figure 5.2.

In python this structure is implemented with two `Pandas`[1] dataframes.

The API provides methods for importing into a GDataframe a generic Pandas dataframe, provided that it has certain characteristics:

- If the data are regions, then there must be at least the columns for chromosome, start and stop. Every other column is up to the user and there can be an unlimited number of columns
- If no column for the sample id is provided, the API supposes that there is only one sample in the dataset and puts the sample column to one single value in both region and metadata table
- If no metadata table is provided, then an empty table is generated having only the sample column coherent with the region one.

[1]Pandas is a very famous and established data analysis library. It is used both by researchers, data scientists and machine learning engineers for storing data in a tabular format

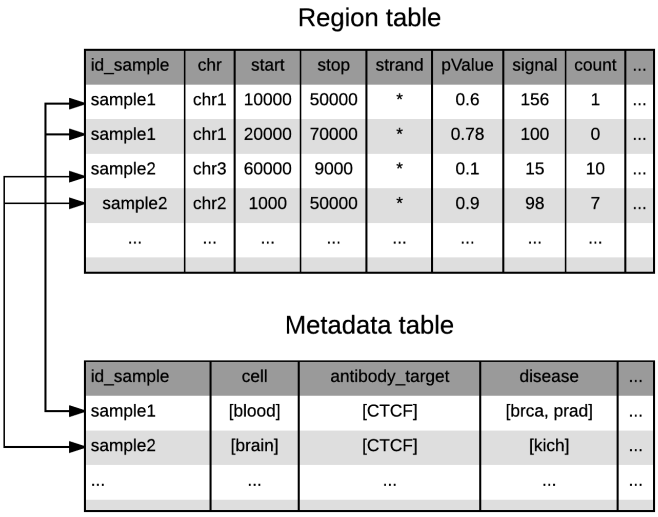


Figure 5.2: Visual representation of the region data and the metadata structures in a GDataframe. Notice how the sample ids must be the same and coherent in both the tables.

5.3 Example

Here we provide a simple example of usage of the library. One can see this as a brief tutorial that explore some of the most important functionalities of PyGMQL.

As any python package it must be loaded as follows

```
import gmql as gl
```

Since we are going to use the remote service we also need to authenticate ourselves to the system, in order to use its functionalities and to access our private remote dataset and also the public ones.

```
gl.login(username="luca", password="password_luca")
```

The first thing that the user wants to do is to load some dataset. This can be done using a local dataset or a remote one. Let's take as an example a remote dataset.

```
remote_dataset = gl.load(remote_name="remote_dataset")
```

This action basically only downloads the dataset schema and creates a local

pointer to the remote dataset. No real action is performed yet.

Let's say that we want to filter this dataset and take only the regions that are on chromosome 11 and they start after 100000 bases

```
filtered_remote_dataset = remote_dataset.reg_select(
              (remote_dataset.chr == 'chr11') &
              (remote_dataset.start > 100000))
```

Now consider having a local dataset that is stored in one tab separated file having as columns `chr, start, stop, gene` but without any metadata or schema file. It is a simple file representing genomic regions and the gene name associated to that region. This is clearly not a GMQL dataset since it does not respect its requirements.

We can import this file in a GDataframe and then convert it in a GMQL-Dataset which is a GMQL standard dataset. This will generate a schema file and an empty metadata file. The resulting dataset will be local.

```
import pandas a pd
gene_dataset = pd.read_csv("path/to/gene_dataset.tsv", sep="\t")
gene_dataset = gl.from_pandas(gene_dataset)
gene_dataset = gene_dataset.to_GMQLDataset()
```

Now we want to join these two GMQL dataset using associating to each left region the right regions that are distant less than 100 kb

```
join_dataset = gene_dataset.join(
      experiment=filtered_remote_dataset,
      genometric_condition=[gl.DLE(100000)])
```

Now we want to materialize the result into a Python structure. As we have seen the behavior of the library depends on its status. By default PyGMQL is set in *local* mode. Suppose that we have now changed our mind and we want to perform this operation remotely:

```
gl.set_mode("remote")

result = join_dataset.materialize(output_path="where/to/save/the/result")
```

In this way PyGMQL will upload to the remote server the GMQL dataset `gene_dataset`, send the serialized DAG to the server, wait for the execution of the query and finally download the results into a GDataframe, which is put in the `result` variable. In addition it will also save the downloaded files in the `"where/to/save/the/result"` location in the local machine.

In the chapter 6 are present lots of other examples that show also the management of the GDataframe structure after a query.

Chapter 6

Biological applications

> *"My name is Maximus Decimus Meridius, commander of the Armies of the North, General of the Felix Legions and loyal servant to the true emperor, Marcus Aurelius. Father to a murdered son, husband to a murdered wife. And I will have my vengeance, in this life or the next."*
>
> Gladiator

In this chapter we will test the developed package on real biological problems.

During the development of the library a set of biological problems were addressed in order to:

- Test the library functionalities and do bug checking
- Acquire additional requirements and suggestions to improve the quality of the project and add new functionalities

We can say that the direct application of the library to these problems was the main tool used to evaluate the performances, the usability and the versatility of the python package.

In particular this chapter will be organized as follows:

1. A set of initial examples of GMQL biological queries (the same that were discussed in 2) will be transposed to python pipelines for showing the mapping between the two languages
2. We will deepen into the research problem of topological domains. In particular, after having described how to extract the TADs from raw Hi-C data and the various datasets used during this study, we will show:

(a) how gene expression is related to topological domains both in the case of normal and tumoral tissues

(b) if topological domains conserve across different species

(c) how TADs cluster together based on gene expression and ChIA-PET links

6.1 Some examples

As first examples we can translate the queries shown in section 2.3.4 to python programs. This is only to show the direct mapping between the classical GMQL operations and the API ones.

Find somatic mutations in exons

The query reported at 2.3.4 can be translated as follows:

```
import gmql as gl

# as first thing we need to login to the remote service
gl.login("username", "password")
# we want to perform the query remotely
gl.set_mode("remote")

mut = gl.load(name = "HG19_TCGA_dnaseq")
mut = mut[ (mut['manually_curated|dataType'] == 'dnaseq') &
       (mut['clinical_patient|tumor_tissue_site'] == 'breast') ]

exon = gl.load(name = "HG19_BED_ANNOTATION")
exon = exon[ (exon['annotation_type'] == 'exons') &
        (exon['original_provider'] == 'RefSeq') ]

exon1 = exon.map(experiment = mut)
exon2 = exon1.reg_select(exon1.count_Exon_Mut >= 1)
exon3 = exon2.extend({
             'exon_count': gl.COUNT()
            })
exon_res = exon3.order(meta=['exon_count'], meta_ascending = [0])

# local materialization: this functions starts the computation
# remotely, downloads the result locally and puts it into the
# pandas tabular structure that we described
exon_res = exon_res.materialize()
```

```
# from now on we can continue working in python with the materialized
# data structures
```

Find distal bindings in transcription regulatory regions

The query reported at 2.3.4 can be translated as follows:

```
import gmql as gl

# in this example we want to process the data locally.
# Therefore we do not need to login to the remote service.
# We do not even need to set the mode with "gl.set_mode("local")"
# since it is the default behavior

tf = gl.load(name="HG19_ENCODE_NARROW")
tf = tf[ (tf["dataType"] == "ChipSeq") &
       (tf["view"] == "Peaks") &
       (tf["antibody_target"] == "CTCF") ]

hm = gl.load(name="HG19_ENCODE_BROAD")
hm = hm[ (hm["dataType"] == "ChipSeq") &
       (hm["view"] == "Peaks") &
       (hm["antibody_target"] == "H3K4me1") ]

bed_annotation = gl.load(name="HG19_BED_ANNOTATION")

tss = bed_annotation[ (bed_annotation["annotation_type"] == "TSS") &
                (bed_annotation["provider"] == "UCSC") ]
en = bed_annotation[ (bed_annotation["annotation_type"] == "enhancer") &
                (bed_annotation["provider"] == "UCSC") ]

tf1 = tss.join(tf, [gl.DGE(100000), gl.MD(1)], output="right")
hm1 = tss.join(hm, [gl.DGE(100000), gl.MD(1)], output="right")
hm2 = en.join(hm1, [gl.DLE(0)], output="int")

tf_res_0 = tf1.map(hm2, joinby=["cell"])
tf_res = tf_res_0.reg_select(tf_res_0.count_hm2_tf1 > 0)

# The materialization is done locally
tf_res = tf_res.materialize()
```

These examples show how one can easily map a GMQL query to a python program. We cannot say that it is possible to do the reverse in all the cases. The intrinsic power of python, its additional libraries and the possibility

to work autonomously on the resulting data and to use personal crafted ones in GMQL queries makes this new computational environment way more powerful and complete than the original architecture.

In the next section we will apply the library to a very complex and unsolved biological problem: we will try to understand the role and how *Topologically Associated Domains* (TADs) are organized in the genome.

6.2 TADs research

In chapter 1 we have briefly discussed the main components of the genome and how it is structured.

We didn't put a lot of attention on the *3D* structure of the chromosomes. Since they are molecules, they have also a 3D shape and this characteristic was studied since the first days of genomics.

In the last years a lot of attention was put on the study of chromosome organization, also thanks to the new advanced method and the huge amount of data that has been collected [13]. This renewed focus on these aspects has put the understanding of the 3D structure of chromosomes, and in general of the genome, at the higher level of research importance.

It has been proven that inside chromosome there exist genomic regions within which the physical interactions occur much more frequently than out of them. These regions were called Topologically Associated Domains and they can be seen as a "summarization" of the 3D conformation of the genome [11]. The size of a TAD can vary from a few thousand to millions of bases.

During the project we tried to gain more knowledge about the role of these genomic regions, how they interact with each other and if they form higher level structures.

In the next section we briefly describe the main process for finding the TADs given interaction data between zones of the genome.

6.2.1 Extracting the TADs

The Hi-C method [18] is a specific procedure belonging to the set of *Chromosome conformation capture* (3C) techniques. The aim of this algorithm is to unravel the spatial organization of chromatin[1] in a cell. The steps are the following:

1. We *cross-link* the chromatin. This operation connects the parts of DNA that are spatially close

[1] With the term *chromatin* we mean the set of macromolecules consisting of DNA, protein and RNA

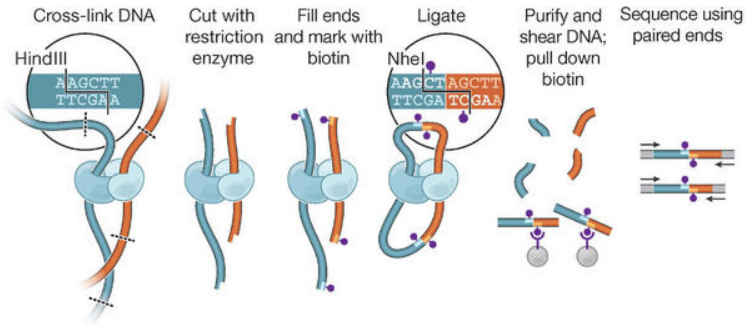


Figure 6.1: Schematic representation of the Hi-C method

2. We then cut the DNA in *bins* using a predefined resolution (in [11] they used a resolution of 40 kb)

3. The loose ends of the cross-linked DNA are ligated together. The result is a set of pairs of connected DNA fragments

4. We apply *pair-ended sequencing* to identify these pairs

In figure 6.1 we can see a visual representation of the process.

The output of Hi-C is a matrix I where the cell I_{ij} represents how many times bin i interacted with bin j. Obviously this matrix is symmetric.

Using this method and applying later a directionality index, which is fed into an Hidden Markov Model[2], [11] found out that there exist *domains* with a lot of *intra-domain* interactions less *inter-domain* interactions. Pairs of regions inside these domains are closer than pairs in different domains. This demonstrate that the 3D structure of the genome il constituted of self-interacting segments. A visual representation of this concept can be seen at figure 6.2.

It has been also shown that the areas between TADs, also called *domain boundaries* have important features: they have been associated with transcription start sites and CTCF[3] binding sites[4].

After [11] other methods were developed: [12] proposed an alternative to the previous method. They argue that TADs are nested and that their method can find them at different scales. In [26] they used new high res-

[2]The *directionality index* of a bin is a function of the ratio between the number of upstream and downstream interactions for that bin

[3]CTCF is a transcription factor encoded by the homonym gene that is involved in a lot of cellular process and in particular affects the regulation of chromatin architecture

[4]With *binding site* we mean a zone of the genome that proteins like transcription factors and other use to bind themselves to.

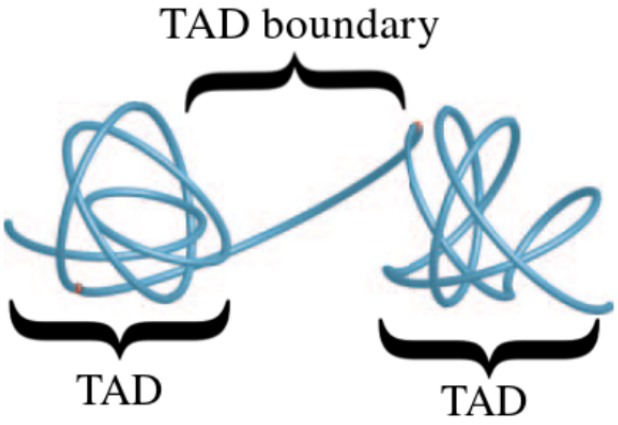


Figure 6.2: Schematic representation of two TADs and the boundary between them. Taken from [11]

olution (the new bin size is around 1 kb) Hi-C data getting more precise results.

6.2.2 Overview of the used data

TADs datasets

During our experiments we used TADs extracted both from the Dixon work [11] and the Rao one [26]. In particular these two dataset differ both in the cell lines that they describe and the size of the topological domains. In table 6.1 we can see the different cell lines in each of these two datasets.

In figure 6.3 we can see some statistics about these two dataset including mean size of the TAD, number of TADs for each cell line and so on. In figure 6.4 we can also see the distribution of sizes in the TADs, for each cell line.

Expression dataset

In the following we will put a lot of attention of the *expression* profiles of the various TADs. Expression is usually associated to genes and represent the process by which the information contained in a gene is used to synthesize the functional product of it. These products are proteins, RNA or other molecular structures. In figure 6.5 we can see a representation of this process.

Different cell types express different genes and this is the main characteristic that differentiates the tissues between each other. Gene expression is

TAD dataset	Cell lines
Dixon [11]	• Embryonic stem cells (ES) • Lung (IMR90)
Rao [26]	• B-Lymphocyte (GM12878) • Mammary epithelial cells (HMEC) • Umbilical vein endothelial cells (HUVEC) • Immortal cells (HeLa) • Lung (IMR90) • Lymphoblast (K562) • Myelogenous leukemia (KBM7) • Normal human epidermal Keratinocytes (NHEK)

Table 6.1: Cell lines for each TAD dataset considered in the study

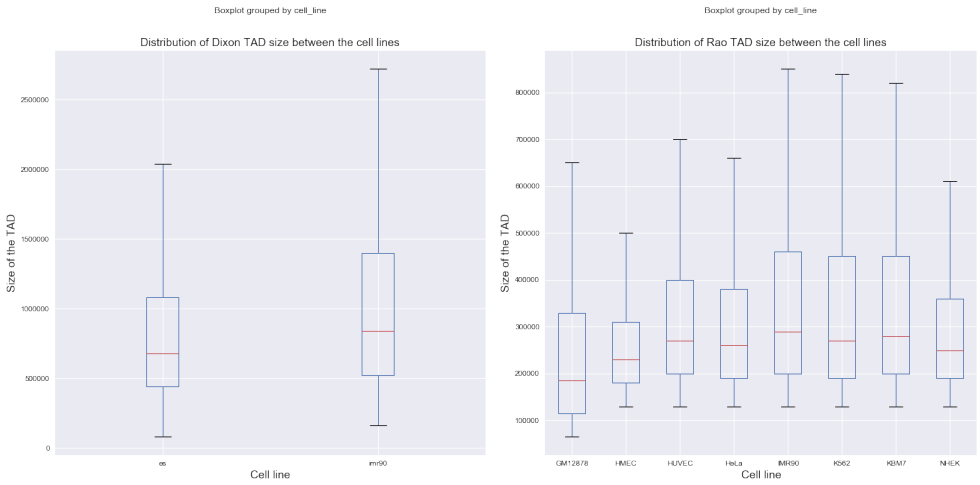
	mean size	max size	min size	count
es	843930	4440000	80000	2931
imr90	1101302	6480000	160000	2207

(a) Descriptive data about Dixon [11] TADs

	mean size	max size	min size	count
GM12878	257263	2690000	65000	9048
HMEC	325147	2530000	130000	4010
HUVEC	407758	6580000	130000	4117
HeLa	361968	5360000	130000	4344
IMR90	383654	5450000	130000	7486
K562	372191	3830000	130000	5817
KBM7	376260	3460000	130000	4771
NHEK	316199	4240000	130000	5826

(b) Descriptive data about Rao [26] TADs TADs

Figure 6.3: Descriptive data about the two TADs dataset used in the study



(a) Distribution of TAD sizes in Dixon [11] TADs

(b) Distribution of TAD sizes in Rao [26] TADs

Figure 6.4: Distribution of sizes in the two TADs dataset used in the study

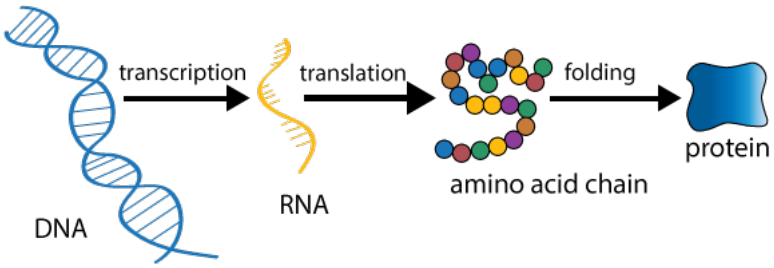


Figure 6.5: Schematic illustration of the gene expression process

modified both by internal processes[5] or external ones: cancer, for example, is often associated with a modified gene expression localized in particular zones of the genome.

The measurement of gene expression today is usually done through a technique called *RNA sequencing*. RNA-seq is composed of the following phases:

1. *Library preparation*: the first step is to prepare a collection of DNA fragments obtained by complementing the RNA. This is done by firstly *isolating* the RNA from the DNA, *selecting* the signals of interest (for example, if we only want to analyze mRNA we need to add some chemical components to link to that structure) and *synthesize* the DNA using reverse transcription.

2. *Sequencing*: now we have to assign to these raw sequence reads the genomic features. This is done by aligning the sequences to the reference genome

3. *Gene expression extraction*: expression is quantified by counting the number of reads that were mapped to each zone of the transcriptome.

From this we can model this kind of data like follows: given a total number of genes G and a number of samples I, the expression Y_{gi} is an integer number representing the number of times a read was mapped to gene g for sample i. Typically the distribution of data is not normal and a further preprocessing step is performed: we compute the *Reads per Kilobase Million* as follows [9]:

$$RPKM_{gi} = \frac{10^9 \cdot Y_{gi}}{\sum_{g'=1}^{G} Y_{g'i} \cdot L_g}$$

where L_g is the length of gene g and $\sum_{g'=1}^{G} Y_{g'i}$ is the total number of counts for sample i.

During our experiments we used two different datasets containing gene expressions:

- *Genotype-Tissue Expression* (GTEx) [24]: contains gene expressions from health individuals divided by tissue. In figure 6.6 we can see the number of samples for each tissue in GTEx.

- *The Cancer Genome Atlas* (TCGA) [14]: gene expressions both from cancer cells and normal cells

[5] For example there exists DNA sequences called *enhancers* that promote the expression of a particular gene. On the other side there are sequences the inhibits the expression

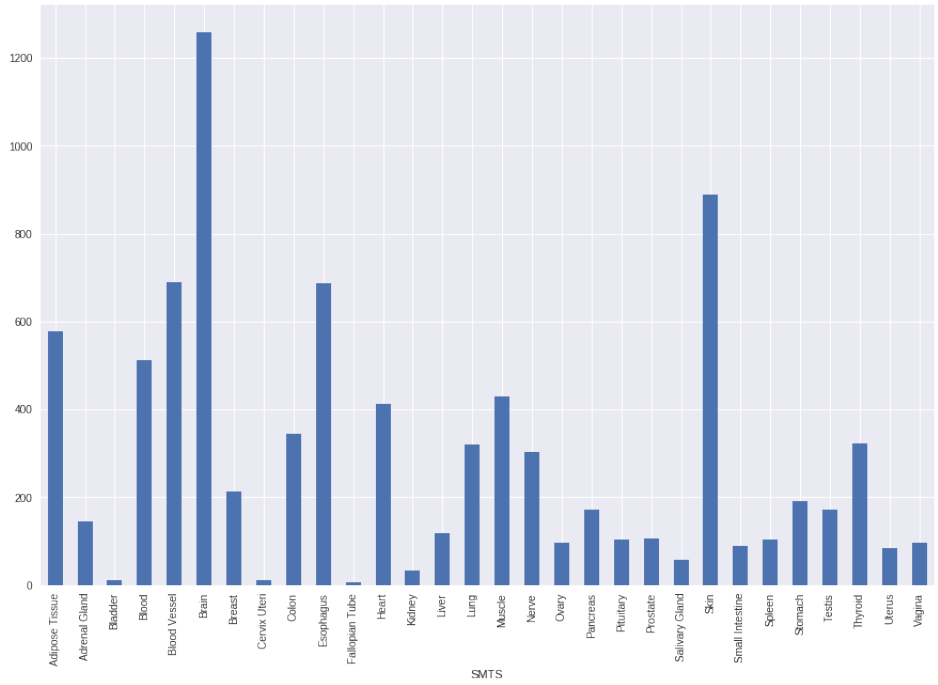


Figure 6.6: Distribution of samples in the different tissues present in the GTEx database

6.2.3 Correlations between gene pairs inside and across TADs

The first thing that we tried to do was to confirm the results that are reported in the literature. In particular we wanted to find the difference in correlation between genes that are inside the *same* TAD and genes that are *across* different TADs.

We followed this procedure:

1. Given a TADs dataset T, consider the data from two cell lines c_1, c_2. We obtain two TADs datasets T_1 and T_2

2. Given a gene expression dataset G, map each gene g_i to the TAD it belongs to in both $T_1 = T\,[cell == c_1]$ and $T_2 = T\,[cell == c_2]$

3. We consider as *same* the genes that falls in the same TAD in both T_1 and T_2 and *cross* the ones that do not fall in the same TAD. The genes that are *same* in just one of T_1 and T_2 are disregarded.

The expressions of the genes were taken from GTEx. Each experiment involved a specific tissue or a set of them: in particular we used brain, breast, liver, muscle and an ensemble of brain, breast and liver.

For each experiment we plot the *Pearson correlation* of the expression of the genes in the *same* and *cross* set as a function of the distance between the two genes. In particular, given the expression vector $\mathbf{x}^{g_1}$ relative to gene g_1 and $\mathbf{x}^{g_2}$ relative to gene g_2, the Pearson correlation coefficient of $\mathbf{x}^{g_1}$ and $\mathbf{x}^{g_2}$ is

$$\text{pearson}(\mathbf{x}^{g_1}, \mathbf{x}^{g_2}) = \frac{\sum_{i=1}^{n} \left(x_i^{g_1} - \overline{x^{g_1}}\right)\left(x_i^{g_2} - \overline{x^{g_2}}\right)}{\sqrt{\sum_{i=1}^{n} \left(x_i^{g_1} - \overline{x^{g_1}}\right)^2}\sqrt{\left(x_i^{g_2} - \overline{x^{g_2}}\right)^2}}$$

This plot is obtained using *Locally Weighted Scatterplot smoothing*[6] on the data points for the various couples of genes. We plot also the mean correlation between the pairs in *same*, *cross* and *random* couples of genes in order to assess the different data distribution.

6.2.4 GMQL query and Python pipeline

The creation of the *same* and *cross* set can be done using GMQL. Follows an example of GMQL query that uses gene expressions from the cortex, the TADs cell lines from lung and B-Lymphocyte and a dataset of gene coordinates.

```
GENES = SELECT(tissue=="cortex") GTEx;
J1 = SELECT(cell=="imr90") TADs;
J2 = SELECT(cell=="gm12878") TADs;
PAIRS = JOIN(distance < 500000; CONTIG) GENES GENES;
MAPPING_t = MAP() PAIRS L1;
MAPPING = MAP() MAPPING_t L2;
SAME = SELECT(left.id < right.id and L1.count == 0 and L2.count == 0) MAPPING;
CROSS = SELECT(left.id < right.id and L1.count > 1 and L2.count > 1) MAPPING;
```

The execution of this query has as assumption to have the GTEx dataset available in GMQL format. This is not a simple task to achieve. Here we demonstrate the power of mixing Python data management with GMQL computations.

```
# location of the datasets
coords_path = "./TSS_coordinates/" # gene coordinates
TADs_path = "./Dixon_TADs/"   # TADs dataset
```

[6] LOESS is a technique used to fit a scatterplot to a line. At each point in the initial dataset we fit a low degree polynomial to a small subset of data.

```
# Since the datasets that we are using are not standard, we need to define
# a personalized parser for each of them.
coords_parser = gl.parsers.BedParser(parser_name="coordinates_parser",
                            chrPos=0, strandPos=None, startPos=2,
                                stopPos=3,
                            delimiter="\t", otherPos=[(4,"gene","string")])

tads_parser = gl.parsers.BedParser(parser_name="tads_parser",
                           chrPos=0, strandPos=3, startPos=1, stopPos=2,
                           delimiter="\t")
# loading the coordinates dataset
genes_coords = gl.load_from_path(coords_path, parser=coords_parser)
# loading the TADs dataset
tads = gl.load_from_path(cell_lines_path, parser=tads_parser)

# take the IMR90 TADs
imr90 = tads[tads['cell'] == 'IMR90']
gm12878 = tads[tads['cell'] == 'GM12878']

# compute the pairs
pairs = genes_coords.join(experiment=genes_coords,
                    genometric_predicate=[gl.DLE(500000)],
                    output="CONTIG")

# do the mapping between the pairs and the TADs locations
mapping_temp = pairs.map(experiment=imr90,refName="", expName="imr90")
mapping = mapping_temp.map(experiment=gm12878, refName="",
    expName="gm12878")

# computing the same and cross set
same = mapping.reg_select(
       (mapping.RegField("left.gene") < mapping.RegField("right.gene")) &
       (mapping.RegField("count_left_gm12878") == 0) &
       (mapping.RegField("count_left_imr90") == 0)).materialize()
cross = mapping.reg_select(
       (mapping.RegField("left.gene") < mapping.RegField("right.gene")) &
       (mapping.RegField("count_left_gm12878") > 1) &
       (mapping.RegField("count_left_imr90") > 1)).materialize()
```

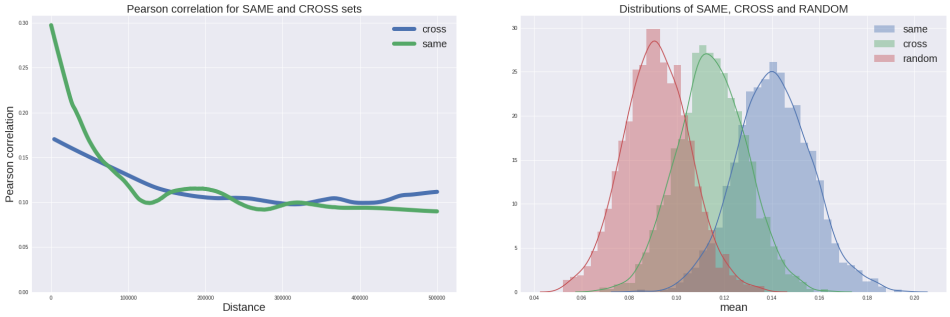
Using Dixon TADs

Using the TADs from [11] and the tissues that we previously explained we obtained the results displayed in figure 6.7, 6.8 and 6.9

In the various experiments we can easily see that at near distances the

(a) (b)

Figure 6.7: Comparison of gene correlation in the same and cross sets in the brain tissue, using TADs from [11]



(a) (b)

Figure 6.8: Comparison of gene correlation in the same and cross sets in the ensemble of brain, breast and liver tissues, using TADs from [11]

(a) (b)

Figure 6.9: Comparison of gene correlation in the same and cross sets in the muscle tissue, using TADs from [11]

correlation of genes that are in the *same* TAD is much higher than the one of genes a*cross* TADs. Also the distributions of *same* and *cross* differ

substantially.

Using Rao TADs

Using the TADs from [26] we obtained the results in figure 6.10, 6.11 and 6.12.

Also in this case the results are even more prominent.

We can conclude the following: *TADs creates insulated regions in which inner genes show high correlation in expression between themselves and low correlation with genes on different TADs.*

6.2.5 Correlations in tumor and normal tissues

Following the logic of the previous experiment we wanted to see the difference between correlations in normal tissues and in cancerous tissues. We basically plotted the Pearson correlation of the *same* and *cross* genes in a given tissue, both in the healthy and in the tumor one.

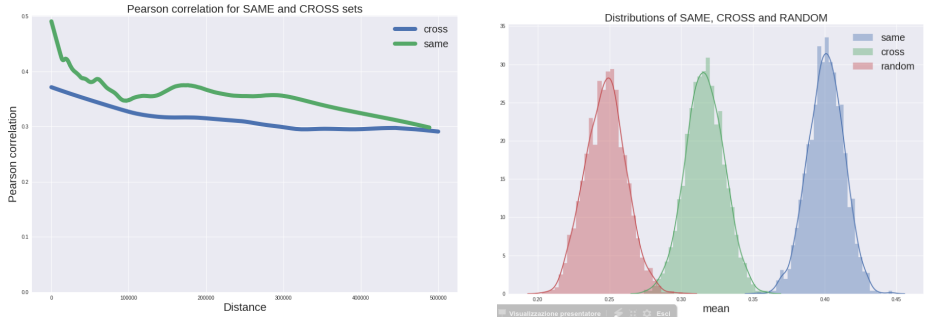
We used expression data from TCGA, which contains data both from tissues and tumors. In particular we considered the tissues/tumors in table 6.2.

The pipeline is very similar to the one described before and can be summarized as follows:

1. Consider tissue t from the $TCGA$ dataset
2. Extract the datasets $N_t = TCGA\,[type == NORMAL \wedge tissue == t]$ and $T_t = TCGA\,[type == TUMOR \wedge tissue == t]$
3. Compute the *same* and *cross* sets $SAME_{N_t}$ and $CROSS_{N_t}$ for the normal tissue
4. Compute the *same* and *cross* sets $SAME_{T_t}$ and $CROSS_{T_t}$ for the tumor
5. Plot the Pearson correlation for genes in $SAME_{N_t}$, $CROSS_{N_t}$, $SAME_{T_t}$ and $CROSS_{T_t}$

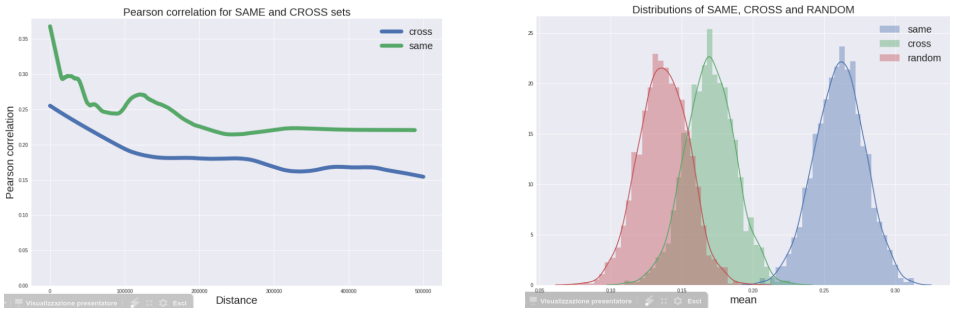
Our objective is to see if there is a substantial change in correlation in the two pairs of *same* and *cross* sets when a tissue is affected by cancer. In figure 6.13 we can see some results.

We can see that systematically the cancerous tissues give higher correlation scores both in the *same* and *cross* sets and that the relative relationship between the two sets in conserved. We can say that *gene correlation always*



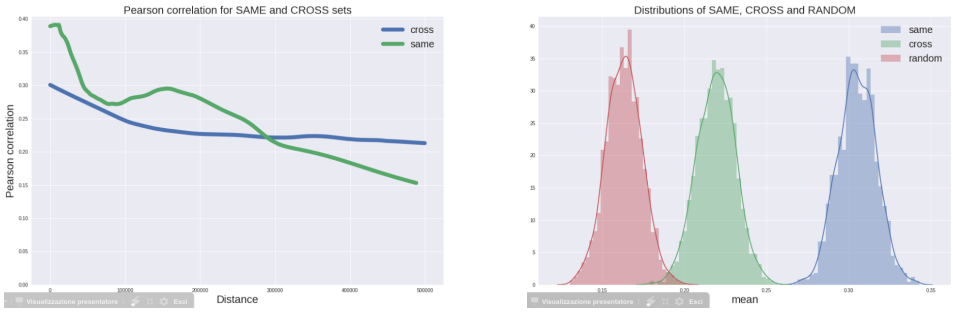
(a) (b)

Figure 6.10: Comparison of gene correlation in the same and cross sets in the brain tissue, using TADs from [26]



(a) (b)

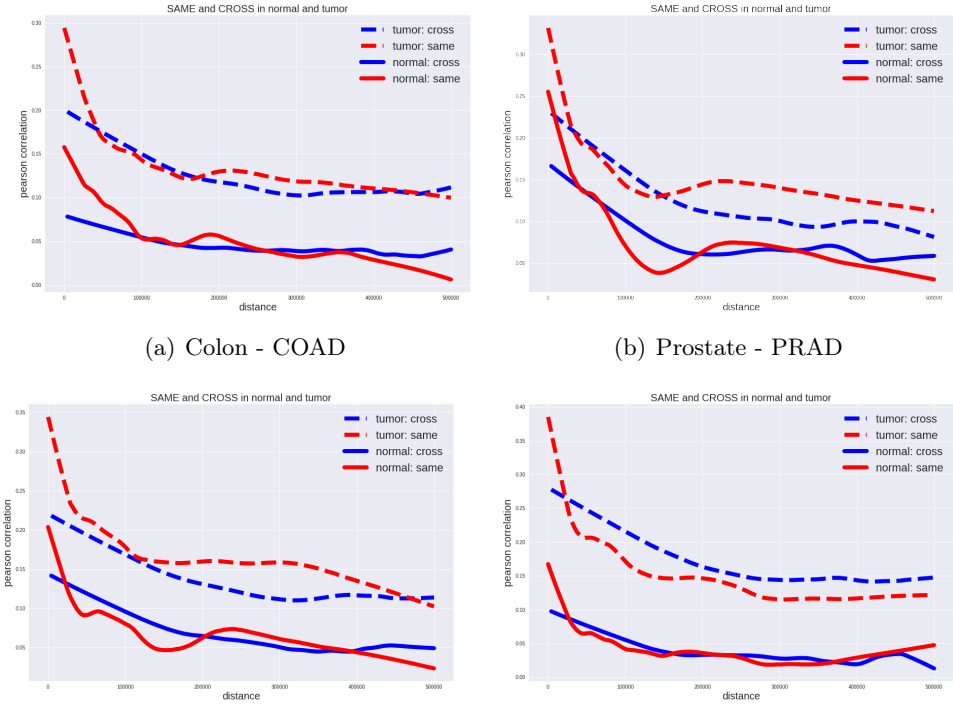
Figure 6.11: Comparison of gene correlation in the same and cross sets in the ensemble of brain, breast and liver tissues, using TADs from [26]



(a) (b)

Figure 6.12: Comparison of gene correlation in the same and cross sets in the muscle tissue, using TADs from [26]

increases when we go from a normal to a cancerous tissue. In table 6.2 we can find, for each tissue/tumor type the difference in correlation between the



(a) Colon - COAD

(b) Prostate - PRAD

(c) Head/Neck - HNSC

(d) Kidney - KICH

Figure 6.13: Some example in four tissue/tumor comparison. Red dashed line: tumor same. Blu dashed line: tumor cross. Red continuous line: normal same. Blu continuous line: normal cross

normal and the tumor type.

The coefficient in table 6.2 can be computed as follows:

$$\Delta_{TN_t} = \frac{1}{|P|} \sum_{i=1}^{|G|} \sum_{j=1 \wedge i>j}^{|G|} \text{pearson}\left(\mathbf{x}_T^{g_i}, \mathbf{x}_T^{g_j}\right) - \text{pearson}\left(\mathbf{x}_N^{g_i}, \mathbf{x}_N^{g_j}\right)$$

where

- G is the set of genes
- P is the set of all the gene combinations
- $\mathbf{x}_T^{g_i}$ is the tumor expression vector for gene g_i
- $\mathbf{x}_N^{g_i}$ is the normal expression vector for gene g_i

Tumor	Tissue	$\Delta_{TN_t} = \text{pearson}(T_t) - \text{pearson}(N_t)$
THCA	Thyroid	0,01306
PRAD	Prostate	0,05235
BLCA	Bladder	0,05460
KIRP	Renal clear cell	0,06080
LIHC	Liver	0,06297
BRCA	Breast	0,06389
UCEC	Uterus	0,06496
KIRC	Renal papillary cell	0,06506
HNSC	Head/Neck	0,08005
LUAD	Lung	0,09389
COAD	Colon	0,09438
LUSC	Lung squamous	0,09491
KICH	Kidney	0,12402

Table 6.2: For each tumor type the difference of correlation between the normal and tumor tissue.

6.2.6 TADs conservation across species

An other question that we want to address is *how TADs conserve across the species?* If this is true then topological domains can be thought as a general genomic structure that regulates life at an higher level than the single species.

In order to address this question we model the problem as follows:

1. Consider a set of TADs T_h^c belonging to human for a cell line c and an other set of TADs T_m^c belonging to mouse on the same cell line
2. Perform the lift-over[7] of T_m^c to the assembly of T_h^c obtaining $\widetilde{T}_m^c$
3. Take a gene set G lifted over to the assembly of T_h^c and map each gene to the TAD it belongs both in T_h^c and T_m^c
4. A gene pair (g_1, g_2) is co-occurent in both $t_h \in T_h^c$ and $t_m \in T_m^c$ if g_1, g_2 are both mapped to t_h and to t_m
5. We count how many co-occurent gene pairs there with respect to the total number of gene pairs.

In our experiment the cell line c is embryonic stem cell, the human assembly is HG19 and the mouse assembly is MM8.

[7]The *lift-over* is a procedure to translate a set of coordinates from a specified genome assembly to an other

We found the following:

- There are 178324 possible pairs of genes in the human TADs
- Among them, 105105 pairs are confirmed also in the mouse TADs
- *Therefore there is a 60% overlapping in the distribution of genes in the TADs between the human and the mouse*

Follows the python pipeline for this experiment. It shows the interaction between a GMQL query and python specific operations.

```
import gmql as gl
import itertools
import numpy as np

mouse_TADs_path = "./cell_lines_mouse_hg19/"
human_TADs_path = "./cell_lines_human_hg19/"
coords_path = "../Data/GMQLFormat/TSS_coordinates/"

tads_parser = gl.parsers.BedParser(parser_name="tads_parser",
                           chrPos=0, startPos=1, stopPos=2,
                           strandPos=3, delimiter="\t")

mouse_TADs = gl.load_from_path(local_path=mouse_TADs_path,
    parser=tads_parser)
human_TADs = gl.load_from_path(local_path=human_TADs_path,
    parser=tads_parser)

coords_parser = gl.parsers.BedParser(parser_name="coords_parser",
                            chrPos=0, startPos=2, stopPos=3,
                            strandPos=1, otherPos=[(4, "gene", "string")],
                            delimiter="\t")

gene_coords = gl.load_from_path(local_path=coords_path,
    parser=coords_parser)

cell_line = "es"
mouse_TADs = mouse_TADs.meta_select(mouse_TADs["cell"] == cell_line)
human_TADs = human_TADs.meta_select(human_TADs["cell"] == cell_line)

mapping_mouse_genes = mouse_TADs.map(experiment=gene_coords,
                     new_reg_fields={
                       "genes": gl.BAG("gene")
                     }).materialize()
mapping_human_genes = human_TADs.map(experiment=gene_coords,
```

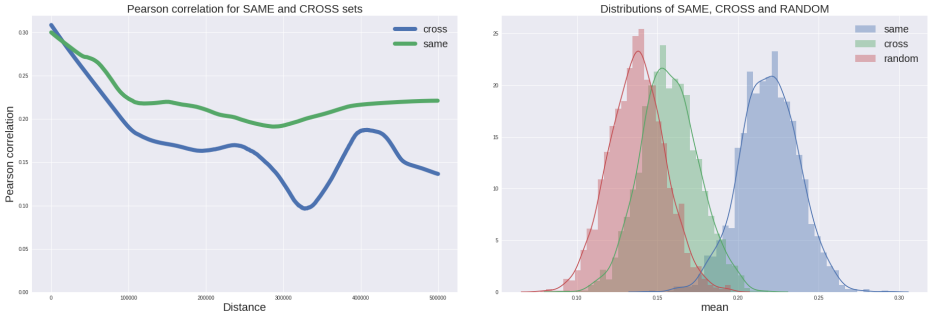
```
                    new_reg_fields={
                      "genes": gl.BAG("gene")
                    }).materialize()

def to_couples(genes_str):
   genes_list = sorted(genes_str.split(" "))
   pairs = list(itertools.combinations(genes_list, 2))
   return pairs

mapping_human_genes_couples = mapping_human_genes.regs.copy()
mapping_human_genes_couples['genes_couples'] =
    mapping_human_genes_couples.genes.apply(to_couples)
all_human_pairs = mapping_human_genes_couples.genes_couples.tolist()
all_human_pairs = [p for x in all_human_pairs for p in x]
n_human_pairs = len(all_human_pairs)

mapping_mouse_genes_couples = mapping_mouse_genes.regs.copy()
mapping_mouse_genes_couples['genes_couples'] =
    mapping_mouse_genes_couples.genes.apply(to_couples)
all_mouse_pairs = mapping_mouse_genes_couples.genes_couples.tolist()
all_mouse_pairs = [p for x in all_mouse_pairs for p in x]
n_mouse_pairs = len(all_mouse_pairs)

intersection = set(all_human_pairs).intersection(set(all_mouse_pairs))
n_intersection = len(intersection)
```

We also wanted to see if the gene expression correlation between the *same* and *cross* sets respect the same relationship as the ones shown in figure 6.7, 6.8 etc... when we use the mouse TADs instead of the human ones. Basically we map the genes to the lifted over mouse TADs and perform the same analysis that is described in section 6.2.3. In figure 6.14 we can see the results in the case of the ensemble of brain, breast and liver tissues.

We can see that *the gene correlation in the same and cross sets is conserved when using the mouse TADs and the human expression.* This indicates that *TADs are an higher level genomic structure that regulates gene expression with similarities between different species.*

6.2.7 TADs clustering

The last part of the analysis of Topological Domains properties is dedicated to their clustering based on some features that we associate to them. In particular we will show the results of TAD clustering based on gene expression and on ChIA-PET links.



(a) (b)

Figure 6.14: Comparison of gene correlation in the same and cross sets in the ensemble of brain, breast and liver tissues, using the mouse TADs

The aim of this research is to show that topological domains can communicate or form higher order structures that can span different chromosomes. Since it has been proven that TADs are conserved across different species but chromosomes are not, we can say that topological domains and their interaction constitute a more stable biological structure.

The application of clustering algorithms to biology is very common and a lot of different techniques have been tried [15], [17]. Usually gene expression is the main feature that biologists try to cluster in order to find relevant functional areas or mutations. We will use gene expression to find set of TADs that may share common characteristics and this will require to associate to each of them an expression vector. On the other side we will use a completely different set of data called ChIA-PET, to find communities of TADs that share different connections.

ChIA-PET stands for *Chromatin Interaction Analysis by Paired-End Tag Sequencing* and combines Chromatin immunoprecipitation (ChIP)[8], Chromosome conformation capture (3C) and next generation sequencing technologies for determining long range genome interactions. From our point of view, a ChIA-PET dataset can be seen as mapping between regions in the genome and can be seen as two GMQL dataset with a join column like in figure 6.15.

In our experiments we used ChIA-PET data of different cell lines and targeted to different proteins. In particular we used the following datasets:

In figure 6.16(a) and 6.16(b) we can see a visual representation of the

[8]ChIP is an experimental technique used to reveal interactions between proteins and the DNA of a cell. It determines if a specific protein is associated to genomic regions like transcription factor binding sites or others.

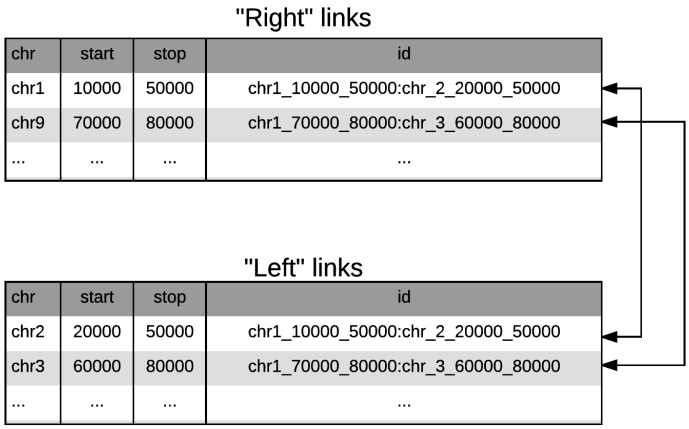


Figure 6.15: Example of a ChIA-PET dataset as two GMQL dataset with a common id column

Cell line	**Protein**
GM12878	RAD21
HCT116	POLR2A
HESC	SMC1
K562	CTCF
K562	RAD21
MCF7	CTCF
NB4	POLR2A

ChIA-PET of two cell lines.

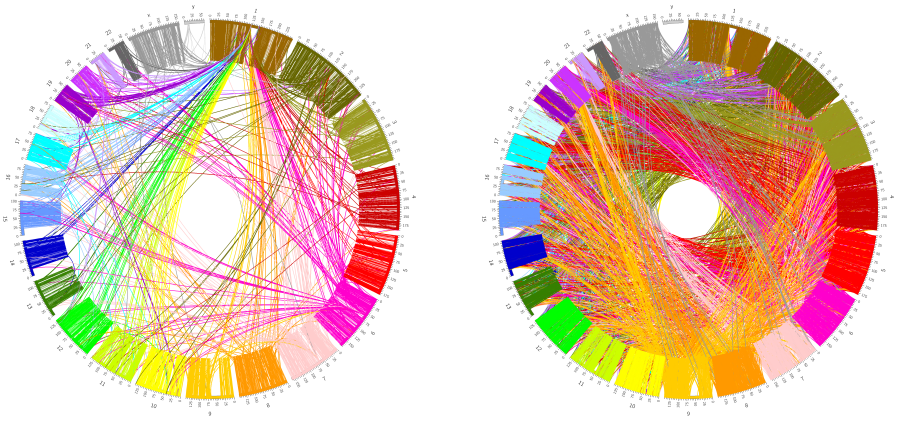
Clustering using gene expression data

Being G the set of genes and P the set of patients (features) in the RNA-seq dataset, the expression matrix $\mathbf{E}$ is a $|G| \times |P|$ matrix. Each cell $\mathbf{E}_{gp}$ of the matrix is the read count for gene g and patient p.

The considered dataset is constituted already of RPKM normalized data as follows:

$$RPKM\left(\mathbf{E}_{gp}\right) = \frac{10^9 \cdot \mathbf{E}_{gp}}{\sum_{g'=1}^{G} \mathbf{E}_{g'p} \cdot L_g} \tag{6.1}$$

where L_g is the length of gene g and $\sum_{g'=1}^{G} \mathbf{E}_{g'p}$ is the total number of counts for patient p.



(a) ChIA-PET connections in HESC cell (b) ChIA-PET connections in MCF7 cell

We rescale these values using a log10 transformation as follows:

$$\widetilde{\mathbf{E}}_{gp} = \log_{10}\left(RPKM\left(\mathbf{E}_{gp}\right)\right) \tag{6.2}$$

Given a set of TAD T we map each gene $g \in G$ to the TAD it belongs to. For each TAD t, we take all the expression vectors $\widetilde{\mathbf{e}}_g$ of the genes mapped to it and we do the punctual mean for each feature. We obtain a new expression matrix $\mathbf{X}$, having a row for each TAD:

$$\mathbf{X}_{tp} = \frac{1}{|G_t|} \sum_{g \in G_t} \widetilde{\mathbf{E}}_{gp} \tag{6.3}$$

where G_t is the set of genes mapped to TAD t.

We filter out the TADs that have the mean expression $\frac{1}{|P|}\sum_{p \in P} \widetilde{\mathbf{X}}_{tp} = 0$

Before doing clustering we normalize each row of $\mathbf{X}$ subtracting it by its mean and dividing by its standard deviation. This is also called z-score:

$$\widetilde{\mathbf{x}}_t = \frac{\mathbf{x}_t - \overline{\mathbf{x}_t}}{\mathrm{Var}[\mathbf{x}_t]} \tag{6.4}$$

We use **hierarchical clustering** using the *ward* method and *euclidean* distance. For cutting the dendrogram we use a visual method, therefore the cut is done manually. In figure 6.16 we can see an example of dendrogram with a cutting at distance 350.

Each TAD is associated to a cluster $c \in C$. Using a distance of 350 we obtain 25 clusters.

In order to assess the quality of the clustering, we need to measure how much the TADs in each cluster are correlated at the level of expression.

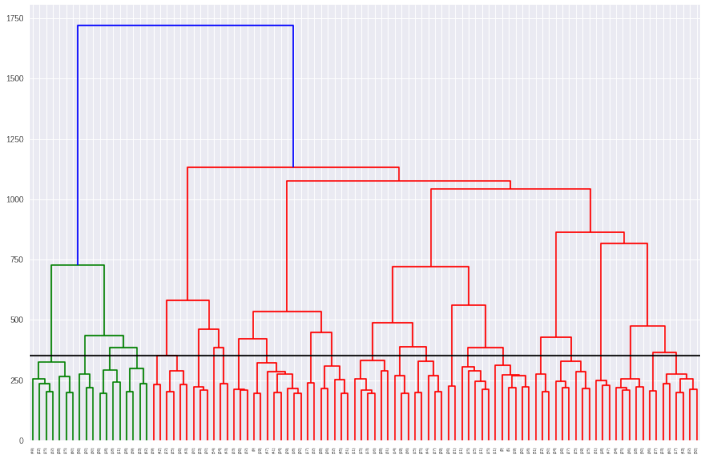


Figure 6.16: Visualization of the dendrogram of the hierarchical clustering with a cutting point at distance 350

Therefore we build a correlation matrix where rows and columns are ordered by cluster number and each cell $\mathbf{R}_{ij}$ is the pearson correlation of the expression vectors $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$:

$$\text{pearson}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\sum_{k=1}^{|P|} (x_{ik} - \overline{\mathbf{x}_i})(x_{jk} - \overline{\mathbf{x}_j})}{\sqrt{\sum_{k=1}^{|P|} (x_{ik} - \overline{\mathbf{x}_i})^2} \sqrt{\sum_{k=1}^{|P|} (x_{jk} - \overline{\mathbf{x}_j})^2}} \tag{6.5}$$

We summarize the clusters in matrix $\mathbf{R}$ by constructing a new matrix $\mathbf{C}$ such that:

$$\begin{cases} \mathbf{C}_{ij} = \frac{1}{|C_i| \times |C_j|} \sum_{k \in C_i} \sum_{h \in C_j} \mathbf{R}_{kh} & \text{if } i \neq j \\ \mathbf{C}_{ii} = \frac{1}{|A_i|} \sum_{k \in C_i} \sum_{h \in C_i, h > k} \mathbf{R}_{kh} \end{cases} \tag{6.6}$$

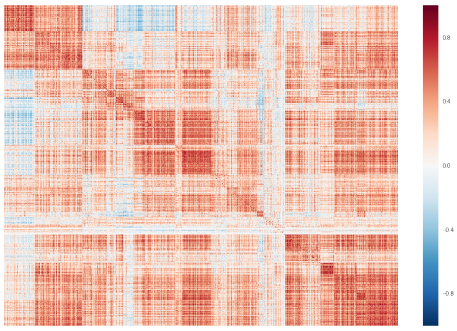
where A_i is the number of combination of TADs in cluster i and C_i, C_j are the set of tads in cluster i and j. Therefore we obtain a $|C| \times |C|$ matrix that correlate each cluster to any other. In figure 6.17(a) and 6.17(b) we can see both the correlation matrix and matrix $\mathbf{C}$ for the case with cutting at 350 distance.

To evaluate the clustering performances we compute the following indicator on matrix $\mathbf{C}$.

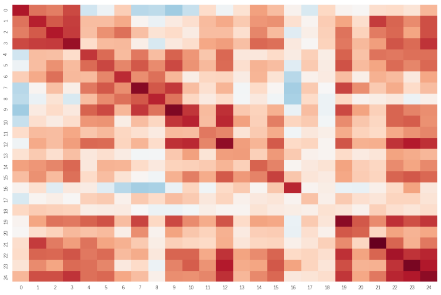
$$\text{diagonality}(\mathbf{C}) = \frac{\sum_{i \in C} \mathbf{C}_{ii}}{\sum_{i \in C} \sum_{j \in C, j > i} \mathbf{C}_{ij}} \tag{6.7}$$

For the considered case we have a diagonality index of **0.205**.

In table 6.3 we plot some quantitative information about the found clusters.



(a) **R** correlation matrix. TAD vs TAD



(b) The matrix **C**

Table 6.3: Cluster statistics for the gene expression clustering

cluster	#TADs	#chrom	#contiguous	#connections	#inter	#intra
1	221	22	11	29	25	4
2	106	21	1	2	2	0
3	73	19	3	33	10	23
4	152	22	5	9	8	1
5	71	20	0	1	1	0
6	106	22	1	7	7	0
7	63	21	0	0	0	0
8	54	16	5	10	10	0
9	67	17	3	32	26	6
10	81	21	1	4	4	0
11	217	22	16	70	66	4
12	49	15	1	5	5	0
13	202	21	8	16	16	0
14	114	22	5	16	12	4
15	69	19	1	2	2	0
16	140	21	8	16	15	1
17	47	16	0	2	2	0
18	67	20	0	11	11	0
19	85	18	1	2	2	0
20	123	22	7	34	31	3
21	126	21	8	17	17	0
22	96	19	1	2	1	1
23	163	22	11	71	69	2
24	63	17	2	2	2	0

25	225	21	9	25	25	0

We can see that the found cluster are inter-chromosomal (the TADs inside a cluster are likely to belong to different chromosomes). This should indicate that TADs structures "communicate" in some way across different chromosomes and this affects also the gene expression of them.

Clustering using ChiA-PET links

With gene expression clustering we tried to cluster TADs on the base of an *indirect* property that they have, the expression of the genes that happen to be on them. Now we try a completely different approach, which tries to find *communities* of TADs on the basis of physical ChIA-PET connection between them.

Consider the graph $G \subset V \times V$ with V the set of ChiA-PET vertices. We map each ChiA-PET vertex to the TAD it belongs. If the vertex does not intersect any TAD it is not considered.

If the set of TADs is T we build a *connection matrix* $\mathbf{Y}$ having dimensions $|T| \times |T|$. Each cell $\mathbf{Y}_{ij}$ represents the number of ChiA-PET connections that are present between TAD i and TAD j.

The clustering method that we use is borrowed from the network analysis field and it is called *Louvain Modularity* [5]. This algorithm tries to optimize *modularity*, which is a scale value between -1 and 1 measuring the density of edges inside communities with respect to the one outside the communities:

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{6.8}$$

where

- A_{ij} is the edge value (weight) between the nodes (TADs) i, j
- k_i, k_j is the sum of weights of edges attached to i and j
- m is the sum of all the weight in the graph
- c_i, c_j are the communities of nodes i, j
- $\delta(c_i, c_j) = \begin{cases} 0 & \text{if } c_i \neq c_j \\ 1 & \text{if } c_i = c_j \end{cases}$

Of course it is infeasible to scan all the possible groups of nodes, therefore some heuristic is used to speed up the process. In this method firstly local small communities are found and then each community is considered as a single node and the first step is repeated.

Since a lot of TADs are connected with no other TADs (due to the lack of data in the ChIA-PET dataset) we are forced to discard all the communities of only one element. Therefore this clustering method *does not cluster all the TADs* but only the one that share connections between each other. We are confident that having more data about connection will increase the number of connected TADs and therefore the coverage of this method. In figure 6.17 can be found a set of aggregate statistics about the cluster found with this method.

	#TADs	#chromosomes	#contiguous_tads	%contiguous_tads	mean correlation	p_value
mean	5,732558	2,3720930233	2,1162790698	37,5766083192	0,1745559101	0,152292
median	2	1	1	50	0,1643725122	0,13
std	10,82711	3,3562249457	3,6271187949	22,3689477121	0,1057271154	0,113489
min	2	1	0	0	-0,0388816837	0
max	81	19	25	75	0,8936413375	0,764

Figure 6.17: Aggregate statistics for the found clusters using the community detection algorithm

In particular, consider a cluster $c \in C$ and the set of TADs $\{t_1, ..., t_n\}$ belonging to c:

- Mean correlation: average correlation of expression of all the genes that are mapped to the TADs c
- p-value:

$$p{-}value = \frac{\sum_{i=1}^{M} [\text{mean\_correlation}(\text{sample}_i) > \text{mean\_correlation}(c)]}{M}$$

  where mean_correlation(sample_i) computes the average correlation of expression of all the genes inside a sample of TADs of the same size of cluster c. This operation is repeated M times and the value is normalized by that number. A low p-value is index of a strong correlation inside cluster c with respect to any other possible combination of TADs.

We can see from the statistics that the majority of cluster is small (2 to 5 TADs) and we have a few large clusters, with a maximum size of 81 TADs. It is frequent that two contiguous TADs share a ChIA-PET link also due to the dataset of links that we are using.

A good part of the clusters are inter-chromosomal like in the case of gene expression clustering.

We also notice the fact that for the big clusters determined with this method we have very low p-value. This means that when a strong ChIA-PET network is established, gene regulation is affected in the same way in all the network.

Chapter 7

Conclusions

This is the end, beautiful friend
This is the end, my only friend, the end
Of our elaborate plans, the end
Of everything that stands, the end
No safety or surprise, the end
I'll never look into your eyes, again

Apocalypse Now

In this work we have developed a python framework for the Genometric Query Language (GMQL) and we have tested and it on real and complex biological problems.

GMQL is a big data engine targeted to genomics leveraging on several computational engines like Spark and Flink. It is currently developed at the Department of Electronics, Information and Bioengineering of the Politecnico of Milano. Its main users are biologists, bioinformaticians and general researchers.

An initial study of the GMQL engine and language was performed (chapter 2) analyzing the various ways that the user has for interfacing with the system, the various language operators and their semantics and also the internal implementation of the language.

Then the differences between the Scala (on which GMQL bases its implementation) and Python language have been studied in order to find the best communication strategy between the python package and the Scala GMQL back-end. Also a comparison of possible library architectures has been performed underlining the advantages and disadvantages of each approach (chapter 3).

The resulting architecture wraps the GMQL operators into Python functions to be applied to a GMQLDataset, which is the python abstraction of

a GMQL dataset (chapter 5). The library also provides a data structure for holding the results of a query, the GDataframe. This is a double-tabular structure that manages the link between the region data and the metadata. On top of this structure, a machine learning module, implemented by Anil Tuncel for his master thesis, has been developed. The library offers both local and remote computation of the queries. In the second case the package communicates with a remote server with the GMQL engine and repository installed, sends the query (in a serialized tree-structural form), waits for the results and finally downloads them in a GDataframe (chapter 4). The library was finally distributed as a public package in the international python repository.

In the second part of the project, the developed library was used in a set of biological applications (chapter 6). In particular we explored the Topologically Associating Domains, which are genomic regions within which the physical interactions occur much more frequently than out of them. We have seen that the correlation between pairs of genes that fall inside a TAD is higher than between genes in different TADs unraveling the fact that topological domains are zones of the genome having similar regulation (gene expression). We have also seen that gene pairs correlation is higher in tumoral tissues than in normal ones both in the case of same and different TADs. We wanted also to see if the topological domains conserve across different species and discovered that 60% of the pairs of genes cohabiting the same human TAD also happen to be in the same mouse TAD. In the last part of the research we wanted to cluster the human TADs based on two features. The first feature was the expression level of the genes inside the TAD and we ended up finding 20 clusters. The second feature was the ChIA-PET connections (regions of the genome that "communicate" through protein binding sites) and we built a TADs network and found a set of "communities". Both the results show that there exist interactions between TADs in the same and across different chromosomes. Work to understand the real nature of these interaction needs to be done yet.

Bibliography

[1] Apache. Apache pig. https://pig.apache.org/.

[2] Apache. Spark. https://spark.apache.org/.

[3] Apache. Spark. http://spark.apache.org/docs/2.1.0/api/python/pyspark.html.

[4] Regina C Betz, Rita M Cabral, Angela M Christiano, and Eli Sprecher. Unveiling the roots of monogenic genodermatoses: genotrichoses as a paradigm. *The Journal of investigative dermatology*, 132 3 Pt 2:906–14, 2012.

[5] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10:10008, October 2008.

[6] Python community. Pypi - the python package index. https://pypi.python.org/pypi.

[7] Python community. What is python? executive summary. https://www.python.org/doc/essays/blurb/.

[8] Barthélémy Dagenais. Py4j - a bridge between python and java. https://www.py4j.org/.

[9] S. Datta and D. Nettleton. *Statistical Analysis of Next Generation Sequencing Data*. Frontiers in Probability and the Statistical Sciences. Springer International Publishing, 2014.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[11] Jesse R Dixon, Siddarth Selvaraj, Feng Yue, Audrey Kim, Yan Li, Yin Shen, Ming Hu, Jun S Liu, and Bing Ren. Topological domains in mammalian genomes identified by analysis of chromatin interactions. *Nature*, 485(7398):376, 2012.

[12] Darya Filippova, Rob Patro, Geet Duggal, and Carl Kingsford. Identification of alternative topological domains in chromatin. *Algorithms for Molecular Biology*, 9(1):14, 2014.

[13] Johan H. Gibcus and Job Dekker. The hierarchy of the 3d genome. *Molecular Cell*, 49(5):773 – 782, 2013.

[14] National Cancer Institute. The cancer genome atlas. https://cancergenome.nih.gov/.

[15] Pablo A. Jaskowiak, Ricardo JGB Campello, and Ivan G. Costa. On the selection of appropriate distances for gene expression data clustering. *BMC Bioinformatics*, 15(2):S2, Jan 2014.

[16] A. Kaitoua, P. Pinoli, M. Bertoni, and S. Ceri. Framework for supporting genomic operations. *IEEE Transactions on Computers*, 66(3):443–457, March 2017.

[17] G. Kerr, H.J. Ruskin, M. Crane, and P. Doolan. Techniques for clustering gene expression data. *Computers in Biology and Medicine*, 38(3):283 – 293, 2008.

[18] Erez Lieberman-Aiden, Nynke L Van Berkum, Louise Williams, Maxim Imakaev, Tobias Ragoczy, Agnes Telling, Ido Amit, Bryan R Lajoie, Peter J Sabo, Michael O Dorschner, et al. Comprehensive mapping of long-range interactions reveals folding principles of the human genome. *science*, 326(5950):289–293, 2009.

[19] M. Masseroli and A. Kaitoua. Gmql architecture. https://github.com/DEIB-GECO/GMQL/blob/master/docs/gmql_architecture.md.

[20] 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.

[21] National Human Genome Research Institute (NHGRI). A brief guide to genomics. https://www.genome.gov/18016863/a-brief-guide-to-genomics/.

[22] Martin Odersky. A brief history of scala. http://www.artima.com/weblogs/viewpost.jsp?thread=163733.

[23] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, April 2014.

[24] National Institute of Health. Genotype-tissue expression. https://www.gtexportal.org/home/.

[25] 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*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[26] 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.

[27] Nguyen Thanh Tam and Insu Song. Big data visualization. *Information science and applications (ICISA)*, 2016.