Exploring Genomic Datasets: from Batch to Interactive and Back

Luca Nanni
Dip. Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Milan, Italy
luca.nanni@polimi.it

Pietro Pinoli
Dip. Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Milan, Italy
pietro.pinoli@polimi.it

Arif Canakoglu
Dip. Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Milan, Italy
arif.canakoglu@polimi.it

Stefano Ceri
Dip. Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Milan, Italy
stefano.ceri@polimi.it

ABSTRACT

Genomic data management is focused on achieving high performance over big datasets using batch, cloud-based architectures; this enables the execution of massive pipelines, but hampers the capability of exploring the solution space when it is not well-defined, by choosing different experimental samples or query extraction parameters. We present PyGMQL, a Python-based interoperability software layer that enables testing of experimental pipelines; PyGMQL solves the impedance mismatch between a batch execution environment and the agile programming style of Python, and provides transparency of access when exploration requires integrating local and remote resources. Wrapping PyGMQL and Python primitives within Jupyter notebooks guarantees reproducibility of the pipeline when used in different contexts or by different scientists. The software is freely available at https://github.com/DEIB-GECO/PyGMQL.

CCS CONCEPTS

• **Mathematics of computing → Exploratory data analysis**; *Cluster analysis*; • **Applied computing → Computational genomics**;

KEYWORDS

Scientific data management; impedance mismatch; data transparency; interactive data exploration

ACM Reference Format:
Luca Nanni, Pietro Pinoli, Arif Canakoglu, and Stefano Ceri. 2018. Exploring Genomic Datasets: from Batch to Interactive and Back. In *ExploreDB 2018 : 5th International Workshop on Exploratory Search in Databases and the Web , June 15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3214708.3214710

1 INTRODUCTION

With the growth of availability of well-curated scientific databases, data exploration is becoming a crucial aspect of scientific research; experimental results can be confirmed by queries over repositories of curated knowledge; this trend is particularly relevant in life science, as many communities are investing huge efforts in openly accessible data collections.

Among life science applications, genomic data management is the most relevant for the database community, due to the recent development of massive genome sequencing technologies, that is producing a huge amount of genomic datasets – by 2025, the global size of genomic data is expected to exceed the size of all YouTube videos by one or two orders of magnitude [16]. Worldwide international data sequencing efforts have already produced important results in terms of curated repositories; classical examples include the Encyclopedia of DNA Elements (ENCODE, [6]) the Cancer Genome Atlas (TCGA, [17]), the 1000 Genomes Project [15]. According to many biologists and clinicians, a wealth of information is undisclosed within such repositories; their discovery requires a combination of data extraction, analysis, and exploration tools.

As part of a large project in genomic data management, we developed GMQL [10] [9], a data management system for integrating heterogeneous datasets which are either produced by experimental activities or retrieved from repositories of curated data. The peculiar aspects of GMQL are the provisioning for a high-level, declarative approach to data extraction and the support of a cloud-based implementation over genomic repositories, which have the typical big data dimensions. As GMQL is implemented using cloud engines (the currently preferred engine is Apache Spark [18]), GMQL is best suited to support batch queries over large datasets, typically part of complex pipelines, with executions that can require hours of computing time. Such data-driven, query-based approach is clearly inadequate for data exploration.

This paper describes PyGMQL, our approach to data exploration that uses GMQL. Technically, PyGMQL is a relatively simple software component when compared to GMQL, as it is essentially a Python interface to GMQL. However, such interface had to solve classical data exploration problems, such as:

- **Impedance mismatch**, i.e the need of transforming data from set-oriented to record-oriented interaction: such need is stronger in an exploratory session where we need to support several of these transformations.
- **Distribution transparency**, i.e. the ability of transparently combining local and remote datasets in the operations of the query language.

ExploreDB 2018 , June 15, 2018, Houston, TX, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5847-7/18/06.
https://doi.org/10.1145/3214708.3214710

- **Reproducibility**, i.e. the ability of an entire experiment to be replicated, either by the same researcher or by someone else working independently.

In this paper, after a short description of GMQL, we describe the technical features of PyGMQL, and specifically how it addresses the above challenges. For demonstrating the effectiveness of our exploration approach, we dedicate a long section to a biological example, which however is much simplified in its biological description so that it can be understood with limited domain knowledge. We conclude by showing the limitations of current predominant approaches, which are based on low-level scripting embedded within rather static workflow languages.

2 METHODS

GMQL: batch analysis of genomic datasets

GMQL is a data management and query system for biological data-driven research. The main components of the system are: (a) a repository where we import large public datasets as well as private data; (b) a query language, which combines classical relational algebra operators with domain specific ones; (c) a Web accessible interface for browsing the repository, composing and launching queries and retrieving the results; and (d) an implementation based on cloud computing technologies to cope with the big data nature of the biological research.

The data model used by GMQL is a significant improvement with respect to its competitors. GMQL organizes data within datasets, each of which consists of a collection of samples. A sample consists of two components: experimental observations (e.g., list of mutations, expression of genes) and metadata, providing clinical and macroscopic features of the patient/sample as well as information on how the experiment has been performed.

GMQL is an algebraic language whose operations apply to either one or two datasets and produce a result dataset; a GMQL query is invoked by requiring the materialization of its result, which in turn causes the recursive computation of all the datasets building intermediate results, up to the source datasets stored in the repository. GMQL operations include classic relational operations as well as domain-specific ones; their peculiarity is to apply both to observations and to metadata, thereby progressively aggregating the metadata that explains how the result is built from the source datasets.

The repository integrates curated datasets from several open sources (currently, 18 datasets and 138K samples) and describes heterogeneous information, including mutations, expression (activity) of each gene, protein-DNA interaction and 3D conformation of the genome.

PyGMQL: motivation

GMQL is a batch system; the scientist has to write a query, setting all the needed parameters, run it and wait for the results. Unfortunately, this approach is not suited for data exploration. In fact, in the preliminary step of a study, scientists do not clearly know what to ask to the data. In this case, the full query should be written in an interactive procedure, where the results of the current step guide the formulation of the next step. Even when the query is clear from the very beginning, many parameters may not be known *a priori* (e.g., which genomic distance to consider, which level of significance of a statistical test to accept) and have to be selected via a trial and error process.

Furthermore, the evaluation of intermediate results requires statistical and visualization methods which are much easier to be expressed in an imperative and interactive environment, using freely available libraries and resources. PyGMQL attempts to combine the power of the declarative formalism of GMQL with the many advantages of a scripting language such as Python: high quality packages for statistics, machine learning and data visualization, an interactive execution model, a huge community support, the availability of advanced development tools (in particular we enforce the use of Jupyter Notebooks[1]).

PyGMQL: architecture

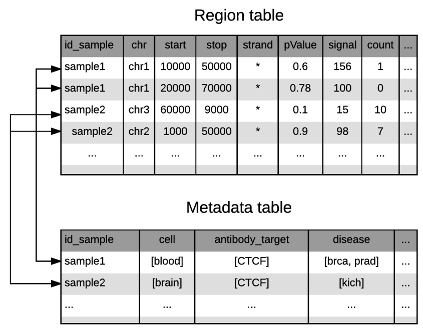
Library abstractions to solve the impedance mismatch. PyGMQL is an open source Python library designed to make the embedding of GMQL queries within scripts as natural as possible. The first issue it solves is the impedance mismatch between the imperative record-based approach of Python and the declarative set-based approach of GMQL by introducing two abstract data types that represent the *state* of a GMQL variable. Those are:

- *GMQLDataset*: represents a dataset in a GMQL query and is used to perform genomic operations in a set-based fashion, coherently with the system approach. The user can transform a dataset by using the operators of the library (which directly map to the primitives of GMQL). In fact, a GMQLDataset does not include any data, but it contains a *direct acyclic graph* (DAG) that represents the flow of GMQL operations to build the associated dataset. Every operation on a GMQLDataset returns another GMQLDataset with a modified DAG, with the exception of the `materialize` operator, which instead triggers the execution.
- *GDataframe*: represents the result of a GMQL query, as returned by a `materialize` invocation. It is made of two Pandas[2] Dataframes (equivalent to relational tables) holding the resulting regions and metadata. The two tables are synchronized by using a common column representing the sample identifier (see Figure 1). Regions of a sample correspond to distinct rows, each with the same sample identifier; instead, all the metadata of a given sample are collected in a single row, with a single sample identifier. Note that samples can have a bag (array) of values for the same metadata attribute. The GDataframe enables a record-based data manipulation.

Pandas Dataframes can be easily manipulated by the most common Python utilities for data analysis. A GDataframe can be imported back into a GMQLDataset, by means of a `to_GDB` primitive. Technically, this is done by referencing the GDataFrame as one of the inputs of the DAG associated with the GMQLDataset. Figure 2 provides a visual representation of the possible transitions between GMQLDataset and GDataframe and the related functions.

The possibility of exporting data to GDataframes and then importing them back to GMQLDatasets is one of the most important

[1]http://jupyter.org/
[2]https://pandas.pydata.org/


Region table

id_sample	chr	start	stop	strand	pValue	signal	count	...
sample1	chr1	10000	50000	*	0.6	156	1	...
sample1	chr1	20000	70000	*	0.78	100	0	...
sample2	chr3	60000	9000	*	0.1	15	10	...
sample2	chr2	1000	50000	*	0.9	98	7	...
...	...	...	...	...	...	...	...	

Metadata table

id_sample	cell	antibody_target	disease	...
sample1	[blood]	[CTCF]	[brca, prad]	...
sample2	[brain]	[CTCF]	[kich]	...
...	...	...	...	...

Figure 1: Example of a GDataframe structure

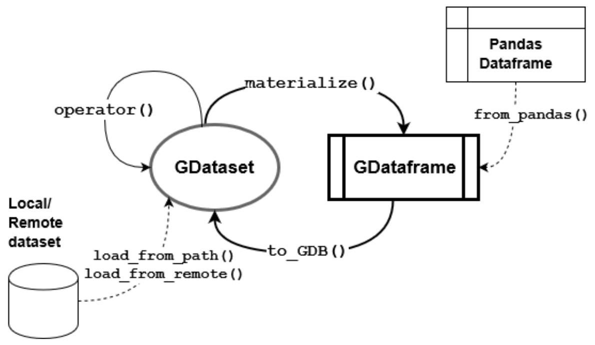


Figure 2: Main functions for the communication between GMQLDataset and GDataframe

features of PyGMQL, allowing to exploit the power and the performances of GMQL for data retrieval and a variety of Python resources for data analysis. A PyGMQL pipeline is therefore an interleaving of set- and record-oriented processing.

Lazy evaluation of queries and caching of the results. PyGMQL has been designed with the objective of adapting to the *lazy evaluation* technology of GMQL; thus, no operation is actually performed until the `materialize` operation is applied to a GMQLDataset. This strategy of execution is inspired by other big data frameworks like Apache Spark and enables a variety of optimization and parallelization strategies, as the entire DAG is considered and global decisions can be taken.

The DAG structure of PyGMQL holding the query is also used to cache the intermediate results. Since the user may want to run the same code multiple times, for example when working in an interactive programming environment like Jupyter Notebooks, this feature provides a considerable speed-up in performance. If a user decides to materialize a GMQLDataset `D_1` in order to explore it and then decides to define the new variable `D_2 = D_1.operator(...)`, a new node is added to the computation DAG, but the caching mechanism keeps the previous result in memory. In this way, when a `D_2.materialize()` operation is encountered, the computation of the nodes building `D_1` is skipped.

Local and remote execution modes. The library can operate in two different modes of execution, which can be changed at any time and alternated during the program using the function `set_mode`:

- *Local execution*: the computation of the query result is performed directly in the user machine using a local GMQL back-end.
- *Remote execution*: the computation of the query is performed by a remote GMQL engine and the results are later downloaded for follow-up Python processing. The transmission of the query information is done by passing to the remote engine a serialization of the query DAG.

In addition, PyGMQL can operate on both local and remote datasets: in the first case, data is stored in the user machine while in the second it resides in a remote GMQL repository. When the materialization operation is requested on a variable, the library automatically manages the synchronization between the sources and decides where the actual query execution must be performed. This depends on the mode in which the library is set, leading to four different scenarios, which are described in Table 1.

Note that a query can apply to many local and remote datasets. The library keeps track of the origin of each dataset and when the execution is triggered, on the basis of the data location and mode of the library, the source nodes of the DAG are renamed to fit the actual location of the data that the engine will use.

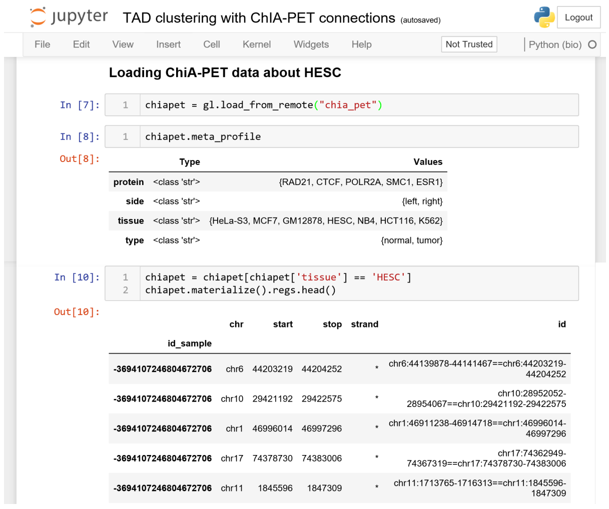
	local mode	**remote mode**
local dataset	Load the dataset from the local storage	Upload the dataset to the remote repository
remote dataset	Download the dataset from the remote repository	Define a local pointer to the remote dataset

Table 1: Possible scenarios of data synchronization depending on the location of the dataset and the mode of the library

Integration in Jupyter Notebooks. PyGMQL is optimized for being used inside a Jupyter Notebook; this setting ideally supports users in alternating data processing, data presentation and visualization steps, in the context of a well-defined interaction. Figure 3 shows a Notebook at work in two steps of the use case described below, the first one for exploring an initial dataset (in particular, the profile includes metadata attributes and values), the second one for building and then exploring the regions of a result dataset, imported as a GDataframe.

3 CLUSTERING OF DNA REGIONS

To illustrate both the expressive power and the flexibility of PyGMQL, we propose an example of biological data analysis; in the example we integrate several types of datasets, having different sizes and formats. PyGMQL is used in conjunction with standard Python routines and external libraries, showing the high interoperability of the library with other well known interactive tools. Since the focus of this work is not on the biological implications of the described analysis, we will present a high-level description of the problem which is incomplete from a biological perspective; it assumes little biological background of the reader.



Loading ChiA-PET data about HESC

```
In [7]:  chiapet = gl.load_from_remote("chia_pet")

In [8]:  chiapet.meta_profile
```

Out[8]:

	Type	Values
protein	<class 'str'>	{RAD21, CTCF, POLR2A, SMC1, ESR1}
side	<class 'str'>	{left, right}
tissue	<class 'str'>	{HeLa-S3, MCF7, GM12878, HESC, NB4, HCT116, K562}
type	<class 'str'>	{normal, tumor}

```
In [10]:  chiapet = chiapet[chiapet['tissue'] == 'HESC']
           chiapet.materialize().regs.head()
```

Out[10]:

	chr	start	stop	strand	id
id_sample					
-3694107246804672706	chr6	44203219	44204252	*	chr6:44139878-44141467==chr6:44203219-44204252
-3694107246804672706	chr10	29421192	29422575	*	chr10:28952052-28954067==chr10:29421192-29422575
-3694107246804672706	chr1	46996014	46997296	*	chr1:46911238-46914718==chr1:46996014-46997296
-3694107246804672706	chr17	74378730	74383006	*	chr17:74362949-74367319==chr17:74378730-74383006
-3694107246804672706	chr11	1845596	1847309	*	chr11:1713765-1716313==chr11:1845596-1847309

Figure 3: Example of usage of PyGMQL in a Jupyter notebook environment

Problem setting

The genome can be partitioned into regions called topologically associating domains (TADs); two consecutive TADs on the genome are separated by a genomic region, called *junction*. DNA elements (primarily genes, but also other biological actors) within a TAD are known to physically interact with each other more intensively than with DNA elements in different TADs [5]. Many groups in biological research centers are engaged in understanding how the subdivision of the genome into highly connected portions may affect gene expression and regulation [14]. Along this direction, we are supporting biologists in studying the physical interactions *between* connected TADs, which hint to higher-level regulatory mechanisms.

Connections between regions of the genome can be revealed by several experimental techniques; among them, we consider *Chromatin Interaction Analysis by Paired-End Tag Sequencing* (ChIA-PET) - a technique used to determine zones in the genome to which a specific protein binds [7]. Our pipeline uses such ChIA-PET pairs as edges for building a connection map between the topological domains in the genome.

Simply stated, the objective of our exploratory activity is the search of clusters of nodes in a network, where nodes are TADs and edges are ChIA-PET connections (see Fig. 4), where we explore alternatives in the choice of nodes and edges and in the definition of a TAD junction window (see Fig. 5). The pipeline is divided in two main sections:

- *Cluster extraction (local)*: an adjacency matrix between TAD is created on the basis of ChIA-PET connections. TADs are stored in a small-size dataset locally available, ChIA-PETs are stored in a mid-size remote dataset. The exploratory session uses the local GMQL system, by selectively loading relevant samples from the ChIA-PET dataset from the remote repository and using them together with the local TAD dataset.

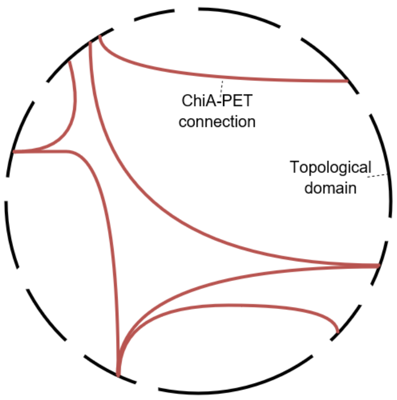


Figure 4: Schematic representation of the genome with nodes representing TADs and edges representing ChiA-PET connections

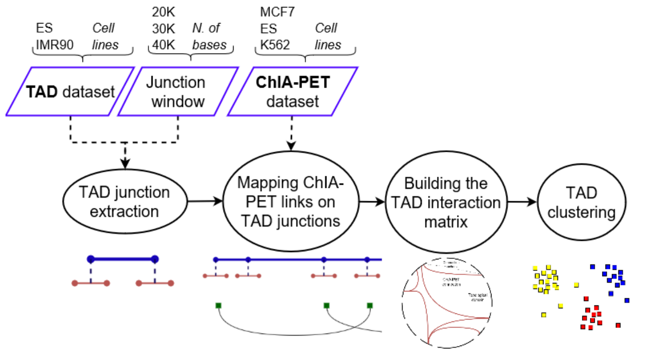


Figure 5: Cluster extraction pipeline

- *Cluster analysis (remote)*: once the most interesting clusters are selected, several different features of TADs forming a cluster and of DNA elements within each TAD are studied, by loading the cluster to the remote GMQL system, and using other datasets which are remotely stored in the repository.

TAD dataset selection and manipulation

The first parameter of the pipeline that must be chosen is the cell type (attribute `cell`) of considered TADs. The *meta-selection* function retrieves the samples which have a particular value for the selected attribute. We decide to select the `es` cell line.

Next, we must adapt a TAD dataset to the needs of our pipeline. In particular, each TAD has the simple schema $< chr, start, stop >$ describing the chromosome and the coordinates of its start and stop position on the genome, but it has no identifier. As the unique identification of TADS is required, we provide it by first extracting a TAD sample, then using the `numpy` python library to assign a numeric identifier to each TAD region, and finally storing it back as a dataset, to be used in following queries. Note the use of `materialize` followed by the use of `to_GDB` functions, and that each resulting TAD will have a new schema $< chr, start, stop, tad\_id >$.

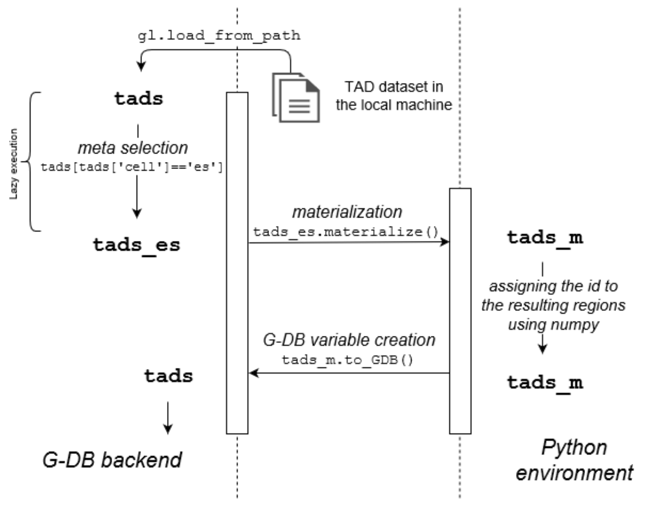


Figure 6: Schematic representation of the data exchange between the query engine and the Python environment

```
import gmql as gl
import numpy as np

gl.set_mode("local")

tads = gl.load_from_path(local_path="./path/to/tads")
tads_es = tads[tads['cell'] == 'es']
tads_m = tads_es.materialize()
tads_m.regs['tad_id'] = np.arange(len(tads_m.regs))
tads = tads_m.to_GDB()
```

A schematic representation of the sequence of operations can be found in figure 6, where the interaction between the query engine and the Python environment is highlighted.

Extraction of the TAD junctions

We next build the junctions of each TAD, which represent its endpoints. They depend on the *junction window* parameter, representing the size of junctions; these are experimentally ill-defined, hence windows may range between few thousands and forty thousand bases. The `junction_window` is a parameter of the pipeline and needs to be tuned by the scientist. Junction are defined for both the `start` and `stop` positions by using the *region projection* operation (`reg_project` operator of GMQL), expressed by a PyGMQL function (we show the case for `start`):

```
junction_window = 20000

junctions = tads.reg_project(new_field_dict={
    'start': tads.start - junction_window,
    'stop' : tads.start + junction_window})
```

Creating the TAD interaction map

The next step concerns the definition of the edges of the interaction graph between TADs. At this stage, we must select the ChIA-PET dataset; like in the TAD case, the researcher should use the `cell` metadata to select a compatible cell line. Given that we use the `es` cell line for TADs, we select the `es` cell line also for ChIA-PET; note that in many cases identical cell lines are not available and the biologist must select compatible cell lines.

```
chiapet = gl.load_from_remote("chia_pet")
chiapet = chiapet[chiapet['cell'] == 'es']
```

In summary, at this stage a pipeline depends on the selection of three parameters: the Window size and the cell lines for the TAD and ChIA-PET samples.

Next, a domain-specific operation of GMQL, embedded by a PyGMQL function, builds the TAD interaction map as a network of nodes and edges; explaining this code goes beyond the purposes of this paper, although the code is very compact and effectively combines PyGMQL and Pandas primitives. Eventually, the final data structure is an adjacency matrix $\mathcal{M} \in \mathbb{N}^{n_t \times n_t}$ where n_t is the total number of topological domains.

Clustering

The next step of the analysis consists in uncovering clusters of TADs which are as numerous as possible and are also strongly connected. This is easily achieved using the Louvain Modularity community detection algorithm [2]. In Table 2 we report the highest cardinality clusters that were found.

cluster id	#TADs	#diff. chr	%contig. TADs
2	30	13	30.0%
1207	6	1	66.7%
364	5	1	60.0%
171	5	1	40.0%

Table 2: Top 4 clusters by number of TADs involved

The first cluster is an outlier with respect both to the size and the number of different chromosomes it spans; it represents a good candidate for follow-up cluster analysis. In general, by altering the parameters of our pipeline, we can find interesting candidates for follow-up analysis. Provisioning of new experimental samples in the TADs and ChIA-PET datasets may trigger as well new data analysis sessions.

Cluster analysis

The second part of our pipeline focuses on the analysis of a selected cluster of TADs. For example, understanding which genes are active in various biological conditions or which transcription factors bind to specific regions inside the cluster could help in the identification of its function. Since the TADs cluster spans a large section of the genome (13 chromosomes, more than 200 genes and in the order of millions bases), its analysis is computationally demanding. Moreover, the analysis can take advantage of publicly available biological datasets; many of them are available on the GMQL repository. For all these reasons, the cluster analysis must move from the local machine to the remote, cloud-based environment; this change is simply performed by setting the execution mode to `remote`.

The list of clusters, locally stored in a Pandas Dataframe `clusters`, must be loaded to the remote server. Each row of this dataset represents a TAD, identified by a `cluster-id`. We can transform a given cluster (e.g. the second one) into a GMQL variable (with schema $< chr, start, stop, cluster-id >$) in the following way:

```
cluster = clusters[clusters.cluster_id == 2]
cluster_tads = gl.from_pandas(cluster).to_GDB()
```

Since the cluster analysis depends on the biologist objective, it is neither feasible nor in our goals to continue this example along all directions, we will just briefly mention one of them; this final part requires some understanding of epigenomics.

In the following, we show the processing of a remote dataset of peaks of *transcription factors*, which represent zones of the genome that regulate the production of a particular protein. These proteins in turn regulate activation/de-activation of genes, basically defining the physiological characteristics of the studied tissue. Since the dataset stores information about multiple cell types and organisms, we select only human samples from the `es` cell. We then use the `map` function to assign to each TAD in the cluster the set of transcription factors that lie inside of it (variable `tad_tf`). Since a transcription factor can be present multiple times in the same TAD, an aggregation is necessary to sum all its signal values (`gl.SUM("signal")`).

```
gl.set_mode("remote")

tf_dataset = gl.load_from_remote("tf_dataset")
tf_dataset = tf_dataset[(tf_dataset['cell'] == 'es')
        & (tf_dataset['data_type'] == 'TF')
        & (tf_dataset['cell_organism'] == 'human')]
tad_tf = cluster_tads.map(tf_dataset,
                    {'signal_sum': gl.SUM("signal")})
tad_tf = tad_tf.materialize()
```

Each region of the resulting GDataframe will have schema $< chr, start, stop, tf\_id, signal\_sum >$ and will represent how each transcription factor interacts with each TAD. Using Python is now easy to create a matrix $\mathcal{T} \in \mathbb{R}^{n_t \times n_{tf}}$ representing the aggregated signal of each transcription factor for each TAD. We can exploit this structure to derive statistics about the most relevant transcription factors in the selected cluster.

4 RELATED WORK

While GMQL has many competitors, PyGMQL is a unique first step in the direction of bridging data exploration to genomic data management. Genomic data analysis is mostly performed by using programming languages such as R and Python, the classical data scientist languages. Within the R community, Bioconductor[8] is a popular genomic toolbox; its components are published after a verification process by a wide community of users. Similarly, the Python community uses BioPython[4], an open source set of tools for computational biology and biological data manipulation. The main limitation of these approaches is the lack of scalability.

BEDTools[13], BEDOPS[11] and GROK[12] are very popular tools among bioinformaticians for genomic region manipulations. They provide operations similar to those of GMQL, but they operate at the sample level (iteration over samples must be programmed). Thus, complex queries have to be composed via command line scripts, and critical operation such as data conversion and pipeline controls have to be coded. Although quite popular, this approach is obviously neither reproducible nor highly productive from a software engineering perspective.

A higher-level software organization, that guarantees better reproducibility, is achieved by describing programmatic steps as tasks of a workflow; two relevant systems in genomic data management are Galaxy and Firecloud. Galaxy [1] is a web-based workflow platform that is used by bioinformaticians to perform batch pipelines which are quite similar to the ones described in this paper. Firecloud [3] is an open platform for secure and scalable workflow-based analysis of genomic data, hosted by the Broad Institute in Boston, a center of excellence for biomedical and genomic research. FireCloud provides a storage and billing system through GoogleCloud resources.

The FireCloud project is in the process of providing a Python interface matching with its pipelines; we are discussing with the FireCloud administrators the integration of GMQL as one of the supported tools within FireCloud and the possibility of connecting its forthcoming Python interface to PyGMQL; the goal is providing an integrated environment which seamlessly combines batch processing over the GoogleCloud and interactive data exploration.

5 CONCLUSION

PyGMQL has shown the huge potential that is offered to genomic scientists when they can bridge data extraction capabilities which scale and at the same time an agile data analysis context. The library is publicly available and can be installed through the Python Package Index with the command `pip install gmql`.

ACKNOWLEDGEMENTS

This work is supported by the ERC Advanced Grant 693174 "Data Driven Genomic Computing" (GeCo).

REFERENCES

[1] Enis Afgan et al. 2016. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. *Nucleic acids research* 44, W1 (2016), W3–W10.

[2] Vincent D. Blondel et al. [n. d.]. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 ([n. d.]), P10008.

[3] Broad Institute. 2017. FireCloud. (2017). https://software.broadinstitute.org/firecloud

[4] Peter JA Cock et al. 2009. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics* 25, 11 (2009), 1422–1423.

[5] Jesse R. Dixon et al. [n. d.]. Topological domains in mammalian genomes identified by analysis of chromatin interactions. *Nature* (apr [n. d.]), 376.

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

[7] Melissa J. Fullwood et al. [n. d.]. Chromatin Interaction Analysis Using Paired-End Tag Sequencing. *Current Protocols in Molecular Biology* 89, 1 ([n. d.]), 21.15.1–21.15.25. https://doi.org/10.1002/0471142727.mb2115s89

[8] Robert C Gentleman et al. 2004. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology* 5, 10 (2004), R80.

[9] A. Kaitoua et al. 2017. Framework for Supporting Genomic Operations. *IEEE Trans. Comput.* 66, 3 (March 2017), 443–457. https://doi.org/10.1109/TC.2016.2603980

[10] Marco Masseroli et al. 2015. GenoMetric Query Language: a novel approach to large-scale genomic data management. *Bioinformatics* 31, 12 (2015), 1881–1888. https://doi.org/10.1093/bioinformatics/btv048

[11] Shane Neph et al. 2012. BEDOPS: high-performance genomic feature operations. *Bioinformatics* 28, 14 (2012), 1919–1920.

[12] Kristian Ovaska et al. 2013. Genomic region operation kit for flexible processing of deep sequencing data. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* 10, 1 (2013), 200–206.

[13] Aaron R Quinlan. 2014. BEDTools: the Swiss-army tool for genome feature analysis. *Current protocols in bioinformatics* (2014), 11–12.

[14] Sadia Saeed et al. 2014. Epigenetic programming of monocyte-to-macrophage differentiation and trained innate immunity. 345, 6204 (2014). https://doi.org/10.1126/science.1251086

[15] Nayanah Siva. 2008. 1000 Genomes project. (2008).

[16] Zachary D. Stephens et al. 2015. Big data: astronomical or genomical? *PLoS biology* 13, 7 (2015), e1002195.

[17] John N. Weinstein et al. 2013. The cancer genome atlas pan-cancer analysis project. *Nature genetics* 45, 10 (2013), 1113.

[18] Matei Zaharia et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.