Experiences in the Development of a Data Management System for Genomics

Stefano Ceri, Arif Canakoglu, Abdulrahman Kaitoua, Marco Masseroli, and Pietro Pinoli

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, Milano, Italy
{firstname.lastname}@polimi.it

Abstract. GMQL is a high-level query language for genomics, which operates on datasets described through GDM, a unifying data model for processed data formats. They are ingredients for the integration of processed genomic datasets, i.e. of signals produced by the genome after sequencing and long data extraction pipelines. While most of the processing load of today's genomic platforms is due to data extraction pipelines, we anticipate soon a shift of attention towards processed datasets, as such data are being collected by large consortia and are becoming increasingly available.
In our view, biology and personalized medicine will increasingly rely on data extraction and analysis methods for inferring new knowledge from existing heterogeneous repositories of processed datasets, typically augmented with the results of experimental data targeting individuals or small populations. While today's big data are raw reads of the sequencing machines, tomorrow's big data will also include billions or trillions of genomic regions, each featuring specific values depending on the processing conditions.
Coherently, GMQL is a high-level, declarative language inspired by big data management, and its execution engines include classic cloud-based systems, from Pig to Flink to SciDB to Spark. In this paper, we discuss how the GMQL execution environment has been developed, by going through a major version change that marked a complete system redesign; we also discuss our experiences in comparatively evaluating the four platforms.

Keywords: Genomic Computing · Data Translation and Optimization · Cloud Computing · Next Generation Sequencing · Open Data

1 Introduction

Thanks to Next Generation Sequencing, a recent technological revolution for reading the DNA, a huge number of genomic datasets have become available [24]. Massive pipelines are used to extract processed datasets from DNA sequences, and expose heterogeneous genomic and epigenomic signals; among them, TCGA

(The Cancer Genome Atlas) [29], ENCODE (the Encyclopedia of DNA Elements) [14], and 1000 Genomes [1]. Open datasets of processed data are collected by worldwide consortia; they constitute a wealth of information, as they can be used for answering complex biological and clinical queries. While most of the processing load of today's genomic platforms is due to data extraction pipelines (e.g., [23]), we anticipate soon a shift of attention towards processed datasets, as such data are becoming increasingly available, collected by large consortia and by commercial sequencing centers. Genomics is expected to produce more data than any other scientific discipline by 2025, and is also expected to produce more data than all the social sources, including YouTube [27].

Processed datasets are used in *tertiary data analysis* for giving a global sense to heterogeneous genomic and epigenomic signals; a few systems are dedicated to tertiary data analysis, including SciDB Paradigm4 [3], and BLUEPRINT [2].

In the context of the GenData 2020[1] and GeCo Projects[2], we developed the Genomic Data Model (GDM) [21], a unifying model for processed data formats, and the GenoMetric Query Language (GMQL) [20,18], a query language for genomics. GDM is a very simple data model which mediates all existing data formats; GMQL is a high-level, declarative query language which supports data extraction as well as the most standard data-driven computations required by tertiary data analysis.

GMQL is the core of several, subsequent implementations of data management systems. In all cases, GMQL queries were translated to queries for cloud-based database engines. Version 1, described in [11], was developed between the spring 2014 and the spring 2015 and was based on Apache Pig [6] and Hadoop 1 [28]. Version 2 is described in [18]; its development started in the summer of 2015 and is still ongoing; Version 2 is based on Hadoop 2 [16] and uses Apache Spark [7]; project branches were developed for the engines Apache Flink [4] and SciDB [3]. In this paper, we discuss the development of the various GMQL versions, including the architectural choices, the supported optimizations, and the approaches to parallelism.

2 GMQL Resources: Data Model, Query Language, Integrated Repository

GMQL is based on a representation of the genomic information known as Genomic Data Model (GDM). Datasets are composed of samples, which in turn contain two kinds of data:

– Genomic region values (or simply regions), aligned w.r.t. a given reference, with specific left-right ends within a chromosome: Regions of the model describe processed data, e.g., mutations, expression or bindings; they have

[1] `http://www.bioinformatics.deib.polimi.it/gendata/`, PRIN Italian National Project, 2013-2016.

[2] Data-Driven Genomic Computing, `http://www.bioinformatics.deib.polimi.it/geco/`, ERC Advanced Grant, 2016-2021.

a schema, with 5 common attributes (id, chr, left, right, strand) including the id of the region and the region coordinates, along the aligned reference genome, and then arbitrary typed attributes. This provides interoperability across a plethora of genomic data formats;
– Metadata, storing all the knowledge about the particular sample, are arbitrary attribute-value pairs, independent from any standardization attempt; they trace the data provenance, including biological and clinical aspects.

Figure 1 shows a GDM dataset consisting of 3 samples, each associated with a patient affected by Breast Cancer (BRCA). The region part has a simple schema, describing a particular value associated to each region; regions are aligned along the whole genome and belong to specific chromosomes (not shown in the figure). The metadata part includes free attribute-value pairs, describing the patient's age and sex; in GDM attributes are freely associated to samples, without specific constraints.

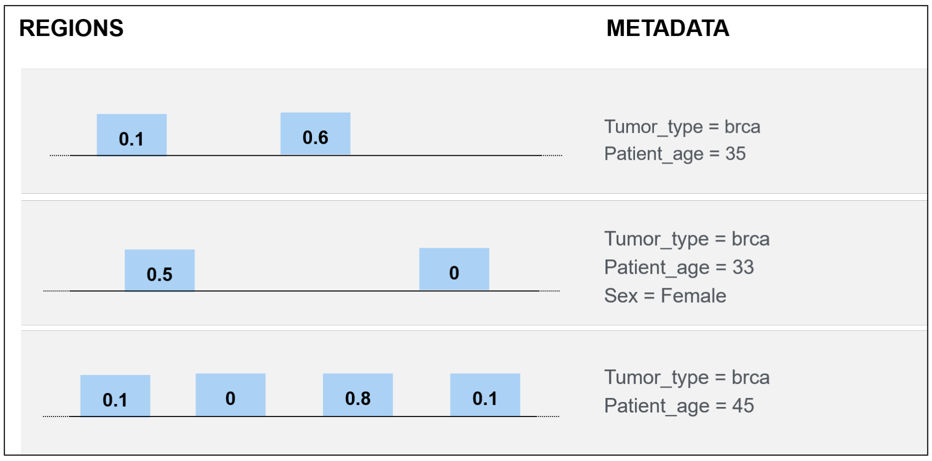


Fig. 1. Genomic Data Model

GMQL is an abbreviation for GenoMetric Query Language - the name highlights the language's ability of computing distance-related queries along the genome, seen as a sequence of positions. GMQL is a closed algebra over datasets: results are expressed as new datasets derived from their operands. Thus, GMQL operations compute both regions and metadata, connected by sample identifiers; they perform schema merging when needed. The language brings to genomic computing the classic algebraic abstractions, rooted in Ted Codd's seminal work, and adds suitable domain-specific abstractions. In [20] we show GMQL at work in many heterogeneous biological contexts.

GMQL operations include classic algebraic transformations (SELECT, PROJECT, UNION, DIFFERENCE, JOIN, SORT, AGGREGATE) and domain-specific transformations (e.g., COVER deals with replicas of a same experiment; MAP

Operation		Description
Select	Meta	Selects complete samples when a metadata predicate is true
		(predicate can refer to other datasets using a semijoin condition)
	Regions	Selects regions satisfying a region predicate
Project	Meta	Projects metadata or adds/updates metadata attributes
	Regions	Projects regions or adds/updates region attributes
Extend	Meta	Adds to metadata aggregate function results computed over regions
Order	Meta	Sorts samples and selects top ones
	Regions	Sorts regions and selects top ones
Group	Meta	Groups samples and computes aggregate values
	Regions	Groups regions and computes aggregate values
Merge	Meta	Merges all metadata of all samples
	Regions	Merges all regions of all samples
Cover		Merges all samples; regions must satify an accumulation constraint
Union		Builds the union of samples (meta and regions independently)
Difference	Meta	Metadata of results are from the first operand
	Regions	Regions of second operand are subtracted from first operand
Map	Meta	Metadata of results are from the second operand
	Regions	Aggregates values of second operand's regions intersecting a region of the first one
Join	Meta	Builds pairs of samples whose metadata satisfy join predicate
	Region	Builds pairs of regions satisfying distal and equi-join predicates

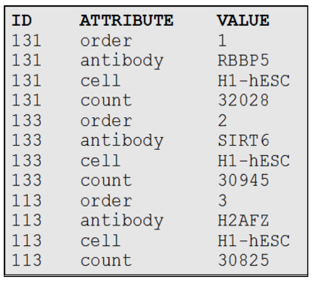
Table 1. Informal description of GMQL operations

refers genomic signals of experiments to user selected reference regions; GENOMETRIC JOIN selects region pairs based upon distance properties); their semantics is informally described in Table 1. Tracing provenance both of initial samples and of their processing through operations is a unique aspect of our approach; knowing why resulting regions were produced is quite relevant.

An example of GMQL is the following, simple query which selects the genes' promoter regions from a dataset of annotations, selects al experiments of a given datatype from the NarrowPeaks dataset of ENCODE, and maps the selected peaks to the selected promoters; the query counts how many peaks intersect with each promoter region, then counts how many peaks in each sample intersect with any promoters, then selects the top thee samples based on the latter count. The counter is a global property of each sample and therefore is included by the EXTEND operation into the metadata of results. The language supports implicit iteration over all samples.

```
PROMS = SELECT(annotationType == 'promoter') ANNOTATIONS;
PEAKS = SELECT(dataType == 'ChipSeq') ENCODE_NarrowPeaks;
RESULT = MAP(peakCount AS COUNT) PROMS PEAKS;
GLOBAL = EXTEND(count AS SUM(peakCount)) RESULT;
TOP = ORDER(count; TOP 3) GLOBAL;
```

This query was executed over 2,423 ENCODE samples including a total of 83,899,526 peaks mapped to 131,780 human promoters, producing as result 29 GB of data. Figure 2 shows three samples with the highest counters (e.g., sample 131, corresponding to antibody target `RBBP5` of cell line `H1-hESC`, has 32,028 peaks intersecting with gene promoters.) Note that metadata of the result are partially computed by the query, partially derived from the initial description of experimental conditions in the `NarrowPeaks` dataset.

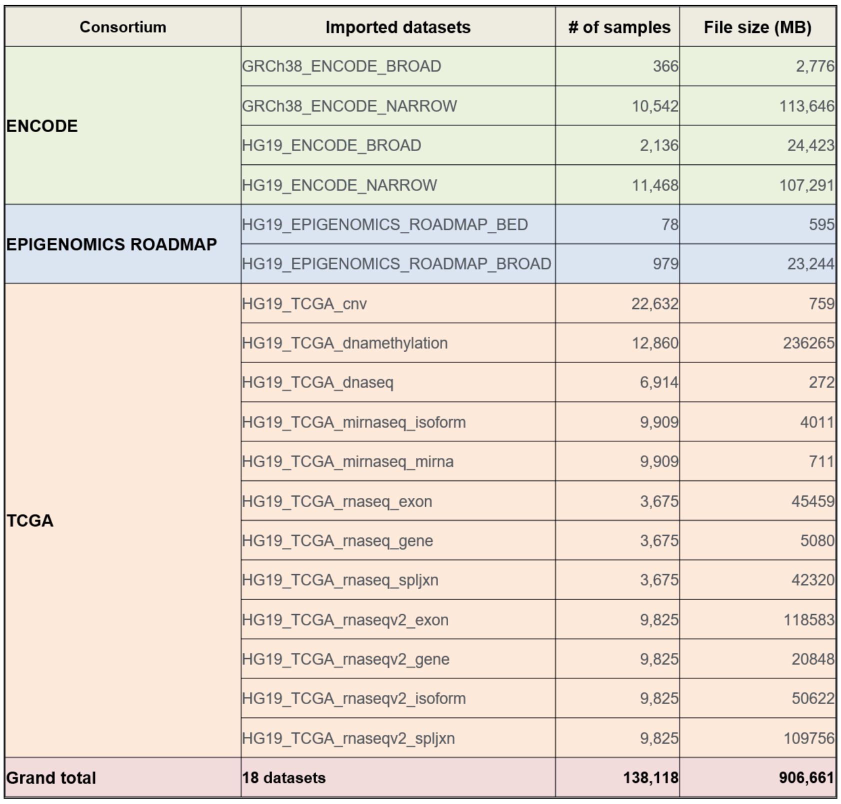


ID	ATTRIBUTE	VALUE
131	order	1
131	antibody	RBBP5
131	cell	H1-hESC
131	count	32028
133	order	2
133	antibody	SIRT6
133	cell	H1-hESC
133	count	30945
113	order	3
113	antibody	H2AFZ
113	cell	H1-hESC
113	count	30825

Fig. 2. Metadata of the query result

In our current GMQL installation, we provide global curated datasets which have been converted to GDM, including all processed datasets available in TCGA [29] and ENCODE [14]; the transformation of TCGA datasets to GDM is discussed in [12]. The repository (as of January 1^{st} 2018) contains 18 datasets and

138,118 samples for a total size of 906 GB, as illustrated in Fig. 3; we are working towards its extension to many other source datasets, and the conceptual organization and integration of their most relevant metadata. We are also currently working on the transformation of datasets from The Genomic Data Commons (GDC) [31], which includes the latest version of TCGA datasets.



Consortium	Imported datasets	# of samples	File size (MB)
ENCODE	GRCh38_ENCODE_BROAD	366	2,776
	GRCh38_ENCODE_NARROW	10,542	113,646
	HG19_ENCODE_BROAD	2,136	24,423
	HG19_ENCODE_NARROW	11,468	107,291
EPIGENOMICS ROADMAP	HG19_EPIGENOMICS_ROADMAP_BED	78	595
	HG19_EPIGENOMICS_ROADMAP_BROAD	979	23,244
TCGA	HG19_TCGA_cnv	22,632	759
	HG19_TCGA_dnamethylation	12,860	236265
	HG19_TCGA_dnaseq	6,914	272
	HG19_TCGA_mirnaseq_isoform	9,909	4011
	HG19_TCGA_mirnaseq_mirna	9,909	711
	HG19_TCGA_rnaseq_exon	3,675	45459
	HG19_TCGA_rnaseq_gene	3,675	5080
	HG19_TCGA_rnaseq_spljxn	3,675	42320
	HG19_TCGA_rnaseqv2_exon	9,825	118583
	HG19_TCGA_rnaseqv2_gene	9,825	20848
	HG19_TCGA_rnaseqv2_isoform	9,825	50622
	HG19_TCGA_rnaseqv2_spljxn	9,825	109756
Grand total	**18 datasets**	**138,118**	**906,661**

Fig. 3. Repository of processed open datasets

3 GMQL Implementation V1

The overall software architecture of GMQL V1 is shown in Fig. 4. From bottom to top, it includes the **repository layer**, the **engine layer** and the **GMQL layer**, which in turn consists of an **orchestrator** and a **compiler**, and is accessible through a **web service API**. We next briefly explain query execution, a detailed description can be found in [11]. Execution flow is controlled by the orchestrator, written in Java programming language; the processing flow includes compilation, data selection from the repository, scheduling of the Pig code execution over the *Apache Pig* engine [6], and storing of the resulting datasets in the repository in standard format.

When a user submits a GMQL query, the **orchestrator** calls the GMQL compiler, which produces the query translation into Pig Latin and the search criteria for loading the relevant samples from the repository; then, it uses the index manager to select from the repository the samples that comply to the search criteria. Then, it invokes the job optimizer, which sets the execution parameters (such as the parallelization factors); eventually, the orchestrator manages the outcome of the computation, including indexing of the result and storing it in the user space.

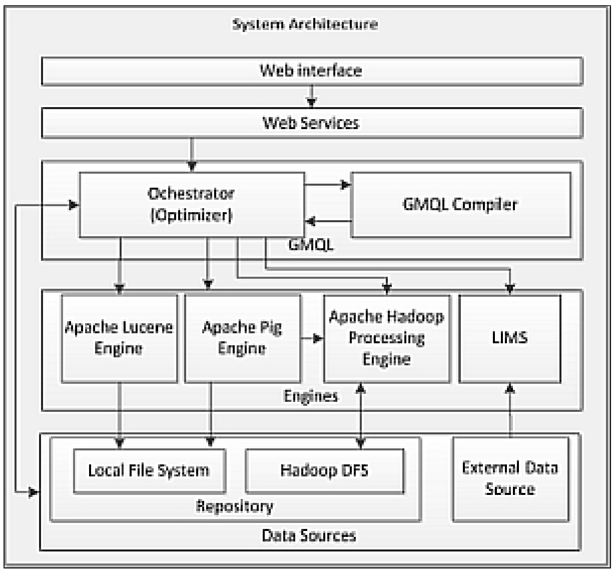
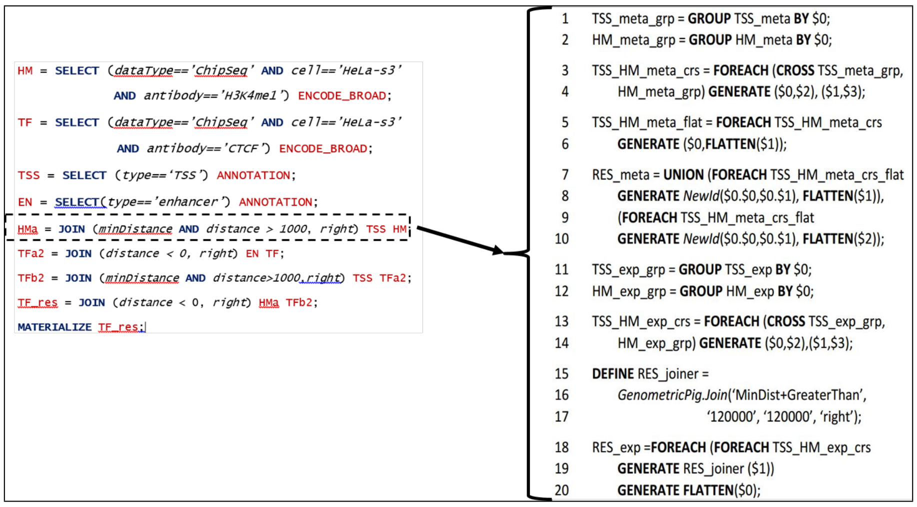


Fig. 4. Architecture of GMQL V1

The **repository** includes a *Local File System* (LFS), organized within the Linux file system of the master node of the computing framework, and an *Hadoop Distributed File System* (HDFS) [26], shared among all the computing nodes. Datasets are stored in the HDFS system, subdivided in metadata and region data. Both the LFS and the HDFS store the control data, which include the schema for each dataset (encoded in Extensible Markup Language - XML) and the *Apache Lucene* [5] indexes for metadata. Moreover, both file systems have a public and a private space. Datasets are stored in their original text format; when they are selected by a GMQL query, they are serialized by suitable adapters and translated to the internal binary GDM format. At that point, they are managed by Apache Pig under Hadoop 1. In this way, we do not replicate data in the native and GDM formats and we minimize data translations from native into GDM format. Version 1 of GMQL was installed at IEO-IIT (`https://www.ieo.it/en/` `http://genomics.iit.it/`), a center of excellence in oncology research, with a direct connection to a Laboratory Information Management System (LIMS) designed for storing processed data as well as the raw datasets and the pipelines for their extractions.



```
HM = SELECT (dataType=='ChipSeq' AND cell=='HeLa-s3'
             AND antibody=='H3K4me1') ENCODE_BROAD;
TF = SELECT (dataType=='ChipSeq' AND cell=='HeLa-s3'
             AND antibody=='CTCF') ENCODE_BROAD;
TSS = SELECT (type=='TSS') ANNOTATION;
EN = SELECT(type=='enhancer') ANNOTATION;
HMa = JOIN (minDistance AND distance > 1000, right) TSS HM;
TFa2 = JOIN (distance < 0, right) EN TF;
TFb2 = JOIN (minDistance AND distance>1000,right) TSS TFa2;
TF_res = JOIN (distance < 0, right) HMa TFb2;
MATERIALIZE TF_res;
```

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

Fig. 5. Translation of GMQL queries in V1

The **compiler** analyses each GMQL statement by performing syntactic and semantic checks; for valid statements, it then infers the schema of the new introduced variable, then updates the internal state and finally emits the Pig Latin code that performs the requested operation. Datasets are loaded into suitable Pig Latin variables; each dataset is mapped into two bags, named `V_region.dat` and `V_meta.dat`. The internal state contains the name and schema of each variable which is either generated or mentioned in the query. Fig. 5 shows the translation of a `JOIN` operation, where lines 1-10 are concerned with metadata and produce the data bag `RES_meta`, and lines 11-20 work on regions by encoding in Java programming language a fast-join algorithm which searches for matching regions at minimal and bound distance. The linking of metadata and regions of each output sample is guaranteed by the use of the same hash function on the two `ids` of the input pairs, at lines 8 and 10 for the metadata and within the Join function for the regions.

3.1 Discussion

The development of GMQL V1 was relatively fast, as the implementation took about one year; Pig Latin was chosen as target language because GMQL syntactically recalls Pig Latin - both languages progressively construct queries by introducing intermediate results and naming them by using variables; this style of query writing was considered as similar to the scripting style which is dominant in bioinformatics, although of course GMQL scripts are high-level and not intertwined with programming language constructs.

The main problem with this approach is that the optimization strategies were hardcoded within the software produced by the translator, which in turn was focused on the specific features of our target language Pig Latin. An additional problem of V1 was the use of Lucene for metadata indexing; it turned out that access to metadata is much faster than access to regions, hence the reduction of execution time achieved by using Lucene at query load was not compensated by the need of preserving Lucene data structures aligned with the spontaneous evolution of the repository.

Moreover, at the end of 2014, it became clear that Apache Pig was no longer strongly supported, while other engines were becoming much more popular; in particular Spark was achieving a dominant position, Flink had very interesting capabilities as a streaming engine, and SciDB had a totally different approach to storage through arrays. Therefore, at the beginning of 2015, we opted for a major system redesign, starting the development of GMQL Version 2.

4 GMQL Implementation V2

The design of GMQL V2 has been centered around the notion of an intermediate representation for the query language, developed as an abstract operator tree (actually a directed acyclic graph or DAG, as operations can be reused), that is at the center of the software architecture. The intermediate representation carries the semantics of the query language in terms of elementary operations that are applicable to metadata and to region data separately, and opens up to various options for expressing the language's syntax (which can be expressed as relational expressions, or in embedded form within programming or workflow languages, or in logical format by using a Datalog-like style) and for deploying over different cloud-based engines. In particular, at various times we used Spark, Flink and SciDB.

The intermediate representation is also the backbone for supporting several language implementations which embed GMQL operations within a programming paradigm (Fig. 6). We have developed a library for Python (called `gmql`), currently registered within the standard Python libraries; it enables running GMQL queries both on a local repository (on the user's desktop) and on a global repository (e.g., the CINECA installation). We are also developing interfaces for R and Galaxy.

A DAG for a complex query is illustrated in Fig. 7. The query consists of five SELECT, two JOIN, one COVER and one MAP statements. It has four input datasets (called `Annotation, Bed, Bed1, Bed2`), each in turn represented by separate metadata and region data structures. Blue nodes apply to metadata, red nodes apply to regions. The orientation of edges indicates that the nodes

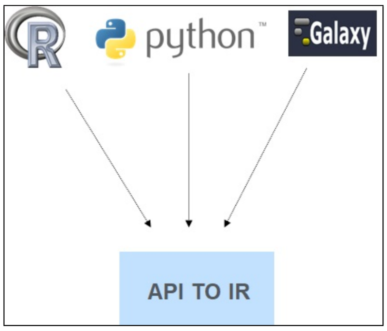
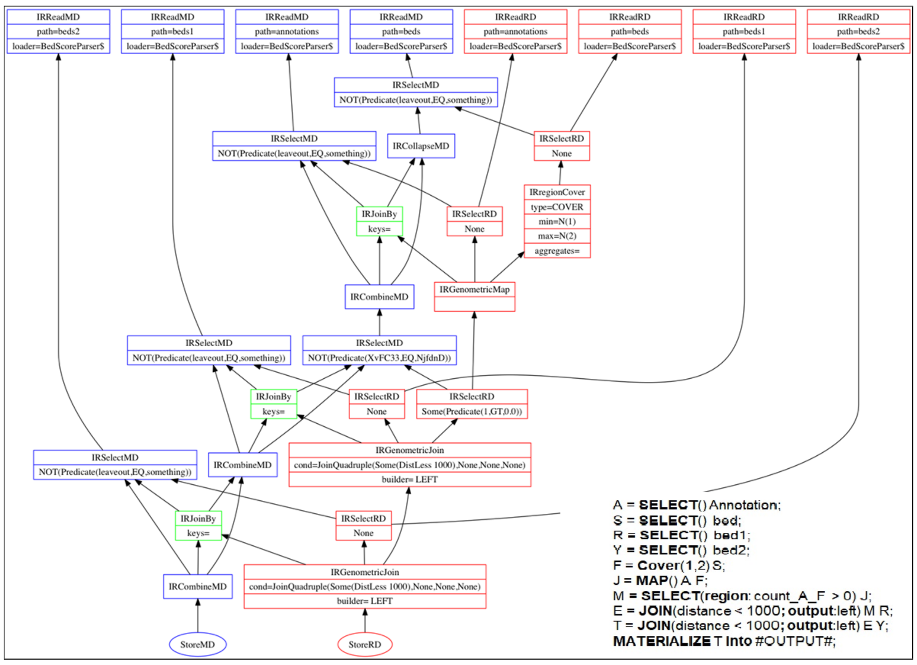


Fig. 6. Language interfaces for V2

are evaluated from the bottom to the top; indeed any evaluation occurs with the execution of the MATERIALIZE statement which indicates which variable of the program should be returned as result (in the specific example, variable `T`) and how the result variable should be named in the private repository of the user issuing the query (in this specific case, `#OUTPUT#`).


IRReadMD path=beds2 loader=BedScoreParser$
IRReadMD path=beds1 loader=BedScoreParser$
IRReadMD path=annotations loader=BedScoreParser$
IRReadMD path=beds loader=BedScoreParser$
IRReadRD path=annotations loader=BedScoreParser$
IRReadRD path=beds loader=BedScoreParser$
IRReadRD path=beds1 loader=BedScoreParser$
IRReadRD path=beds2 loader=BedScoreParser$
IRSelectMD NOT(Predicate(leaveout,EQ,something))
IRSelectMD NOT(Predicate(leaveout,EQ,something))
IRCollapseMD
IRSelectRD None
IRregionCover type=COVER min=N(1) max=N(2) aggregates=
IRJoinBy keys=
IRSelectRD None
IRCombineMD
IRGenometricMap
IRSelectMD NOT(Predicate(leaveout,EQ,something))
IRSelectMD NOT(Predicate(XvFC33,EQ,NjfdnD))
IRJoinBy keys=
IRSelectRD None
IRSelectRD Some(Predicate(1,GT,0.0))
IRSelectMD NOT(Predicate(leaveout,EQ,something))
IRCombineMD
IRGenometricJoin cond=JoinQuadruple(Some(DistLess 1000),None,None,None) builder= LEFT
IRJoinBy keys=
IRSelectRD None
IRCombineMD
IRGenometricJoin cond=JoinQuadruple(Some(DistLess 1000),None,None,None) builder= LEFT
StoreMD
StoreRD

```
A = SELECT() Annotation;
S = SELECT() bed;
R = SELECT() bed1;
Y = SELECT() bed2;
F = Cover(1,2) S;
J = MAP() A F;
M = SELECT(region: count_A_F > 0) J;
E = JOIN(distance < 1000; output:left) M R;
T = JOIN(distance < 1000; output:left) E Y;
MATERIALIZE T Into #OUTPUT#;
```

Fig. 7. DAG for a complex GMQL query

The DAG shows that each GMQL operation is mapped to one or more operations on the meta and region datasets. It also shows that the subtree constituted by the blue nodes, which apply to metadata, can be computed before the subtree constituted by the red nodes, which apply to regions. This property hints to a powerful optimization, called *meta-first*, which only applies to a subset of the GMQL queries - by virtue of such optimization, it is possible to load only regions of the samples which are used to construct the result.

We next briefly describe the software architecture of V2; similarly to V1, the architecture includes as principal components a database engine and a global repository[3]. The software of V2 is organized according to a four layer architecture:

- The **Access Layer** supports the various interfaces available in V2. These include an API to the DAG intermediate representation, a shell command line interface, and a collection of Web Services for accessing GMQL resources. Web Services are used for developing a user-friendly Web interface.
- The **Engine Components**, including the GMQL Compiler, for compiling a GMQL query into a DAG (which embodies execution plans); the DAG Manager, for supporting the creation and dispatching of DAG operations to other components; the Server Manager, for managing multi-user execution and their access capabilities; the Repository Manager, for managing the access to the repository; and the Launch Manager, for launching the executions of implementations.
- The **Implementation Components** (or executors), including the three implementations currently supported. Each package contains the implementation of the operations (abstract classes) of the DAG nodes, respectively for the Spark Implementation (the default option, and the most stable of the current implementations) and the Flink and SciDB implementations (which have been developed essentially for comparing their performances to the Spark implementation).
- The **Repository Implementations**, including a Local File System (LFS) repository, used when the installation is for a single machine, and a Remote File System (RFS), used in a cluster-based architecture. We currently support three installations, respectively at the CINECA Supercomputing center, on the Amazon Cloud, and on a dedicated local server available at our home institution.

The Repository Manager is the system component in charge of storing and managing the datasets imported from external repositories or generated by an user as result of a query execution. We support a private repository for each user and a public, read-only repository shared by all the users, which contains datasets from open public collections, such as ENCODE [14] and TCGA [29], as discussed in Section 2.

4.1 Optimization Options in V2

The new architecture opens up for multiple optimization options, illustrated in Fig. 8. In particular, a query optimizer applies deployment-independent op-

[3] GeCo V2 software is available at `https://github.com/DEIB-GECO/GMQL`.

timizations directly to the intermediate representation, either in the form of node reordering or of refinements of the initial selections; the former ones range from classic algebraic optimizations, e.g., pushing of selection conditions to the nodes where metadata are used for joining or grouping (operations JOIN, MAP, GROUP, and ORDER), or distribution of selections to binary operations (such as JOIN, MAP, UNION and DIFFERENCE), or inversion of the execution order of binary operations (such as JOINS with UNION or DIFFERENCE); these optimizations are subject to applicability constraints. In few cases, it is possible to perform also node deletions, e.g., when after optimization we can infer that meta predicates are contradictions. These optimizations produce an Optimized Intermediate Representation which is next used for the GMQL implementations.

Other optimizations are possible at a low level, and apply to specific implementations. In Version 2, we associate each dataset with a **profile**, which includes several parameters, such as: the number of samples, their sizes, the number of regions in each sample and the length of the genome where sample regions are distributed (these are typically a subset of the overall length of each chromosome, as the initial and final parts of the chromosomes are not typically involved in protein coding or epigenomically relevant regions). Therefore, it is possible to use alternative algorithm for deploying the various operations depending on the profiles of operands, or to dedicate a suitable number of execution nodes to given queries based on their expected load, or to optimally use data partitioning and caching, so as to evenly distribute the load to the various nodes of a cloud-based system.

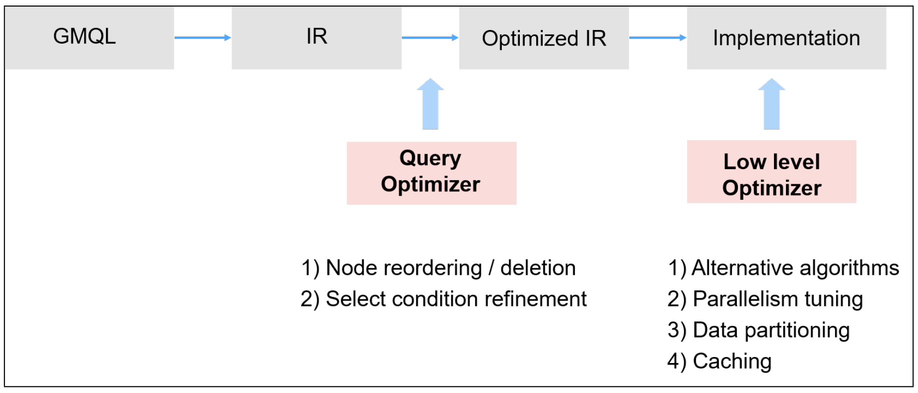


Fig. 8. Optimization options for GMQL V2

The most computationally expensive operations in GMQL require joining regions of two datasets; each dataset may have millions of regions, yielding to operations which potentially range over thousand of billions of region pairs. For coping with such requirements we use genome binning, illustrated in Fig. 9; binning is effective in supporting both effective parallelization and reduction of the pairs of regions to be considered for each join predicate. Binning has been used in the context of genome browsers to speed up query processing over regions [19]; its use in the map-reduce computations was proposed in [13].

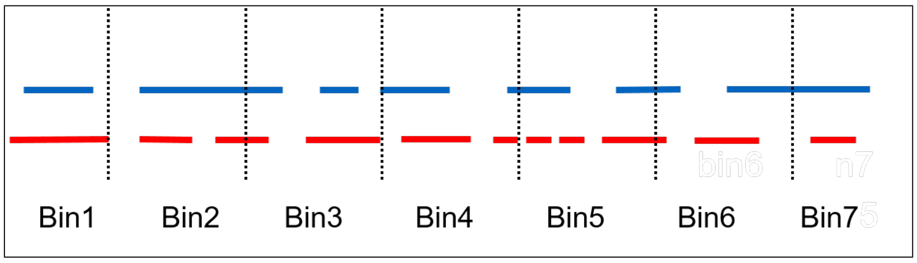


Fig. 9. Genome binning

Genome binning consists of partitioning the genome into equal-size segments; each region of the two operands of a joining operation is assigned to one or more bins, and the joins between regions are executed only within bins, yielding very effective parallel processing. Specifically, the first operand of the join is called *anchor* and the second operand is called *experiment*; the mapping of each anchor region to bins is defined by its *search space*, which in turns depends on the genometric join condition. Experiment regions are simply mapped to bins on the basis of their position. Algorithms discussed in [18] describe the binning algorithms in detail and show that each operation is associated with an *optimal bin size*. Intuitively, small bin sizes produce an excess of parallel execution, while large bin sizes produce an excess of execution of join operations within each bin.

4.2 Performance Comparison

A direct comparison between V1 and V2 is not very significant, because the two systems use different technologies and have been deployed over different platforms.

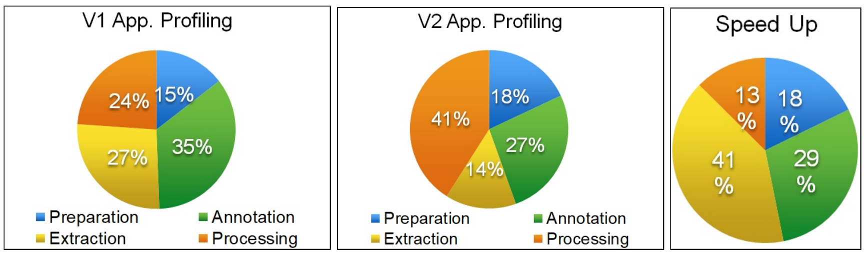


Fig. 10. Use of resources and spedup, V1 vs. V2

A comparative study was done at the time of the first release of V2, on the same platform, using the Spark engine for V2; the performance of V2 has significantly increased since its first release. We considered four GMQL query fragments, specifically dedicated to data preparation (with SELECT and COVER as dominant operation), differential data annotation (with EXTEND and DIFFERENCE as dominant operation), processing (with SELECT and JOIN dominant operations) and extraction of results (with MAP dominant operation). The comparison between V1 and V2, shown in Fig. 10, shows the use of resources for each query fragment and the speed-up in going from Version 1 to Version 2, ranging from 13% to 41%.

Other comparative studies regard the speed-up of specific operations. The left diagram of Fig. 11 corresponds to a MAP operation with a single reference and shows that both V1 and V2 scale linearly with the number of samples, but V2 has about half execution time; the right diagram of Fig. 11 corresponds to a MAP operation with multiple references (N=11) and shows that V1 does not scale linearly.

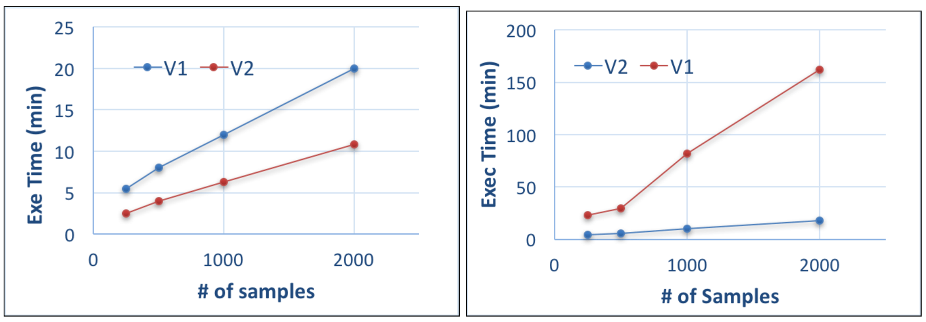


Fig. 11. Spedup of the MAP operation, V1 vs. V2

4.3 Discussion

Compared to V1, V2 has a much more complex software organization. While V1 was cooperatively developed by 3 PhD students, V2 has a larger group of developers, which included in the past several master students; the use of GitHub allowed for several branches to the main Spark implementation, which allowed us to test also the use of Flink and SciDB, served by suitable implementation branches. At the moment, the Spark implementation is fully supported, while the Flink and SciDB implementations are only maintained for specific operations.

Comparative analysis, published in [9] and [10], shows that the performance of Flink and Spark are remarkably similar, while the performance of Spark and SciDB are very different, with SciDB faster then Spark when operations involve selections and aggregates (as they are facilitated by an array organization);

whereas, Spark is faster than SciDB in JOIN and MAP operations (thanks to the general power of the Spark execution engine.)

5 Conclusions

The GMQL System is the center of several other tools and activities. We fully developed a Python library supporting an interface to the full GMQL language; the library is deployed within the international Python library repository and does not require any installation - as it is customary in Python. It supports a local and a remote execution mode, the former runs in the client desktop, the latter runs on a remote server and as such it connects to the global repository. Similar interfaces are being deployed for R and Galaxy.

We also developed a client-side system for data inspection and exploration, described in [17]; the tool connects directly to the results of GMQL queries and specifically to those queries which are completed by a MAP operation, as it is typically used to map known annotations (e.g., genes or epigenomically selected regions) to experimental datasets (e.g., gene expressions or mutations).

Our major current effort is dedicated to the development of an integrated repository for processed datasets, that extends the current repository described in Section 2. We are currently defining core metadata information that is normally available at all data sources, and then methods for extracting and normalizing such information; the conceptual model of the integrated repository is presented in [8]. All these efforts are coherently applied to tertiary data analysis, whose relevance is expected to grow within the near future.

Acknowledgment. This research is funded by the ERC Advanced Grant project GeCo (Data-Driven Genomic Computing), 2016-2021.

References

1. 1000 Genomes Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491, 56-65, 2012.
2. F. Albrecht et al. DeepBlue epigenomic data server: programmatic data retrieval and analysis of the epigenome. *Nucleid Acids Research*, 44(W1), W581-586, 2016.
3. Anonymous paper, Accelerating bioinformatics research with new software for big data to knowledge (BD2K), Paradigm4 Inc., 2015 downloaded from: `http://www.paradigm4.com/`).
4. Apache Flink. `http://flink.apache.org/`
5. Apache Lucene. `http://lucene.apache.org/core/`
6. Apache Pig. `http://pig.apache.org/`
7. Apache Spark. `http://spark.apache.org/`
8. A. Bernasconi et al. Conceptual modeling for genomics: building an integrated repository of open data. In: *Proc. Entity-Relationship*, Valencia, ES, 2017.
9. M. Bertoni et al. Evaluating cloud frameworks on genomic applications. *Proc. IEEE Conference on Big Data Management*, Santa Clara, CA, 2015.
10. S. Cattani et al. Evaluating big data genomic applications on SciDB and Spark. *Proc. Web Engineering Conference*, Rome, IT, 2017.

11. S. Ceri et al. Data management for heterogeneous genomic datasets. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 14(6), 1251-1264, 2016.
12. F. Cumbo et al. TCGA2BED: extracting, extending, integrating, and querying The Cancer Genome Atlas. *BMC Bioinformatics*, 18(6), 1-9, 2017.
13. B. Chawda et al. Processing interval joins on Map-Reduce. In *Proc. EDBT*, 463-474, 2014.
14. ENCODE Project Consortium. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414), 57-74, 2012.
15. FireCloud. `https://software.broadinstitute.org/firecloud`
16. Hadoop 2. `http://hadoop.apache.org/docs/stable/`
17. V. Jalili et al. Explorative visual analytics on interval-based genomic data and their metadata. *BMC Bioinformatics*, 18, 536, 2017.
18. A. Kaitoua et al. Framework for supporting genomic operations, *IEEE-TC*, 10.1109/TC.2016.2603980, 2016.
19. W.J. Kent. The human genome browser at UCSC. *Genome Research*, 12(6), 996-1006, 2002.
20. M. Masseroli et al. GenoMetric Query Language: a novel approach to large-scale genomic data management. *Bioinformatics*, 31(12), 1881-1888, 2015.
21. M. Masseroli et al. Modeling and interoperability of heterogeneous genomic big data for integrative processing and querying. *Methods*, 111, 3-11, 2016.
22. C. Olston et al. Pig Latin: A not-so-foreign language for data processing. *ACM-SIGMOD*, 1099-1110, 2008.
23. A. Roy et al. Massively parallel processing of whole genome sequence data: An in-depth performance study. In *ACM Sigmod*, Boston, MA, 2017.
24. S. C. Schuster. Next-generation sequencing transforms today's biology. *Nature Methods*, 5(1), 16-18, 2008.
25. SciDB. `http://www.scidb.org/`
26. K. Shvachko et al. The Hadoop distributed file system. In *Proc. MSST*, 1-10, 2010.
27. Z. D. Stephens et al. Big data: astronomical or genomical? *PLoS Biology*, 13(7), e1002195, 2015.
28. R. C. Taylor et al. An overview of the Hadoop MapReduce HBase framework and its current applications in bioinformatics, *BMC Bioinformatics*, 11(Suppl 12), S1, 2010.
29. J. N. Weinstein et al. The Cancer Genome Atlas Pan-Cancer analysis project. *Nature Genetics*, 45(10), 1113-1120, 2013.
30. M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX*, 15-28, 2012.
31. M. A. Jensen et al. The NCI Genomic Data Commons as an engine for precision medicine. *Blood*, 130(4), 453-459, 2017.