Data Management for Heterogeneous Genomic Datasets

Stefano Ceri, Abdulrahman Kaitoua, Marco Masseroli, Pietro Pinoli, Francesco Venco

Abstract—Next Generation Sequencing (NGS), a family of technologies for reading the DNA and RNA, is changing biological research, and will soon change medical practice, by quickly providing sequencing data and high-level features of numerous individual genomes in different biological and clinical conditions. Availability of millions of whole genome sequences may soon become the biggest and most important "big data" problem of mankind. In this exciting framework, we recently proposed a new paradigm to raise the level of abstraction in NGS data management, by introducing a GenoMetric Query Language (GMQL) and demonstrating its usefulness through several biological query examples. Leveraging on that effort, here we motivate and formalize GMQL operations, especially focusing on the most characteristic and domain-specific ones. Furthermore, we address their efficient implementation and illustrate the architecture of the new software system that we have developed for their execution on big genomic data in a cloud computing environment, providing the evaluation of its performance. The new system implementation is available for download at the GMQL website (http://www.bioinformatics.deib.polimi.it/GMQL/); GMQL can also be tested through a set of predefined queries on ENCODE and Roadmap Epigenomics data at http://www.bioinformatics.deib.polimi.it/GMQL/queries/.

Index Terms—Genomic data management, Operations for genomics, Data modeling, Query languages, Cloud-based systems.

1 Introduction

Next Generation Sequencing (NGS), also known as high-throughput sequencing, is a family of technologies for reading the DNA and RNA precisely, quickly and cheaply [1], [2]; in the next decade, it will offer fast (few hours) and inexpensive (hundreds of dollars) readings of the whole human genome [3]. Large-scale sequencing projects are spreading, and huge amounts of sequencing data are continuously collected by a growing number of research laboratories, often organized through world-wide consortia (such as ENCODE [4], TCGA [5], 1000 Genomes Project [6], or Roadmap Epigenomics [7]). Answers to fundamental questions for biological and clinical research are hidden in these data, e.g., how protein-DNA interactions and DNA three-dimensional conformation affect gene activity, how cancer develops, how driving mutations occur, how much complex diseases such as cancer are dependent on personal genomic traits or environmental factors. Personalized and precision medicine based on genomic information is becoming a reality; the potential for data querying, analysis and sharing may be considered as the biggest and most important big data problem of mankind.

So far, the bioinformatics research community has been mostly challenged by the so-called NGS *primary analysis* (production of "reads", i.e., nucleotide sequences representing short DNA or RNA segments) and *secondary analysis* (alignment of reads to a reference genome and search for specific features on the reads, such as variants/mutations and peaks of expression); but the most important emerging problem is the NGS *tertiary analysis*, concerned with multi-sample processing of heterogeneous information, annotation and filtering of variants, and integration of genomic features (e.g., specific DNA variants, or signals and peaks of expression, i.e., DNA regions with higher density of reads). While secondary analysis targets raw data in output from NGS machines by using specialized methods, tertiary analysis targets processed data in output from secondary analysis, and it is responsible of sense making, e.g., discovering how heterogeneous regions interact with each other (Fig. 1).

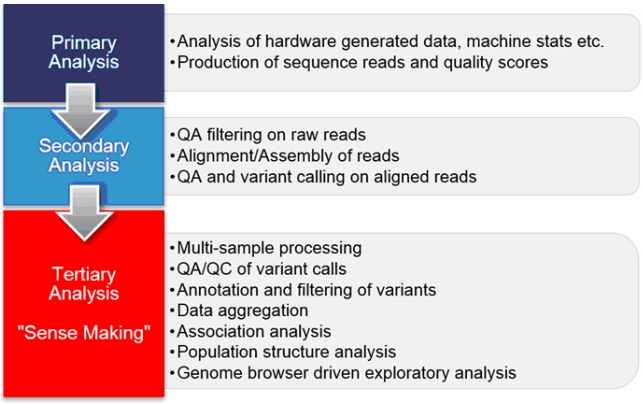

Fig. 1. Phases of genomic data analysis, source: http://blog.goldenhelix.com/grudy/a-hitchhikers-guide-to-next-generation-sequencing-part-2/

NGS data are managed by a variety of tools focused on *ad-hoc* processing targeted to specific tasks, data extractions and transformations (e.g., alignment to a reference, mutation and peak calling, reading of gene expression); each tool manages data in specific technology-driven formats, with no emphasis on interoperability, format-independent representations, powerful abstractions, and out-of-the-box thinking and scaling. Cloud-based approaches to genomics

• *S. Ceri, A. Kaitoua, M. Masseroli, P. Pinoli and F. Venco are with the Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy. E-mail: {firstname.lastname}@polimi.it*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2016.2576447, IEEE/ACM Transactions on Computational Biology and Bioinformatics

have been targeted to speeding up specific data extraction, transformation and analysis processes, but not to combining results from different data processes.

In this scenario, we recently proposed a new holistic approach to genomic data modeling and querying[1] that takes advantage of cloud-based computing to manage heterogeneous data produced by NGS technology. In [8], we introduced the novel *GenoMetric Query Language* (GMQL), built on an abstracted model for genomic data; we sketched out its main operations and demonstrated its usefulness, expressive power and flexibility through multiple different examples of biological interest (including finding distal bindings in transcription regulatory regions, associating transcriptomics and epigenomics, and finding somatic mutations in exons).

Here, we fully formalize GMQL operations, focusing on domain-specific operations for genomics, and then we address their efficient implementation in the context of standard cloud systems; we provide full evaluation of their performance, compare them with the state-of-the-art, and show that indeed they scale linearly with the number of considered regions, which can rise up to hundreds of millions. We focus on three operations: `COVER`, which computes genomic region-based operations such as unions of intersections; `JOIN`, which compares genomic regions by distance, typically subject to minimality or cardinality constraints; and `MAP`, which changes reference regions for looking at experiments.

A new, web service based, system implementation of GMQL is freely provided for download at http://www.bioinformatics.deib.polimi.it/GMQL/, in which GMQL is translated into *Apache Pig* [9] and executed on the *Apache Hadoop YARN* framework [10]. GMQL can also be tested in a web application that provides a set of predefined, parametric GMQL queries on ENCODE and Roadmap Epigenomics data, available at http://www.bioinformatics.deib.polimi.it/GMQL/queries/. Thus, while [8] proves user focused innovation for biologists, this paper presents solid domain-specific innovation for genomics in cloud-based data management, associated with a new implementation which operates on any server or cloud supporting Hadoop and Pig.

The remaining of this paper is structured as follows. Section 2 presents the state of the art and discusses other related current approaches and implementations. Section 3 provides the formalization of the syntax and semantics of the defined Genomic Data Model, as well as its motivation and examples. Then, Section 4 dwells into the description and formalization of our proposed GenoMetric Query Language, discussing first its general operations, then its most characteristic and domain-specific ones, including `COVER`, `JOIN` and `MAP`; for each of them motivation, syntax and semantics, and examples are provided. Section 5 illustrates the architecture of the new software system that we developed for the execution of GMQL queries on big genomic data, focusing on its repository layer and on the GMQL orchestrator/optimizer. In Section 6 and Section 7 we dwell into the details of the implementation of our proposal for big data management support in a cloud computing environment, including parallelism options and optimal implementation of some domain specific operations. Section 8 illustrates and discusses the performance evaluation of our implementation, both with respect to some state-of-the-art alternatives and to scaling with big datasets, and provides a full GMQL query example with its biological interpretation. Section 9 concludes.

With our approach, very complex genomic operations can be written as simple queries, with implicit iteration over thousands of heterogeneous samples, and computed in few minutes over servers or clouds supporting Hadoop and Pig. Bringing genomics to the cloud is becoming more and more essential [11]; we apply cloud computing to genomic processed data, with the objective of querying thousands of heterogeneous samples. Other available cloud computing tools start from raw read data or aligned DNA sequences, and are complementary to our system. A unique aspect of our approach is to include metadata in the data model and in the query language. Each dataset includes metadata describing properties of the data samples; GMQL progressively builds new datasets out of existing ones, thus the query result also carries metadata, as an indication of data provenance.

2 Related Work

Several organizations are considering genomics at a global level. *Global Alliance for genomics and Health*[2] is a large consortium of over 200 research institutions with the goal of supporting voluntary and secure sharing of genomic and clinical data; their work on data interoperability is producing a conversion technology for the sharing of data on DNA sequences and genomic variation [12]. *Google* recently provided an API to store, process, explore, and share DNA sequence reads, alignments and variant calls, using Google's cloud infrastructure [13].

We compare our work with recent papers on genomic data management. Works by Röhm and Blakeley [14], by Tata, Patel et. al. [15], [16], and by Bafna et al. [17], [18] address the querying of NGS data using either the Structured Query Language (SQL) (in the former case) or SQL extensions (in the latter two cases). Use of plain SQL was attempted in [14], which highlights the performance bottlenecks of conventional SQL optimization when dealing with domain-specific functions and parallelization. Tata, Patel et. al. developed *Periscope* [15], [16], a system supporting matching operators over DNA sequences, encoded as character strings; they report fast execution times and some collaborations working on the full mouse genome. Several domain-specific extensions are proposed in [17] to overcome SQL limitations in expressing genomic computations.

Each of these systems works on raw read data or aligned DNA sequences and highlights specialized processing for given aspects (e.g., sequence matching or genomic feature calling) as they must reproduce genome processing tasks which are normally solved by ad-hoc specific tools (e.g., aligners, and mutation or peak callers). Carrying out such tasks from within an integrated system is potentially very effective, as no information is left outside of the query system, but there is as well a big risk of giving up the

1. http://www.bioinformatics.deib.polimi.it/genomic_computing/
2. http://genomicsandhealth.org/

quality of specialized tools; moreover, some widely available experimental data (e.g., from ENCODE or Roadmap Epigenomics) include datasets resulting after the calling processes. None of these systems integrates metadata within their computations, which are only addressed to genomic data.

Other works have proposed the embedding of query processing functions within libraries that can be integrated within programs [19], [20]. In particular, [20] presents a rather elegant mathematical formalism, based on set algebra; its authors propose a genomic region abstraction (that may represent reads, genomic variants, mutations, and so on) and then define a set of region operations, delivered as the *Genomic Region Operation Kit* (GROK) library. In comparison, GROK supports lower-level abstractions than GMQL and some low-level operations (e.g., flipping regions) that are not directly supported by GMQL, but they must be embedded into C++ programming language code. Furthermore, high-level declarative operations, such as `JOIN` and `MAP`, can be encoded in GROK, but they must be invoked from command line editors or C++ programs. GROK shows excellent performance on desktop systems, but it is unsuitable for parallelization and does not deal with metadata.

Several other tools focus on specific data formats or tackle specific needs and processing requirements. Among them, *BEDTools* [21] and *BEDOPS* [22] apply to the BED format; they efficiently process region data, but of individual samples (at most two at a time, for their comparison), requiring verbose scripts for multiple sample processing, with lower performance. Moreover, they cannot manage sample metadata and do not have an internal standard for data format; this may lead different scripts to generate different output formats, with the need for external languages (e.g., AWK, or SED) to manipulate their outputs, which increases their script verbosity. A functional comparison of these tools with GMQL is published as supplemental material to [8], where we illustrate how biologists would comparatively build a long query with the three approaches. BEDTools and BEDOPS can be used from within software environments for bioinformatics (e.g., *BioPerl*, *BioPython*, *R* and *Bioconductor*), but are not designed for cloud computing.

A recent work by Nordberg et al. [23] presents *BioPig*, a set of extensions for specific NGS analysis tasks to the *Pig Latin* data processing language [24]. BioPig includes three modules that can be used in the early phase of NGS data analysis for processing the raw read data files produced by NGS machines. *SeqPig* [25] is another collection of similar modules to manipulate, analyze and query sequencing datasets. The work by Weiwiorka et al. [26] presents analogous analysis tasks implemented on *Apache Spark* [27]. All these works are complementary to our, as they apply on NGS read data instead of on processed data, and indicate a growing interest in parallel processing for genomics.

Besides cloud-based systems, scientific databases can also be used to support genomic computing, including *Vertica* [28] (used by Broad Institute and NY Genome Center) and *SciDB* [29] (further enhanced by *Paradigm4*,[3] a company whose products include genomic adds-on to SciDB and access to NGS data from TCGA and 1000 Genomes Project.)

3. http://www.paradigm4.com/

3 Genomic Data Model

The proposed Genomic Data Model (GDM) is based on the notions of datasets and samples; datasets are collections of samples, and each sample consists of two parts, the *region data*, which describe portions of the DNA, and the *metadata*, which describe sample general properties.

3.1 Motivation

GDM stands as a new data model, with distinguishing features. In contrast to other data models and LIMS [30], it clearly divides observations, i.e., genomic regions, from metadata, i.e., available information about how observations are collected. For the former, GDM provides a flat attribute-based organization, by imposing that each dataset is associated with a given data schema; the first five attributes of such schema are fixed and represent the sample identity and genomic coordinates. We built adapters from several processed data types to GDM schemas, and we will translate more processed data types in the future, as they will become available.

Instead, in GDM metadata are free-format, attribute-value pairs, since metadata are usually unstructured and very arbitrary. We expect metadata to include at least the experiment type, the sequencing and analysis method used for data production, the cell line, tissue, experimental condition (e.g., antibody target for NGS chromatin immunoprecipitation sequencing (ChIP-seq) samples) and organism sequenced; in case of clinical studies, individual's descriptions including phenotypes.

Tens of thousands of samples of different types and hundreds of millions of processed data describing genomic regions from ENCODE, TCGA and Roadmap Epigenomics, collected in our system repository, can be comprehensively queried thanks to their modeling using the GDM.

3.2 Syntax and Semantics

A **genomic region** r_i is a well-defined portion of the genome qualified by a quadruple of values, called **region coordinates**: $r_i =< chr, left, right, strand >$ where chr is the chromosome[4] where the region is located, $left$ and $right$ are the two ends of the region along the DNA coordinates, and $strand$ represents the direction of the DNA region reading[5], encoded as either '$+$' or '$-$', and can be missing (encoded as '$*$')[6]. Regions have associated properties (i.e., values) which are detected by the post-processing of DNA or RNA sequencing reads. Metadata describe the biological and clinical properties associated with each sample; due to the great heterogeneity of this information, they are represented as arbitrary attribute-value pairs.

4. The simplest way of considering DNA is as a string of billions of elements (i.e., nucleotides, represented by the letters `A`, `C`, `G`, `T`) enclosed within chromosomes (23 in humans), which are disconnected intervals of the string.

5. DNA is made of two strands rolled-up together in anti-parallel directions, i.e., they are read in opposite directions by the biomolecular machinery of the cell.

6. According to the UCSC notation, we use *0-based, half-open inter-base coordinates*, i.e., the considered genomic sequence is $[left, right)$. Left and right ends can be identical (e.g., when the region represents a single nucleotide insertion).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2016.2576447, IEEE/ACM Transactions on Computational Biology and Bioinformatics

ID	(CHR,LEFT,RIGHT,STRAND)	P_VALUE
1	(1, 3245, 4535, +)	0.000024
1	(1, 6340, 7400, -)	0.000053
1	(1, 7540, 8563, -)	0.000013
1	(2, 1440, 2506, -)	0.000034
1	(2, 3540, 4541, +)	0.00006
2	(1, 4020, 5073, *)	0.000017
2	(1, 7020, 8061, *)	0.000035
2	(2, 1020, 3064, *)	0.000016
2	(2, 4020, 4101, *)	0.000022

ID	ATTRIBUTE	VALUE
1	cell	CLL
1	kariotype	cancer
1	tissue	blood
1	sex	M
2	cell	H9ES
2	sex	F
2	tissue	embryonic

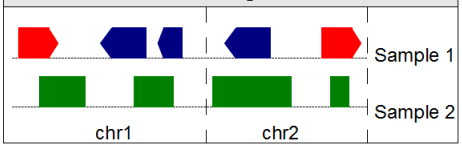

Fig. 2. Regions and metadata of a dataset consisting of two samples.

Formally, a **sample** s is modeled as the following triple $< id, \{< r_i, v_i >\}, \{m_j\} >$, where:

- *id* is the sample identifier, of type `long`.
- Each region is a pair of **coordinates** r_i and **values** v_i. Coordinates are records of the four fixed attributes `chr`, `left`, `right`, `strand`, which are typed `string`, `long`, `long`, `char`, respectively; values are records of typed attributes. We assume attribute names of a sample to be different, and types to be any of `Boolean`, `char`, `string`, `int`, `long`, `double`. The **region schema** of s is the list of attribute names used for coordinates and values.
- Metadata m_j are attribute-value pairs $< a_j, v_j >$, where we assume the type of each attribute a_j and value v_j to be `string`.

A **dataset** is a collection of samples with the same region schema. Each dataset is typically produced within the same project (either at a genomic research center or within an international consortium) using the same technology and tools, but in different experimental conditions; such difference is typically described by the sample metadata. Sample identifiers are unique within each dataset.

3.3 Examples

Each dataset is stored using two normalized data structures, one for regions and one for metadata; an example of these two data structures is shown in Fig. 2. Note that the region values have an attribute `p_value` of type `double` (representing how significant is the peak of expression in that genomic region). Note also that the `id` attribute provides a many-to-many connection between regions and metadata of a sample; e.g., sample 1 has *five* regions and *four* metadata attributes, while sample 2 has *four* regions and *three* metadata attributes. In the bottom part of Fig. 2, the regions of the two samples are aligned along two lines representing the chromosomes 1 and 2 of the DNA; regions of the first sample are stranded (positively or negatively), while regions of the second sample are not stranded.

Although the above example is simple, GDM supports the schema encoding of many types of processed data in different formats, including the BED, bedGraph, broadPeak, narrowPeak, VCF and GTF standard data formats; they have five common attributes, and differ in the subsequent attributes of the schema. Schemas and one example instance of DNA-seq (mutation) and RNA-seq (gene expression) data are:

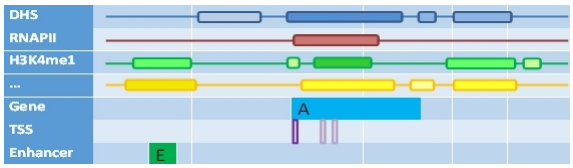

Fig. 3. Regions from heterogeneous samples.

```
DNA-seq
(id, (chr, left, right, strand),
 (A, G, C, T, del, ins, inserted, ambiguous,
  Max, Error, A2T, A2C, A2G, C2A, C2G, C2T))
(1, (chr1, 917179, 917180, +),
 (0, 0, 0, 0, 1, 0, '.', '.',
  0, 0, 0, 0, 0, 0, 0, 0))

RNA-seq
(id, (chr, left, right, strand),
 (source, type, score, frame, geneID,
  transcriptID, RPKM1, RPKM2, iIDR))
(1, (chr8, 101960824, 101964847, *),
 ('GencodeV10', 'transcript', 0.026615, NULL,
  'ENSG00000164924.11', 'ENST00000418997.1',
  0.209968, 0.193078, 0.058))
```

Fig. 3 shows, in an abstract form, regions from heterogeneous samples, i.e., containing different types of data, which are modeled in GDM. Data are stored in file systems as discussed in Section 5.1.

4 GENOMETRIC QUERY LANGUAGE

In this Section we provide detailed description and formalization of our GenoMetric Query Language, discussing its general operations first and then its most typical and domain-specific operations, providing for each of them motivation, syntax and semantics, and an example.

A GMQL query (or program) is expressed as a sequence of GMQL operations, each with the following structure:

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

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

4.1 Motivation

GMQL is inspired by Pig Latin [24]; as in that data processing language, a GMQL program is a sequence of steps, much like in a programming language, each of which carries out a single, high-level data transformation, like in SQL. We agree with [24] that *to experienced system programmers, this method is much more appealing than encoding their task as an*

SQL query. Moreover, an operator-based language is much more flexible, as it can be easily extended in order to meet domain-specific requirements.

GMQL operations form a closed algebra: results are expressed as new datasets derived from their operands. Thus, operations typically have a region-based part and a metadata part; the former one builds new regions, the latter one traces the provenance of each resulting sample. Most GMQL operations are intuitive for readers who are knowledgeable of relational algebra; we informally define these operations first, and then we focus on the most characteristic, domain-specific operations.

4.2 Standard GMQL Operations

Before dwelling into GMQL operations, we describe some common syntactic elements. Parameters of several operations include predicates, used to select and join samples; predicates are built by arbitrary Boolean expressions of simple predicates, as it is customary in relational algebra. Predicates on regions must use the attributes in the region's schema; predicates on metadata may use arbitrary attributes. Thus, when a predicate on regions uses an illegal attribute, the query is also illegal; when a predicate on metadata uses an attribute which is not present, the predicate is `unknown`, but the query is legal. We use a three-value logic (`true`, `false`, `unknown`) for metadata predicates p and consider only those samples s for which $p(s)$ is `true`.
The main standard GMQL unary operations are:

- `SELECT`: applies on metadata and keeps in the result the input dataset samples which satisfy a metadata predicate; their metadata and region data are kept unchanged.
- `ORDER`: uses metadata attributes to order the samples of a dataset, by adding to each sample an `order` metadata attribute with the sample ranking value, and possibly to filter the top samples based upon the ordering.
- `AGGREGATE`: computes aggregate functions over region values of each sample of a dataset and adds the result as new metadata attributes of the sample.
- `PROJECT`: applies on regions and keeps in the result the input region attributes expressed as parameters. It can also be used to build new region attributes as scalar expressions of region attributes (e.g., the `length` of a region as the difference between its `right` and `left` ends). Metadata are kept unchanged.
- `MERGE`: merges all the samples of a dataset into a single sample, having as regions all the input regions and as metadata the union of the sets of input attribute-value pairs of the dataset samples.

Two GMQL binary operations allow building unions or differences of datasets and samples.

- `UNION`: applies to two datasets and builds their union, so that each sample of each operand contributes exactly to one sample of the result; if datasets have different schemas, the result schema is the union of the two sets of attributes of the operand schemas, and in each result sample the values of the attributes missing in the original operand of the sample are set to null. Metadata of each sample are kept unchanged.
- `DIFFERENCE`: applies to two datasets and preserves in output the regions of the first dataset which do not intersect with any region of the second dataset; only the metadata of the first dataset are maintained, unchanged.

We next focus on **domain-specific operations**, responding to particular genomic requirements: the unary operation `COVER`, with its variants `SUMMIT` and `FLAT`, and the binary operations `JOIN` and `MAP` [7].

4.3 COVER Operation

The `COVER` operation applies to a single dataset and produces a single sample from all the dataset samples (each of which may include overlapping regions).

4.3.1 Motivation

The `COVER` operation is widely used in order to select regions which are present in a given number of samples; this processing is typically used in the presence of overlapping regions, or of replicate samples of the same experiment. The grouping option allows grouping samples with similar experimental conditions, and produces a single sample for each group.

4.3.2 Syntax and Semantics

The syntax allows for several optional parts[8]:

```
<S2> = COVER[_FLAT|_SUMMIT](<minAcc>,<maxAcc>
    [; <aggregs>] [GROUP_BY <attribs>]) <S1>
```

Resulting regions of the result sample in `S2` are non-overlapping and are built from the regions of the samples in `S1` complying with the following conditions:

1) 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`; `minAcc` and `maxAcc` are called **accumulation indexes**[9].
2) Resulting regions may have new attributes calculated by means of aggregate expressions (`<aggregs>`) over the attributes of the contributing regions. The `Jaccard Index`[10], a standard measure of similarity of the contributing regions r_i, is added as default attribute.
3) Metadata of the contributing samples are merged, and only their distinct attribute-value pairs are kept.
4) When a `GROUP_BY` clause is present, samples in `S1` are first partitioned by groups, each with a distinct value

7. These operations have a biological interpretation. `COVER` is used to merge multiple replicate samples, or to identify DNA areas where multiple different transcription factor binding sites or histone modifications occur. `JOIN` is set to detect (epi)genomic features (e.g., mutations, transcription factor binding sites, histone modifications, methylation sites) overlapping or at a certain distance from other genomic features or DNA know structural annotations (e.g., gene transcription start sites). `MAP` is used to discover how selected (epi)genomic features map to reference regions, e.g., how many mutations occur within certain genes.

8. In the syntax, square brackets [] denote optionality, and plurals denote optional repetitions.

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

10. The `Jaccard Index` is calculated as the ratio between the intersection and the union of the contributing regions.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2016.2576447, IEEE/ACM Transactions on Computational Biology and Bioinformatics

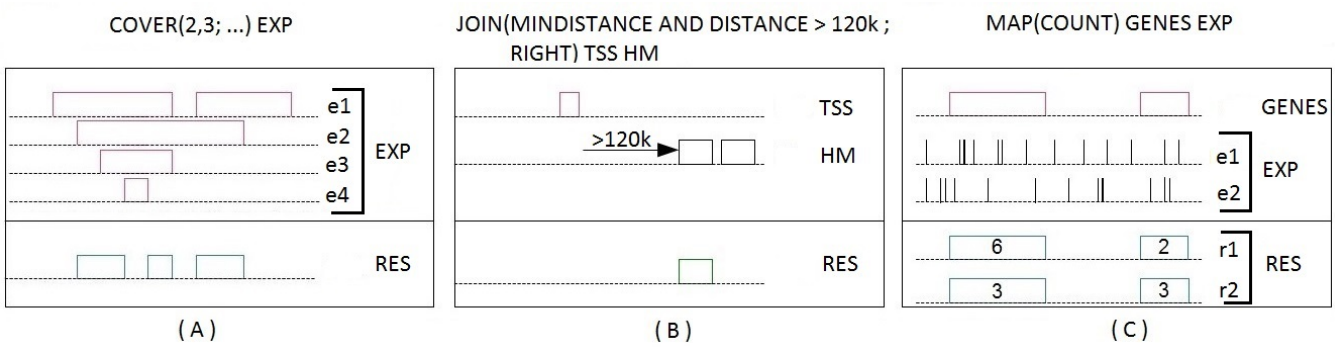

Fig. 4. Sketches of domain-specific GMQL operations used in the motivational examples of Section 4.

of the grouping metadata attributes (<attribs>), and then the cover operation is separately applied to each group, yielding to one sample in the result for each group.

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 input regions which would contribute to each result region of the COVER). The _SUMMIT variant returns only those portions of the result regions of the COVER where the maximum number of input 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).

4.3.3 Example

The following COVER operation produces output regions in the RES dataset where at least 2 and at most 3 regions in the EXP dataset overlap, having as resulting region attributes the max p_value of the overlapping regions and their JaccardIndex; the result has one sample for each input antibody_target value. Fig. 4(A) shows a sketch of the operation applied to an experiment dataset regarding a single antibody target; note that there are four input samples and one result sample, whose regions represent the intervals where there are between 2 and 3 input regions).

```
RES = COVER(2, 3; p_value AS MAX(p_value)
            GROUP_BY antibody_target) EXP;
```

4.4 JOIN Operation

The JOIN operation applies to two operands, called **left** and **right** datasets, and acts in two phases: first, a **meta join predicate** builds new samples from pairs of samples, one of the right and one of the left dataset. Then, a **genometric join predicate**, dealing with distal properties of regions, selects the regions to include in these new samples.

4.4.1 Motivation

Meta join predicates allow selecting sample pairs with appropriate biological conditions (e.g., regarding the same cell line or antibody target); genometric join predicates allow expressing distal conditions on sample regions, which are needed by biologists when they extract pairs of regions on the genome based on their relative position.

4.4.2 Syntax and Semantics

The syntax of JOIN is:

```
<S3> = JOIN([<meta-pred>;] <genometric-pred>;
            <region-constr>) <S1> <S2>
```

Given two datasets S1 and S2, let s_1 denote a sample of S1 and s_2 denote a sample of S2. The join meta predicate is the conjunction of simple predicates <a_1> <comparison> <a_2> (with a_1 and a_2 metadata attributes of s_1 and s_2, respectively). Each simple predicate evaluates to true when s_1 has a pair $< a_1, v_1 >$, s_2 has a pair $< a_2, v_2 >$, and the comparison predicate applied to v_1 and v_2 is true; it evaluates to false otherwise. In this way, pairs $< s_i, s_j >$ of samples are selected from the original datasets; when the join meta predicate is not present, all the sample pairs in the Cartesian product of S1 and S2 are selected[11].

Next, we first discuss the structure of resulting samples, and then the syntax and semantics of genometric join predicates. Assume that the genometric join predicate, applied to regions r_i of s_i and r_j of s_j, is true. Then, new regions r_{ij} are computed by applying the constructor <region-constr> to the regions r_i of s_i and r_j of s_j; the constructor has four options:

1) LEFT returns the left region (i.e., the region with the coordinates of r_i);
2) RIGHT returns the right region (i.e., the region with the coordinates of r_j);
3) INT returns the region intersection (i.e., the region with the coordinates of the common elements of r_i and r_j); if the intersection is empty, no region is produced by the pair $< r_i, r_j >$. If no regions are produced for a given sample s_{ij}, then the sample s_{ij} is not created;
4) CAT returns 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).

The schema of regions r_{ij} is obtained as composition of the schemas of s_i and s_j. It includes a new identifier, the region coordinates and then the concatenation of the other attributes of the schemas of s_i and s_j, respectively (by disambiguating their names with their original dataset name, if necessary); region values v_i and v_j are given to

11. Syntactic disambiguation based on left and right keywords is used to denote the common metadata attributes, e.g., left->antibody_target == right->antibody_target.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2016.2576447, IEEE/ACM Transactions on Computational Biology and Bioinformatics

such attributes. LEFT and RIGHT options of the region constructor have also two optional modifiers:

1) PROJECT_ creates the schema of regions r_{ij} as equal to the schema of r_i (in the case of PROJECT_LEFT) or of r_j (in the case of PROJECT_RIGHT).
2) _DISTINCT (i.e., LEFT_DISTINCT or RIGHT_DISTINCT), if two or more regions r_{ij} of a resulting sample have identical coordinates and also identical values for all their attributes, produces in output only one region of them.

PROJECT_ and _DISTINCT modifiers can also be used together (e.g., PROJECT_LEFT_DISTINCT), where _DISTINCT is particularly useful.

We next turn to genometric join predicates. They are based on the **genometric distance**, defined as the number of bases (i.e., nucleotides) between the closest opposite ends of two genomic regions, measured from the right end of the region with left end lower coordinate. Note that with the GDM choice of interbase coordinates[6], 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). Genometric join predicates are composed from the following simple predicates:

1) DISTANCE < <C> (or DISTANCE <= <C>), true if the distance of two regions is less (or less equal) than a given constant C.
2) MINDISTANCE, true for $< r_i, r_j >$ if r_j is the region of s_j at minimal distance from r_i (the left region); more than one region may be returned when several of them are at the same distance from r_i.
3) FIRST AFTER DISTANCE <C>, true if r_j is the closest region farther than C bases from the left region r_i; more than one region may be returned when several of them are at the same distance from r_i.

Note that the above predicates produce a small number of result regions, as they introduce thresholds either on the number or on the distance of regions which satisfy each predicate; the simple predicate DISTANCE > <C> can be also used, but only in conjunction with anyone of the above predicates. Variants of these clauses allow to specify different genometric conditions for the UPSTREAM and DOWNSTREAM directions of the DNA (i.e., the left and right side, respectively, of the left region r_i).

4.4.3 Example

The following GMQL program searches for those regions, of particular ChIP-seq experiments, called histone modifications (HM), that are at a minimal distance from transcription start sites of genes (TSS), provided that such distance is greater than 120K bases[12]. A sketch on small data is illustrated in Fig. 4(B); note that the result is projected on the second (RIGHT) operand, i.e. on HM.

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

12. This query can be used in the search for *enhancers*, i.e., parts of the genome which have an important role in gene activity regulation; the complete example, with a similar query, is in [8], Section 3.2.

4.5 MAP Operation

A MAP is a binary operation over two samples, respectively called **reference** and **experiment**. The operation is performed by first merging the samples in the reference operand, yielding to a single set of **reference regions**, and then by computing, for each sample in the experiment operand, aggregates over the values of the experiment regions that intersect with each reference region; we say that experiment regions are mapped to the reference regions.

4.5.1 Motivation

A MAP operation produces a regular structure, called **genometric space**, built as a matrix, where each experiment sample is associated with a column, each reference region with a row, and the matrix entries are typically scalars; such space can be inspected using heat maps, where rows and/or columns can be clustered to show patterns, or processed and evaluated through any matrix-based analytical process.

In general, a MAP operation allows a quantitative reading of experiments with respect to reference regions; when the biological function of the reference regions is not known, the MAP helps in extracting the most interesting reference regions out of many candidates.

4.5.2 Syntax and Semantics

The syntax of MAP is:

```
<S2> = MAP(<aggregates>) <R> <S1>
```

The semantics is as follows. Let $< r_k >$ be all the regions in all samples in R, obtained as result of the preliminary operation of merging in a single sample all the samples in the reference operand R. Let $s1_h$ be a sample of the second operand (S1), with $s1_h =<< r_i, v_i >, m_j >$ according to the GDM notation. From every sample $s1_h$ in S1, the operation builds a new sample $s2_h$ in S2 with the same metadata as $s1_h$ and with the regions $< r_k >$ as regions. The attributes of a region r_k of each new sample $s2_h$ are computed by means of aggregate expressions over the attributes of those regions r_i of $s1_h$ which intersect with the region r_k.

4.5.3 Example

In the example below, the MAP operation counts how many mutations occur in known genes, where the dataset EXP contains DNA mutation regions of multiple experiment samples and the dataset GENE contains the known gene regions; e.g., in the sketch in Fig. 4(C), on the first gene we count 6 mutations from the first experiment and 3 mutations from the second experiment.

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

5 Architecture

We developed a new software system for the execution of GMQL queries on big genomic data; its architecture is motivated by the need of bringing GMQL to cloud computing and make it usable. The overall software architecture is shown in Fig. 5. 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**.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2016.2576447, IEEE/ACM Transactions on Computational Biology and Bioinformatics

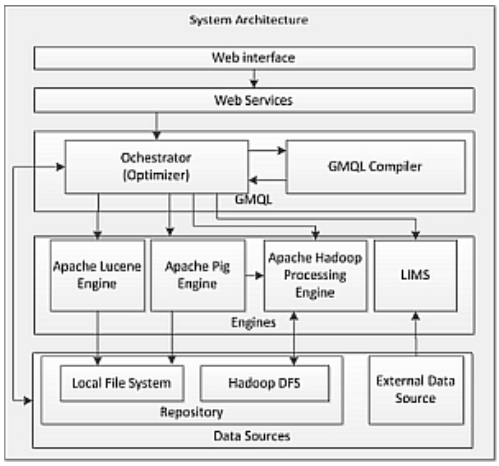

Fig. 5. Components of our software system.

5.1 Repository

The system repository has been designed with the objectives of providing transparency, privacy, ease of use and read-only access to public data; the user should not be aware of how files are internally managed, should register files with simple operations, should have a private space, and should have access to public data, managed by administrators.

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) [31], shared among all the computing nodes. Datasets are stored in the HDFS system, subdivided in metadata and region data. Optionally, a copy can be present in the LFS. Both the LFS and the HDFS store the control data, which are used to guide the processing. Control data include the schema for each dataset (encoded in Extensible Markup Language - XML) and the *Apache Lucene* [32] indexes for metadata. Moreover, both file systems have a public and a private space. Besides the genomic data, the LFS stores system-controlled information, encoded in XML, about the registered users, their security control and privileges, their saved queries and the location of their private resources[13].

All the datasets are stored in their original text format, as usually these files must be concurrently available to users for other computations. Only the datasets that are selected by a GMQL query are serialized by suitable adapters and translated to the internal binary GDM format on demand, when they are loaded in the engine before query execution; at that point, they are managed by Apache Pig under Hadoop. In this way, we do not replicate data in the native and GDM formats and we minimize data translations from native into GDM format.

5.2 Orchestrator

The orchestrator, written in Java programming language, controls the processing flow of the GMQL code, including compilation, data selection from the repository, scheduling of the efficient execution of Pig Latin code over the *Apache Pig* engine [9], and storing of the resulting datasets in the repository in standard format. The orchestrator has four components: the **Repository Manager**, for registering users and creating, deleting and changing datasets and their samples; the **Index Manager**, for creating and searching metadata indexes; the **GMQL Job Manager**, for launching the GMQL compiler, scheduling GMQL jobs and reporting about the status of GMQL jobs to users; and the **GMQL Job Optimizer**, for controlling the parallelization factors and choosing the version of Pig Latin translator.

When a user submits a GMQL query, the orchestrator uses the job manager to call the GMQL compiler, which produces the query translation into Pig Latin and the search criteria for loading the relevant samples from the repository. Then, the orchestrator uses the index manager to search the index and select the samples that comply to the search criteria, produces a list of the URIs of the samples to be loaded and invokes the job optimizer, which sets the execution parameters (such as the parallelization factors discussed in the next section); eventually, the orchestrator manages the outcome of the computation, including indexing of the result and storage in the user space. The system supports two types of execution, a *Local* mode and a *Map-Reduce* mode; the former one is suggested only for small data sizes, during the setup and debugging of GMQL programs.

The orchestrator can be invoked through different interfaces, including:

- Linux shell commands, each supported by suitable APIs, for managing the repository (adding users, adding/deleting datasets, and adding/deleting samples from existing datasets) and for compiling, running and tracking the execution of GMQL queries.
- RESTful web-services, which use the standard HTTP protocol and JSON files, thereby enabling the access to GMQL from within bioinformatics software and workflow engines, such as Galaxy [33].

6 Big Data Management Support

In this section we present the translation of GMQL into Pig Latin, which includes parallelism options; in the next section we present the implementation of domain-specific GMQL operations.

6.1 GMQL Compiler

Our developed translator has two components, the *lexer* and the *parser*. The former one scans the GMQL query and generates a list of tokens; the latter one identifies sub-sequences of the token list which correspond to grammar rules, using a LALR(1) algorithm [34]. When a statement is semantically valid, the compiler first infers the schema of the new introduced variable, then updates the internal state and finally emits the Apache Pig code that performs the requested operation. The internal state contains the name and schema of each variable which is either generated or mentioned in the query. We implemented the GMQL translator in Racket [35], a general-purpose functional programming language in the Lisp/Scheme family associated with a powerful set of tools; Racket has advanced macro system and higher order

13. In the system installation at IEO-IIT (https://www.ieo.it/en/ http://genomics.iit.it/), a center of excellence in oncology research, we connected the repository to a Laboratory Information Management System (LIMS) designed for storing both the raw data after NGS and the workflows for producing processed data into the HDFS [30].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2016.2576447, IEEE/ACM Transactions on Computational Biology and Bioinformatics

functions, which facilitate the production of a concise, clean and safe code. Fig. 6 shows the translation of the JOIN in the example at the end of subsection 4.4.3.

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

Fig. 6. Translation of a GMQL JOIN into Pig Latin.

Datasets are loaded into suitable Apache Pig variables; we use the format illustrated in Fig. 2, where each dataset is mapped into two bags, named V_region.dat and V_meta.dat. In Fig. 6, lines 1-10 are concerned with metadata and produce the data bag RES_meta. The metadata of the two operands are first grouped by sample and then the cross product of samples is generated and flattened; each pair in the cross product is associated with a new sample having as metadata the union of the metadata of the two operands.

Lines 11-20 work on regions. We encoded in Java programming language a fast-join algorithm which searches for matching regions at minimal and bound distance (discussed in Section 7.2). First, samples are grouped and paired (as in the case of metadata). Then, the RES_joiner class is defined as result of invoking the MinDist+GreaterThan Java code; the defined function is invoked on each pair of samples and the result is finally flattened. The linking of metadata and regions of each output sample is guaranteed by the use of the same hash function on the two ids of the input pairs, at lines 8 and 10 for the metadata and within the Join function for the regions.

6.2 Parallelism in the Generated Code

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

- Use of **by-pair parallelism**, generated when an operations can be split into independent computations over pairs of samples.
- Use of **by-chrom parallelism**, which is generated by partitioning the GROUP and CROSS Apache Pig operations by chromosome. This is a classic *distributed join*; as result, regions are only produced from input regions with matching chromosomes[14].
- Use of suitable **parallelism directives** in Apache Pig operations, so as to improve the performance (e.g., setting the number of reducers as a function of the size of the input bags).

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

6.2.1 Effect of By-Chrom Parallelism

In Fig. 7 we show the effect of adding by-chrom parallelism to the by-pair parallelism of the JOIN operation of three reference samples over an increasing number of experiment samples. The figure shows an important reduction of processing time with the addition of by-chrom parallelism, which depends both on the increased parallelism and reduced data sizes of operands (we also observed much smaller intermediate data sizes); moreover, all algorithms shown in Section 7 need the ordering of regions along the genome, and the time of ordering is also reduced with smaller data sizes. We found similar results in other operations and hence generally adopted the by-chrom parallelism together with the by-pair parallelism.

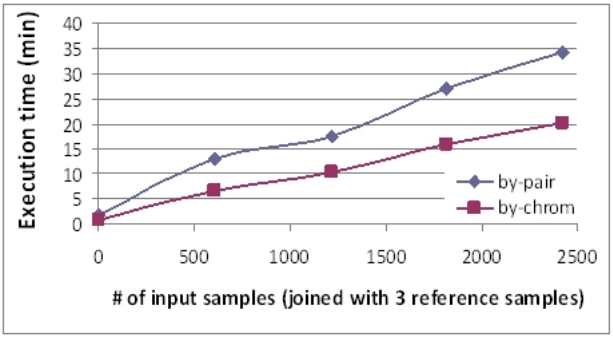

Fig. 7. Performance of by-chrom parallelism added to by-pair parallelism.

6.2.2 Size-Specific Tuning

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

We then considered the behavior of Apache Pig operations with small input sizes. We noted that operations over metadata are less demanding and operations over regions are more demanding. Thus, after several experiments, we empirically produced a simple rule: if the size of inputs of an Apache Pig operation is above 2 GB, we simply set the

14. This parallelism is produced by changing, in Fig. 6, lines 11 and 12, where GROUP has to be applied by using also the chromosome, and lines 13 and 14, where CROSS is turned into a join on the chromosome.

reduce threshold to 0.25 GB; if it is below 2 GB, we assign to the Apache Pig operation a fixed number of reducers, equal to 1 if the operation applies to metadata and to 8 if the operation applies to regions, where 8 is the number of reduce slots available on our server. We tested this rule on many complex queries and obtained a performance gain between 10% and 20%[15].

7 Optimization of Selections and Joins

In this section, we briefly describe the optimization of the Selection operation, then we focus on the Join operation optimization using the `Distance` and `MinDistance` algorithms, with their several variants; Map operation is a variant of Distance Join. Due to space limitations, we omit to discuss the Cover operation optimization.

7.1 Selection

All GMQL queries include an initial `SELECT` statement that typically extracts few samples (i.e., only the ones needed to answer a query) out of all the datasets of large repositories. Thus, we use Apache Lucene indexing for metadata; just the matching samples of each dataset are loaded in GDM format. Lucene allows for arbitrary Boolean expressions over attribute values; its processing time (order of milliseconds) is negligible with respect to execution times on the cloud. Prior to building the index, text of metadata attributes is cleaned from control or reserved characters; the same processing is applied to query constants.

7.2 Join

In the case of `JOIN` and `MAP`, our initial GMQL translator was using the `CROSS` operation of Pig Latin for first building the cross product of the regions in the two input variables, and then computing the join predicates (for `JOIN`), or aggregate functions (for `MAP`). In spite of parallelism, using `CROSS` is too heavy, as it scales with the product of the number of regions (as shown by Fig. 10 and Fig. 11). A better approach leverages the fact that regions are aligned to the same reference. We devised a family of smart join and map algorithms, that apply to pairs of samples and compare much fewer regions by carefully scanning the genome; they are implemented as user-defined functions in Java programming language and embedded in the form of reducers within the generated Pig Latin code, parallelized by chromosome as discussed in the previous section.

7.2.1 Distance Join

The `DistanceJoin` algorithm produces pairs of regions which are at a distance below a given threshold and serves as basis for discussing all the algorithms of this section. It accepts as input the two samples `R` (reference) and `E` (experiment), the maximum distance threshold `m` and the region construction option `con`; it produces as output the list `O` of result regions, assembled according to the region construction option. This algorithm adds genomic-specific features to the classic sort-merge algorithms for temporal intersection joins [36], such as the separation of *upstream* and *downstream* regions (with respect to the left and right side of a reference region, respectively) and richer distance constraints. The main difference with respect to [36] is that our algorithm does not maintain pointers into data sequences, which are disallowed in our computational environment; instead, it carefully manages a cache of regions of the experiment, including those regions that could still be joined with the current reference.

Distances may distinguish the *upstream* and *downstream* directions of the genome[5]; the predicate `DistTest` receives as input two regions (r_1, r_2) and two thresholds on the upstream (`u`) and downstream (`d`) directions of the genome, respectively. `DistTest` is `true` when regions satisfy the thresholds. A simple utility function `out(r1,r2,con)` builds the output region out of two matching regions, according to the option `LEFT`, `RIGHT`, `INT`, or `CAT` for the `con` parameter, described in Section 4.4.2.

We first discuss the `DistanceJoin` algorithm with symmetric distance thresholds. The algorithm is composed of one external loop on each region of the reference `R` and two internal loops, one to analyze cached regions from the previous iteration and one to advance the scan on the regions of the experiment `E` and fill the cache `C`; on the first iteration, the first of these internal loops is not executed, as `C` is empty. In the pseudo-code below, `length` returns the length of a list, `<-` adds an element to a list, and `EOF(L[i])` is `true` when `L[i-1]` is the the last element of the list `L`.

```
1   DistanceJoin(R,E,m,con)
2   { O := []; // output
3     C := []; // cache list
4     j := 0;  // ranges on E
5     for i := 0...R.length-1
6     { T := []; // temp list
7       for k := 0...C.length-1
8       { if (DistTest(R[i],C[k],m,m))
9         { O <- out(R[i],C[k],con);
10          T <- C[k]; }
11        else if (R[i].right < C[k].left)
12        { T <- C[k]; }}
13      C := T;
14      while (NOT EOF(E[j]) AND
15             (R[i].right + m > E[j].left))
16      { if (DistTest(R[i],E[j],m,m))
17        { O <- out(R[i],E[j],con);
18          C <- E[j]; }
19        j++; }}
20    return O; }
```

Note that the Map algorithm is a variant of the `DistanceJoin` algorithm with $m = -1$; it consists of collecting, for each reference region, the bag of experiment regions which intersect the reference region (i.e., which are at negative distance from it) and then applying aggregate functions to the values of that bag of regions.

The general case of the distal join includes as input two distinct thresholds for the upstream (`u`) and downstream (`d`) directions of the genome[5], respectively. The input parameters of the modified algorithm include the two thresholds, and the code at its lines 8-10 is substituted by the following code, where `m` denotes the maximum value between `u` and `d`:

15. In Example 3.2 in [8], which includes a cascade of 4 GMQL joins and is translated into 32 Apache Pig operations that use reducers, we obtained a reduction of execution time of 17%, from 633 to 525 seconds.

```
if DistTest(R[i],C[k],m,m)
{ if (DistTest(R[i],C[k],d,u)
  AND (R[i].strand = '+' OR R[i].strand = '*')
  AND (C[k].strand = '+' OR C[k].strand = '*'))
  { O <- out(R[i],C[k],con); }
  else if (DistTest(R[i],C[k],u,d)
  AND (R[i].strand = '-' OR R[i].strand = '*')
  AND (C[k].strand = '-' OR C[k].strand = '*'))
  { O <- out(R[i],C[k],con); }
  T <- C[k]; }
```

Note that a first invocation of `DistTest` (controlling the caching mechanism) is with `m` for both the third and fourth parameter, and then two cases are considered, first when both strands (one of the sample `R` and one of the matching sample `E`) are either positive or undefined, and then when both such strands are negative or undefined; in these subsequent `DistTest` invocations, the two parameters are swapped. A similar substitution is required for the code at lines 16-18.

7.2.2 MinDistance Join

Also the `MinDistanceJoin` algorithm has an external iteration on the reference sample `R` and an internal iteration on the experiment sample `E`, also subdivided into two parts, respectively used for emptying and filling the cache `C`. Cache management is more complex, as it requires two temporary lists `T` and `B`. The latter one includes the current best regions; it is reassigned whenever the best distance is strictly improved, while more regions with distance tie are appended to it. The algorithm uses the function `Dist(r1,r2)` that computes the distance between regions, which is 0 if two regions are adjacent or -1 if they overlap.

```
1   MinDistanceJoin(R,E,con)
2   { O := []; // output
3     C := []; // cache list
4     j := 0;  // ranges on E
5     for i := 0...R.length-1
6     { T := []; // temp list
7       B := []; // best distance list
8       m := MAX-CHROM-DIMENSION;
9       for k := 0...C.length-1
10      { d := dist(R[i],C[k]);
11        if (d < m)
12        { m := d;
13          if (C[k].left < R[i].left)
14          { T := []; }
15          B := out(R[i],C[k],con); }
16        else if (m = d)
17        { B <- out(R[i],C[k],con); }
18        T <- C[k]; }
19      C := T;
20      while (NOT EOF(E[j]) AND
21             (R[i].right + m >= E[j].left))
22      { d := Dist(R[i],E[j]);
23        if (d < m)
24        { m := d;
25          B := out(R[i],E[j],con);
26          if (E[j].left < R[i].left)
27          { C := []; }}
28        else if (d = m)
29        { B <- out(R[i],E[j],con); }
30        C <- E[j];
31        j++; }
32      O <- B; }
33    return O; }
```

Variants of the above pseudo-code allow to add minimum or maximum distance thresholds, or to extract top k matches.

8 Evaluation

In this section, we compare our system performance with the equivalent state-of-the-art and then we show our linear scale-up. Finally, we present a full example with biological interpretation. We measured performances both on Amazon Web Service cloud, taking advantage of multiple nodes (see Section 8.2), and on our server, an Intel® Xeon® Processor with CPU E5-2650 at 2.00 GHz, six cores, RAM of 128 GB, hard disk of 4x2 TB, and the engines Apache Hadoop 2.6.2, Apache Pig 0.15.0 and Apache Lucene 5.3.1.

8.1 Comparison with the State of the Art

No cloud computing system operates on region-based processed data, but BEDTools [21] and BEDOPS [22] are popular biologists' tools for scripting region-based computations which perform set-oriented operations upon regions; hence, we consider them the closest tools for a state-of-the-art comparison. They provide single machine code that uses multithreading for some computationally expensive operations; they do not support implicit iteration over data samples or metadata management.

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

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

is encoded by the following BEDOPS program:[16]

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

Fig. 8 shows comparative performances of the `DISTANCE JOIN` operation between three reference samples of about 45K regions each and an increasing number of experiment samples with an average of 50K regions and 7.5 MB size each. GMQL performs worse than BEDOPS and BEDTools when experiment samples are less than 50, but it outperforms them above such threshold.

Fig. 9 shows comparative performances of the `MAP` operation with a `count` aggregate function in the same experimental setting, but with a single reference sample; in

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCBB.2016.2576447, IEEE/ACM Transactions on Computational Biology and Bioinformatics

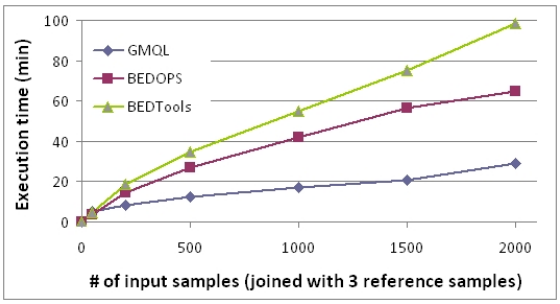

Fig. 8. Performance of the DISTANCE JOIN operation with increasing number of samples; GMQL vs. BEDOPS vs. BEDTools.

this case, GMQL performs worse when experiment samples are less than 500, but it outperforms both BEDOPS and BEDTools above such threshold. Furthermore, GMQL time does not depend on the complexity of operation (e.g., number of computed aggregates), but rather to the need of distributing sample files, partitioned over the number of chromosomes, to the computing nodes. Instead, BEDOPS requires an increase of 30% of execution time for computing two aggregates, and BEDTools does not support multiple aggregates.

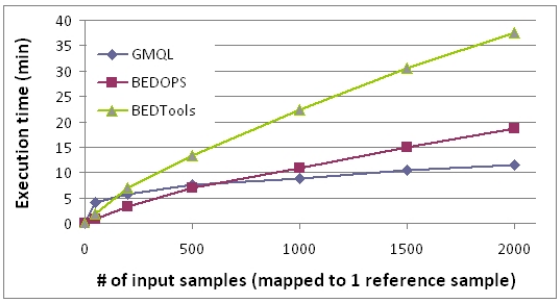

Fig. 9. Performance of the MAP operation with increasing number of samples; GMQL vs. BEDOPS vs. BEDTools.

We conclude that BEDOPS has comparatively better performance than BEDTools, as independently reported in [20], and that beyond given thresholds GMQL performs faster on big data; moreover, GMQL provides a cloud computing solution, whose performance will increase with better operating system, better computing infrastructures and larger clouds.

8.2 GMQL Scaling with Big Datasets

Fig. 10 illustrates the performance of the three kinds of JOIN (DISTANCE, MINDISTANCE and FIRST AFTER DISTANCE) described in Section 4.4.2, when executed with 3 samples of about 45K regions each as fixed references and a growing number of samples (up to 2.5K) as experiment samples; these samples have a variable number of regions and sizes, with an average of 50K regions and 7.5 MB size each. The diagram shows almost linear scaling; it also shows that the encoding of the JOIN as crossproduct (CROSS) has much worse performance.

MAP is a simple case of DISTANCE JOIN; hence, its performance curves are similar. Fig. 11 shows the MAP

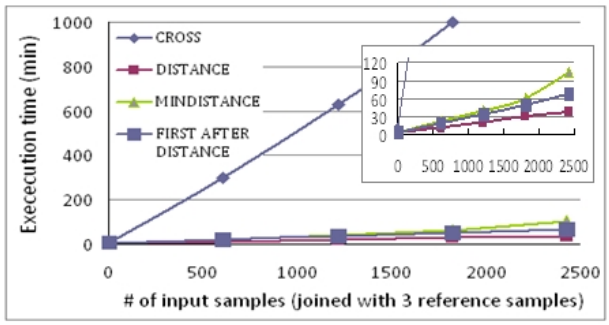

Fig. 10. JOIN performance over big data.

performance when the set of all known human genes (both protein coding and not) is used as single reference sample; note the linear scale up to only 15 minutes with 2500 experiment samples (77,778,000 regions) and 45K genes.

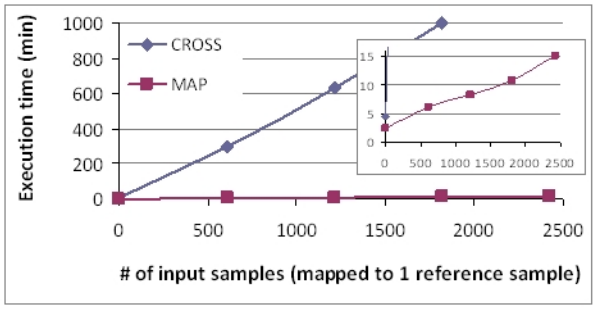

Fig. 11. MAP performance over big data.

We also used the Hadoop framework provided by Amazon Web Services[17] with m3.2xLarge model, 8 CPU, 30 GB RAM, 2x80 GB storage SSD to test the MAP operation with over than 4000 samples, our largest dataset (about 31.1 GB); parallelism was set to 1 master node and 5, 10, and 15 slave nodes, respectively. Table 1 shows a significant reduction in execution time with the increase of the number of nodes, although scalability is lower when going from 10 to 15 nodes, most likely due to higher communication overhead.

TABLE 1
Scalability of MAP execution time by increasing parallelism in the Amazon Web Service cloud

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

8.3 Full Biological Example

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

17. http://aws.amazon.com/

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

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

The `RES` result variable includes both regions and metadata; the former ones indicate interesting promoter regions (that can be further inspected using viewers, e.g., genome browsers), the latter ones allow tracing provenance of resulting samples. Fig. 12 shows 4 metadata attributes of the resulting samples: the `order` of the sample, the `antibody` and `cell` type considered in the ChIP-seq experiment, and the promoter region `count`.

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. 12. Metadata excerpt of the resulting samples.

Further biological use case examples were thoroughly illustrated and discussed previously in [8].

9 Conclusions

In this paper, we thoroughly discussed and motivated our approach to genomic data modeling and querying, formalized GMQL operations, and illustrated GMQL implementation over big genomic data, evaluating its performance. The **main contribution of our new system** is the ability of querying thousands or even millions of processed experimental samples, which will soon become available [37]; indeed they are growing at an extremely fast speed, and are well curated and published online by large consortia. Data complexity is manifesting itself not only in its sheer size, but also in the heterogeneity and large number of underlying samples, conditions, etc. GDM provides interoperability across tens of processed data formats, while GMQL supports high-level query processing through a combination of relational and domain-specific region management operations, tailored to the needs of genomic data management; it brings to biologists well-established (by Edgar F. Codd [38]) database concepts. Our choice of deploying GMQL in a cloud computing environment, rather than using a conventional or scientific DBMS (such as *Vertica* [28] or *SciDB* [29]), is also motivated by our vision of moving our project soon into Apache-incubation, so as to generate a community of users and developers, and also benefit from the existence of complementary frameworks (e.g., *Adam* [39] supporting primary and secondary genomic data analysis and *Apache Mahout* [40] supporting machine learning.)

We deployed two installations of our implemented software system, one at Politecnico di Milano for system development and one at IEO-IIT[18], a center of excellence in cancer research, for conducting joint research projects[19]. Our system can be freely downloaded from: http://www.bioinformatics.deib.polimi.it/GMQL/ and can be tested through a web application freely available at http://www.bioinformatics.deib.polimi.it/GMQL/queries/, which provides a set of predefined parametric GMQL queries on our system repository. We are currently working towards new releases of the system which will use Apache Spark [27] and Apache Flink [41] engines under the Apache Hadoop YARN [10] framework.

Acknowledgments

We acknowledge the essential contributions of Heiko Muller (IIT) and of Gianpaolo Cugola, Matteo Matteucci, Fernando Palluzzi and Vahid Jalili (PoliMI). Research was supported by the Data-Driven Genomic Computing (GenData 2020) PRIN project (2013-2015), funded by the Italian Ministry of the University and Research, and by a grant from Amazon Web Services.

References

[1] J. Shendure, and H. Ji, "Next-generation DNA sequencing," *Nat. Biotechnol.*, vol. 26, no. 10, pp. 1135-1145, 2008.
[2] S. C. Schuster, "Next-generation sequencing transforms today's biology," *Nat. Methods.*, vol. 5, no. 1, pp. 16-18, 2008.
[3] NIH National Human Genome Research Institute, "DNA Sequencing Costs." http://www.genome.gov/sequencingcosts/
[4] ENCODE Project Consortium, "An integrated encyclopedia of DNA elements in the human genome," *Nature*, vol. 489, no. 7414, pp. 57-74, 2012.
[5] Cancer Genome Atlas Research Network, J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, and J. M. Stuart, "The Cancer Genome Atlas pan-cancer analysis project," *Nat. Genet.*, vol. 45, no. 10, pp. 1113-1120, 2013.
[6] 1000 Genomes Project Consortium, G. R. Abecasis, D. Altshuler, A. Auton, L. D. Brooks, R. M. Durbin, R. A. Gibbs, M. E. Hurles, and G. A. McVean, "A map of human genome variation from population-scale sequencing," *Nature*, vol. 467, no. 7319, pp. 1061-1073, 2010.
[7] C. E. Romanoski, C. K. Glass, H. G. Stunnenberg, L. Wilson, and G. Almouzni, "Epigenomics: Roadmap for regulation," *Nature*, vol. 518, no. 7539, pp. 314-316, 2015.
[8] M. Masseroli, P. Pinoli, F. Venco, A. Kaitoua, V. Jalili, F. Paluzzi, H. Muller, and S. Ceri, "GenoMetric Query Language: A novel approach to large-scale genomic data management," *Bioinformatics*, vol. 12, no. 4, pp. 837-843, 2015.
[9] Apache Pig. http://pig.apache.org/
[10] Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[11] L. D. Stein, "The case for cloud computing in genome informatics," *Genome Biol.*, vol. 11, no. 5, pp. 207, 2010.

18. https://www.ieo.it/en/ - http://genomics.iit.it/
19. Most relevant on going projects regard: Study of 3D chromatin structure, DNA replication and gene expression intertwinement, Chromatin state changes in time-course Myc binding, and Co-occurrence of TEAD and other transcription factor binding sites.

[12] Global Alliance Genomics API. http://ga4gh.org/#/documentation
[13] Google Genomics Could Platform. https://cloud.google.com/genomics/
[14] U. Röhm and J. Blakeley, "Data management for high-throughput genomics," in *Proc. CDIR*, 2009, pp. 1-10.
[15] S. Tata, J. M. Patel, J. S. Friedman, and A. Swaroop, "Declarative Querying for Biological Sequences," in *Proc. IEEE ICDE*, 2006, pp. 87-99.
[16] S. Tata, W. Lang, and J. M. Patel, "Periscope/SQL: Interactive exploration of biological sequence databases," in *Proc. VLDB*, 2007, pp. 1406-1409.
[17] V. Bafna, A. Deutsch, A. Heiberg, C. Kozanitis, L. Ohno-Machado, and G. Varghese, "Abstractions for genomics,"*Commun. ACM*, vol. 56, no. 1, pp. 83-93, 2013.
[18] C. Kozanitis, A. Heiberg, G. Varghese, and V. Bafna, "Using Genome Query Language to uncover genetic variation," *Bioinformatics*, vol. 30, no. 1, pp. 1-8, 2014.
[19] M. Cereda, M. Sironi, M. Cavalleri, and U. Pozzoli, "GeCo++: a C++ library for genomic features computation and annotation in the presence of variants," *Bioinformatics*, vol. 27, no. 9, pp. 1313-1315, 2011.
[20] K. Ovaska, L. Lyly, B. Sahu, O. A. Jnne, and S. Hautaniemi, "Genomic Region Operation Kit for flexible processing of deep sequencing data," *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 10, no. 1, pp. 200-206, 2013.
[21] A. R. Quinlan and I. M. Hall, "BEDTools: a flexible suite of utilities for comparing genomic features," *Bioinformatics*, vol. 26, no. 6, pp. 841-842, 2010.
[22] S. Neph, M. S. Kuehn, A. P. Reynolds, E. Haugen, R. E. Thurman, A. K. Johnson, E. Rynes, M. T. Maurano, J. Vierstra, S. Thomas, R. Sandstrom, R. Humbert, and J. A. Stamatoyannopoulos, "BEDOPS: high-performance genomic feature operations," *Bioinformatics*, vol. 28, no. 14, pp. 1919-1920, 2012.
[23] H. Nordberg, K. Bhatia, K. Wang, and Z. Wang, "BioPig: a Hadoop-based analytic toolkit for large-scale sequence data," *Bioinformatics*, vol. 29, no. 23, pp. 3014-3019, 2013.
[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A not-so-foreign language for data processing," in *ACM-SIGMOD*, 2008, pp. 1099-1110.
[25] A. Schumacher, L. Pireddu, M. Niemenmaa, A. Kallio, E. Korpelainen, G. Zanetti, and K. Heljanko, "SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop," *Bioinformatics*, vol. 30, no. 1, pp. 119-120, 2014.
[26] M. S. Weiwiorka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski, "SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision," *Bioinformatics*, vol. 30, no. 18, pp. 2652-2653, 2014.
[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX*, 2012, pp. 15-28.
[28] Vertica. https://www.vertica.com/
[29] SciDB. http://www.scidb.org/
[30] F. Venco, Y. Vaskin, A. Ceol, and H. Muller, "SMITH: a LIMS for handling next-generation sequencing workflows," *BMC bioinformatics*, vol. 15, no. Suppl 14, pp. S3, 2014.
[31] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. MSST*, 2010, pp. 1-10.
[32] Apache Lucene. http://lucene.apache.org/
[33] Galaxy. http://galaxyproject.org/
[34] F. DeRemer and T. Pennello, "Efficient computation of LALR(1) Look-Ahead sets," *ACM Trans. Prog. Lang, Syst.*, vol. 4, no. 4, pp. 615-649, 1982.
[35] Racket. http://racket-lang.org/
[36] H. Gunadhi and A. Segev, "Query processing algorithms for temporal intersection joins," in *Proc. IEEE ICDE*, 1991, pp. 336-344.
[37] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: astronomical or genomical?" *PLoS Biol.*, vol. 13, no. 7, pp. e1002195, 2015.
[38] E. F. Codd, "A relational model of data for large shared data banks," *Comm. ACM*, vol. 13, no. 6, pp. 377-387, 1970.
[39] Adam. http://www.bdgenomics.org/
[40] Apache Mahout. http://mahout.apache.org/
[41] Apache Flink. http://flink.apache.org/

Stefano Ceri is Professor at the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano. He obtained his Dr. Eng. Degree from Politecnico di Milano in July 1978. He was Visiting Professor at the Computer Science Department of Stanford University (1983-1990), Chairman of the Computer Science Section of DEI (1992-2004), Director of Alta Scuola Politecnica (ASP) of Politecnico di Milano and Politecnico di Torino (2010-2013). He has been awarded two advanced ERC Grants on Search Computing (2008-2013) and Genomic Computing (2016-2021). He is currently leading the PRIN project GenData 2020. He is the recipient of the ACM-SIGMOD "Edward T. Codd Innovation Award" (2013), and an ACM Fellow and member of the Academia Europaea.

Abdulrahman Kaitoua received his B.E. degree with high distinction in Computer Systems Engineering from Mamoun Private University (MUST), Syria, in 2009. He received his Masters in Electrical and Computer Engineering department from the American University of Beirut (AUB), Lebanon, in 2013. He is currently a Ph.D. student in Information Technology at Politecnico di Milano University, Italy. His research interests include bioinformatics, cloud computing, software engineering, data mining, distributed computing, and big data processing.

Marco Masseroli received the Laurea Degree in Electronic Engineering in 1990 from Politecnico di Milano, Italy, and a PhD in Biomedical Engineering in 1996, from Universidad de Granada, Spain. He is Associate Professor at the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano, Italy. His research interests are in the area of bioinformatics and biomedical informatics, focused on distributed Internet technologies, biomolecular databases, controlled biomedical terminologies and bio-ontologies to effectively retrieve, manage, analyze, and semantically integrate genomic information with patient clinical and high-throughout genomic data. He is the author of more than 180 scientific articles in international journals, books and conference proceedings.

Pietro Pinoli received the BS/MS degree in Computer Science and Engineering in 2012 from Politecnico di Milano, Italy. He is currently a PhD candidate at the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano, Italy, and a member of the DEIB Bioinformatics Group. He is a collaborator of the Istituto Europeo di Oncologia (IEO). His research interests include databases, big data, machine learning and bioinformatics.

Francesco Venco graduated at University of Padova with a thesis on Bayesian Network automatic learning and received a PhD in Information Technology from Politecnico di Milano, Italy. During the last years he worked with the group of Bioinformatics of Politecnico di Milano on techniques to manage large NGS (Next Generation Sequencing) data, in collaboration with IEO research center. His main focus is on efficient algorithms to manage big genomic data.