GenoMetric Query Language (GMQL) Tutorial

Genomic Computing Group

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

November 9, 2015

Contents

1 Introduction

The following tutorial is divided in two parts:

a. how to register, create and manage datasets and data samples for their GMQL use and deal with supported or custom data formats;

b. how to build a GMQL query from a genomic problem *step-by-step*, describing the formulation and execution of its operations.

Note 1.1. *Please refer to the QuickStart documentation for system installation and to the GMQL complete documentation for further information about the data model and query language.*

2 Creation and Management of Datasets and Data Samples

In order to use GMQL it is necessary to register datasets of the data sample files that need to be processed, which may have different formats. In GMQL, many data formats are natively supported; custom formats, which require the specification of an XML file describing the structure (i.e. schema) of the genomic region data contained in the corresponding textual tab-delimited column data files, can be defined as well (see below GTF and tab format description for details).

Note 2.1. *Columns with genomic region coordinates (i.e. chromosome, start, stop, strand) must be present in the data file at the position defined by the format (e.g. chromosome, start and stop are often the first three columns in this order); thus, they must not be specified in the XML data file schema.*

2.1 Supported Formats

- Bed Format

  The bed format is described at:
  `http://genome.ucsc.edu/FAQ/FAQformat.html#format1`
  Files in this format are referenced with "bed" both for type and extension. We consider only the first six columns of the format, which orderly are: Chrom (string), ChromStart (integer), ChromEnd (integer), Name (string), Score (integer), Strand (string). Only the first three columns are mandatory; if missing, the Name and Score will be set to null (i.e. empty), while the strand will be set to undefined (i.e. "*"). The bed format adopts the 0-base notation for the genomic region coordinates, i.e. coordinate counting starts from 0.

- ENCODE Broad Peak Format

  The broad peak format is described at:
  `http://genome.ucsc.edu/FAQ/FAQformat.html#format13`

Files in this format are referenced with "broadPeak" both for type and extension. The columns of the format orderly are: Chrom (string), ChromStart (integer), ChromEnd (integer), Name (string), Score (integer), Strand (string), signalValue (float), pValue (float). qValue (float). The broad peak format adopts the 0-base notation for the genomic region coordinates, i.e. coordinate counting starts from 0.

- ENCODE Narrow Peak Format

  The narrow peak format is described at:
  `http://genome.ucsc.edu/FAQ/FAQformat.html#format12`
  Files in this format are referenced with "narrowPeak" both for type and extension. The columns of the format orderly are: Chrom (string), ChromStart (integer), ChromEnd (integer), Name (string), Score (integer), Strand (string), signalValue (float), pValue (float). qValue (float), Peak (integer). The narrow peak format adopts the 0-base notation for the genomic region coordinates, i.e. coordinate counting starts from 0.

- GTF Format

  The GTF format is described at:
  `http://genome.ucsc.edu/FAQ/FAQformat.html#format4`
  Files in this format are referenced with "gtf" both for type and extension. The first eight columns are fixed; the ninth column has a variable part consisting of attribute-value pairs; the user should then indicate how these attributes have to be included in the schema in their exact order. The first eight columns of the format orderly are: Seqname (it corresponds to the chromosome; string), Source (string), Feature (string), Start (integer), End (integer), Score (integer), Strand (string), Frame (integer).

  For example, let geneId and transcriptId be attribute-values in column nine for a set of GTF files; the required schema should be:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<gmqlSchemaCollection name="GLOBAL_SCHEMAS"
     xmlns="http://www.bioinformatics.deib.polimi.it/GMQL/">
  <gmqlSchema name="GTF_EXAMPLE" type="gtf">
    <field type="STRING">source</field>
    <field type="STRING">feature</field>
    <field type="INTEGER">score</field>
    <field type="INTEGER">frame</field>
    <field type="STRING">geneId</field>
    <field type="STRING">transcriptId</field>
  </gmqlSchema>
</gmqlSchemaCollection>
```

  Note that, differently from other formats, the GTF format adopts the 1-base notation for the genomic region coordinates, i.e. coordinate counting

starts from 1, while our system starts counting region coordinates from 0. Conversion is done automatically by the system.

- VCF Format

  The Variant Call Format is described at:
  `http://www.1000genomes.org/node/101`
  Files in this format are referenced with "vcf" both for type and extension. The columns of the format are nine in the following order: Chrom (string), Position (integer), Id (string), Ref (string), Alt (string), Quality (integer), Filter (string), Info (string), Format (string(s)). The VCF files are quite complex; we provide a simple support for them by considering all the first eight columns only. For simplicity, we set the stop position equal to the start position for all single base mutations. Note that the VCF adopts the 1-base notation for the genomic region coordinates, while our system starts counting region coordinates from 0. Conversion is done automatically by the system.

- General Tab Format

  This format regards bed-like files composed of a mandatory part and a custom number of columns.
  Files in this format are referenced with "tab" both for type and extension. The mandatory part includes three columns: Chrom (string), ChromStart (integer), ChromEnd (integer); all the others are user-defined. The position of the strand column is not fixed; if present, using the name "strand", it is possible to specify which column contains such information. For example, supposing that after the three coordinate columns a file contains two strings regarding, for instance, gene ID and transcript ID, the file schema should be:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<gmqlSchemaCollection name="GLOBAL_SCHEMAS"
     xmlns="http://www.bioinformatics.deib.polimi.it/GMQL/">
  <gmqlSchema name="TAB_EXAMPLE" type="tab">
    <field type="STRING">geneId</field>
    <field type="STRING">transcriptId</field>
  </gmqlSchema>
</gmqlSchemaCollection>
```

  Instead, if the fourth column contains the strand string and is followed by a column with float score value:

```
...
  <gmqlSchema name="TAB_EXAMPLE" type="tab">
    <field type="STRING">strand</field>
    <field type="FLOAT">scoreValue</field>
  </gmqlSchema>
...
```

2.2 Sample Metadata

Metadata are attributes associated with a whole sample, useful to filter and associate samples in and among datasets. The metadata file must be in the same directory of and named exactly as its companion data file; the extension of this file must be composed of two parts: the data file extension plus the additional suffix *.meta*.

Example. *Suppose that we want to use in our system the data file myBed.bed. The (required) companion metadata file must then be named myBed.bed.meta.*

Note 2.2. *Distinct data sample files belonging to one dataset must not necessarily stay in the same directory; furthermore, a sample file can belong to multiple datasets.*

Metadata files must be simple text files organized as follows:

- Each line of a metadata file must contain an attribute-value pair only.
- The attribute and its value must be separated by a tab.

For example:

```
cell        H1-hESC
antibody    AP-2alpha
...
```

2.3 Dataset Creation

Once at least one data file has been prepared with a companion metadata file, a dataset can be created by using the command:

```
repositoryManagerV1 CreateDS <DataSetName> <Schema> <FileURLs>
```

This command registers a new dataset for GMQL processing. More precisely, the required parameters are:

- *DataSetName*: used as a reference to the dataset, both in data management and query language commands.
- *Schema*: data schema to be used for the dataset. If the data schema to be used is one of the supported standard "fixed" formats (i.e. bed, broadPeak, narrowPeak), then it is enough to specify only the keyword that references the schema type (e.g. BED, BROADPEAK, NARROWPEAK). Otherwise, it is necessary to specify the full path of a user-defined schema file. Note that the system does not perform any check on the schema when loading data.
- *FileURLs*: complete path of each data file composing the dataset. Note that metadata file must not be listed; they are automatically identified by name from their companion data file name. If any metadata file is not

found, the system notifies an error. Multiple URLs can be specified and they must be comma separated. It is also possible to specify the complete path of a directory instead; in this case all files in the directory are added to the dataset.

In summary, when this command is executed, the operations performed by the system are:

a. Create a single metadata file in the repository for the entire dataset.

b. Index the single metadata file.

c. Add the URLs listed for the dataset files to the user $< DataSetName >$ .XML file that the system creates.

d. If the system is set as HDFS (i.e. to work using an Hadoop Distributed File System), then the files are copied to the HDFS.

e. If the system is set as LOCAL MODE (i.e. to work using local computing resources), then the files are kept where they are and their URLs are stored.

2.4 Listing Datasets and Data Samples

After creating and registering a dataset, it is possible to check its presence in the system, by using the command:

```
repositoryManagerV1 List all
```

It returns the list of all datasets present in the system, for example:

```
MYDATASET1
OTHERDATA
BED_EXPERIMENT1
```

It is also possible to list the specific data sample files that compose a dataset, by using the command:

```
repositoryManagerV1 List <DataSetName>
```

The full paths of the sample data files are listed, for example:

```
\home\userName\myExperiment\experiment1.bed
\home\userName\myExperiment\experiment2.bed
\home\userName\secondTest\otherExperiment.bed
```

2.5 Retrieving GMQL Generated Data

Regardless the GMQL running mode (i.e. Local or MapReduce), all data stored during a GMQL query processing are included in the GMQL repository. To access them, first they must be copied to a user local folder, by using the command:

```
repositoryManagerV1 CopyDSToLocal <DatasetName> <DestinationLocalFolder>
```

2.6 Other Operations

It is possible to add sample files to an existing dataset, by using the command:

```
repositoryManagerV1 AddSample <DataSetName> <FileURLs>
```

It is also possible to delete an entire dataset, or specific sample files, by respectively using the commands:

```
repositoryManagerV1 DeleteDS <DataSetName>
repositoryManagerV1 DeleteSample <DataSetName> <FileURLs>
```

Note 2.3. *Both operations do not delete the actual data files on the local file system, but only their created references and copies on the Hadoop distributed file system, if any. This happens both in local and HDFS mode.*

3 Creation and Execution of GMQL Queries

3.1 General Tips

In the following sections we illustrate and discuss how to write queries (i.e. programs) in GMQL. A few introductory general tips can be useful:

- **Unique names**. Each dataset must have a unique name within a GMQL program. Note that datasets registered in the GMQL system (see Section 2.3) are considered already defined at the beginning of each GMQL program; thus, they do not need to be defined in a GMQL program.
- **Always end with semicolon**. A common error is to forget placing a semicolon at the end of each GMQL command.
- **Beware the case**. In GMQL, data attribute and dataset names are case-sensitive, be careful!
- **Selection is necessary**. Before doing any GMQL operation, it is necessary to load the data to be processed in the system, by using the SELECT command (see GMQL complete documentation for more details).
- **What is not saved is lost**. Only datasets that are specifically saved with the MATERIALIZE command are saved. All intermediate data are discarded.

3.2 From Problems to GMQL Queries

Every GMQL query is fundamentally an interrogation done on genomic data in order to solve a (biological) problem. The operations supported by the GMQL language, used together, are very powerful; however, using the full expressiveness of the language can be tricky. The most difficult task is understanding how to formalize a potential biological question in a form that can be translated in a set of GMQL operations. There are no fixed rules; thus, in this tutorial, we consider as an example a possible biological problem that requires to answer the following high-level query:
"Consider a set of ChIP-seq experiments made on the K562 cell line and a set of gene bodies. Consider only the ChIP-seq peaks with a score higher than 500. After merging sample replicates from experiments regarding the same antibody, in the merged sample of each antibody count the number of peaks intersecting each gene body. Finally, in each antibody merged sample, for each gene find the nearest ChIP-seq peak."
In the following sections we illustrate and discuss all the GMQL operations involved in answering this query, giving also a brief overview of the various options available.

3.3 Loading Data: Select

Our example requires only two types of genomic data: peaks (i.e enriched binding regions), obtained by ChIP-seq experiments, and gene bodies. Suppose that two datasets are already in the system: HG19_ENCODE_BED, containing various bed files from the ENCODE project, and HG19_BED_ANNOTATION, containing some annotation data in bed format. Thus, the first two statements of our GMQL query are the two SELECTs:

```
PEAKS = SELECT(dataType == 'ChipSeq' AND
                                  cell == 'K562') HG19_ENCODE_BED;
GENES = SELECT(annotation_type == 'gene' AND
             original_provider == 'Ensembl') HG19_BED_ANNOTATION;
```

Out of all samples in the considered **datasets**, these two operations select the ones that have metadata satisfying the conditions expressed in parenthesis. Note that the metadata attributes and values that we used in the example depend on the used datasets, which in this case include ENCODE provided data and metadata and UCSC provided annotations, respectively. In practice, the created PEAKS dataset contains the samples that in the HG19_ENCODE_BED dataset have both "cell K562" and "dataType ChipSeq" entries in their metadata. Similarly, GENES contains only samples present in the HG19_BED_ANNOTATION dataset and having in their metadata both "annotation_type" equal to "gene" and "original_provider" equal to "Ensembl". In our case, PEAKS includes 218 samples, while GENES has only one sample with the list of the gene body regions. More generally, selection predicates can be built with arbitrary parenthesized expressions combining AND, OR and NOT logical operators.

3.4 Filtering Regions: Project

```
F_PEAKS = PROJECT(score > 500) PEAKS;
```

It is **often** necessary to filter genomic regions within samples. In GMQL it is possible to keep only some regions which we are interested in, by using the PROJECT operator. In our example, we filter out the regions in PEAKS that do not have a score value higher than an arbitrary threshold (e.g. 500). It is possible to concatenate conditions using the AND, OR and NOT logical operators. With PROJECT it is also possible to change the region coordinates, for example extending them by a constant. For the full spectrum of options see the language specifications in the GMQL complete documentation.

3.5 Merging Replicas: Cover

Often, selected and filtered samples still contain redundant information. In our example, for a specific cell line we selected available data samples from experiments made with many different antibodies, on average a couple of samples per antibody. At this point we are interested in merging the information of samples obtained using the same antibody. To do so, we make use of the COVER operator. Roughly speaking, COVER takes as input a group of samples and merges them in a single sample. The genomic regions in the resulting merged sample are formed by combining the input regions according to the specific COVER parameters used. A COVER operation is defined based on a minimum and a maximum accumulation value, which define the considered minimum and maximum number of intersecting regions in each specific genomic position. An usage example in our case is:

```
C_PEAKS = COVER(1, ANY; GROUP BY antibody) F_PEAKS;
```

In this example we set the minimum at "1" and maximum with the keyword "ANY", which implies no maximum number at all. With these settings, we obtain the simple union of all the input regions, as shown in Figure 1 for two samples. We could be more strict and set the minimum to "ALL". In this case we would keep only the region parts intersecting in at least N regions, where N is the number of input data samples involved (i.e. we would keep only the region parts present in all input samples).

Note the GROUP BY option; without it, all the 218 samples in F_PEAKS would be merged together and C_PEAKS would consist of only one sample. Instead, we first group the samples in the input dataset that have the same value for the "antibody" attribute in their metadata, and then the Cover operation is applied on each obtained sample group, providing as a result 86 different samples in C_PEAKS. In Figure 1 we show a small genomic portion of a group of the input samples with the same antibody and the resulting output sample.

COVER is a very rich operator with many optional parameters; its full use requires a careful study of the GMQL complete documentation.

Figure 1: Small portion of COVER(1,ANY) result (orange, bottom regions) on two input samples (blue, top regions).

3.6 Counting Intersections: Map

Once the ChIP-seq data have been prepared, we are ready to use them with the annotation data. What we have to do is counting, in the merged sample of each antibody, the number of peaks intersecting each gene body. These types of operations are easily performed by the MAP operator, which takes as *reference* a set of regions in its first input dataset and compares each of them with the regions in each sample within its second input dataset. In each sample, the regions intersecting with each reference region are used to calculate a set of aggregate functions for the reference region. The result is a new dataset with all the samples in the second input dataset, but each sample includes all the reference regions, each one with the values of the aggregate functions calculated on the regions originally included in the sample. For our example query, the GMQL command to use includes only the Count aggregate function:

```
M_GENES = MAP(COUNT) GENES C_PEAKS;
```

Figure 2 shows a small portion of input and output regions of the Map operation on two samples.

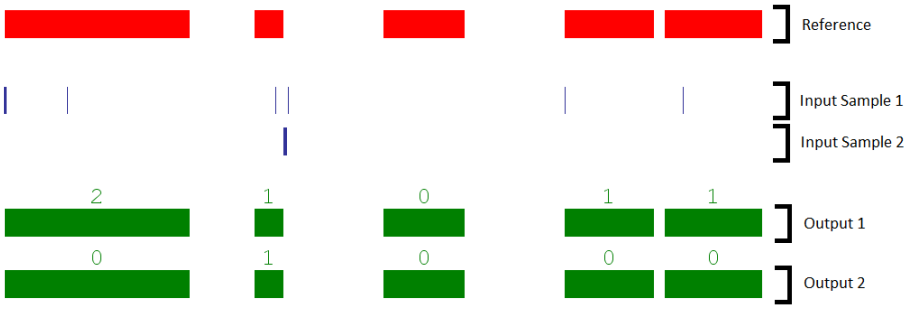

Figure 2: GMQL Map operation. Top (red), gene bodies. Middle (blue), peak regions from two ChIP-seq samples. Bottom (green), the Map result.

In the Map operation several aggregate functions can be used, including the Average of a specific value of the intersecting regions, as well as the Sum, Minimum, Maximum, etc.. Note that the regions in the result contain also all the original information of the reference regions, e.g. the gene ID in our case. As well, metadata of a result sample contain both the original sample metadata and the reference sample metadata. As a general rule, when metadata from different samples are merged, the name of their original sample dataset is added as a prefix to each resulting metadata attribute to avoid ambiguity (e.g. in our case resulting metadata contain the attribute "C_PEAKS.antibody").

3.7 Finding Close Regions: Join

The last operation required by our example query is, for each gene, finding the nearest ChIP-seq peak. We need to perform the search for each antibody used to generate the ChIP-seq samples, keeping all the information collected so far. Problems related to relative distances between genomic regions can be solved in GMQL by using the genometric JOIN operator. In our example case:

```
J_PEAKS = JOIN(left->antibody == right->antibody, MINDISTANCE(1),
               right_distinct) M_GENES C_PEAKS;
```

The genometric JOIN works as follows. First, samples from the left and right operand (i.e. first and second dataset) in input are paired according to the specified condition (if any) on their metadata; in our case, we pair samples obtained with the same antibody (first parameter). It is possible to specify no metadata condition; in this case the Join operation considers the cross product of the samples in the two input datasets.
Once the sample pairs are formed, a specified genometric operation is performed on couples of regions in each sample pair. In our case we specified the MINDISTANCE(1) operation; for each region r_i in the left operand (i.e in the sample of the pair from the first input dataset), it finds the region in the right operand (i.e in the sample of the pair from the second input dataset) at the minimum distance from r_i (Figure 3). All regions that do not satisfy the genometric predicate (i.e. being at minimum distance) are discarded.
Finally, for each pair of samples in input, a new sample is created in output, containing regions determined according to the last Join parameter specified and to the pairs of regions found satisfying the genometric predicate. In our case we specified "*right_distinct*", i.e. to keep the regions from the sample in the second input dataset ("*right*"), but keeping a single instance ("*_distinct*") for each of them if multiple region instances (all with the same coordinates) appear for the same region. All values of all the regions from the right input sample (i.e. the sample in the second input dataset) that contributed with the region from the left input sample (i.e. the sample in the first input dataset) to satisfy the genometric join predicate are added to the output region; thus, in our case all obtained regions contain at least one gene ID.

As an example, a small portion of input and output samples is shown in Figure 3. In the second operand sample (a ChIP-seq sample in our case), the

first region (peak) from the left is not at minimal distance from any region (gene) in the first operand sample (a gene annotation sample in our case); thus it is discarded. The second ChiP-seq peak intersects a gene; thus, the distance between the two is the shortest possible. Finally, the third peak from the left is the closest for the rightmost gene. The last two peaks are therefore included in the result sample.

Figure 3: Join operation. Top (red), gene bodies from a sample in the first input dataset. Middle (green), ChIP-seq peaks from a sample in the second input dataset. Bottom (blue), regions in the result sample.

3.8 Saving Results: Materialize

It is important to remember that all datasets defined in a GMQL query are, by default, temporary. To see and preserve the content of any dataset generated during a GMQL query, the dataset must be materialize. Thus, at the end of our example, we have to write, for instance:

```
MATERIALIZE J_PEAKS;
```

It saves the content of J_PEAKS and registers the J_PEAKS dataset in the system to make it seamlessly usable in other GMQL queries. There is no limit to the number of materialization in a query; every intermediate dataset can be saved to check its content. However, the Materialize operation is time expensive; for best performance save the relevant data only.

3.9 Full Query, Execution, Results

The complete GMQL query that solves the problem can be obtained by adding up everything that has been discussed so far. The complete listing for such query is the following:

```
PEAKS = SELECT(dataType == 'ChipSeq' AND
                                  cell == 'K562') HG19_ENCODE_BED;
GENES = SELECT(annotation_type == 'gene' AND
             original_provider == 'Ensembl') HG19_BED_ANNOTATION;
F_PEAKS = PROJECT(score > 500) PEAKS;
C_PEAKS = COVER(1, ANY; GROUP BY antibody) F_PEAKS;
M_GENES = MAP(COUNT) GENES C_PEAKS;
J_PEAKS = JOIN(left->antibody == right->antibody, MINDISTANCE(1),
               right_distinct) M_GENES C_PEAKS;
MATERIALIZE J_PEAKS;
```

A GMQL query like this must be saved as simple text file with extension .gmql to work properly. To execute a GMQL query, use the command:

```
GMQLScriptManager CompileRun <LOCAL/MAPREDUCE> <queryURL>
```

where queryURL is the complete path of the text file containing the GMQL query to be executed, the LOCAL option makes using local computing resources, whereas the MAPREDUCE option makes the query to be executed in the Hadoop system, if any has been installed to work with the GMQL system (see the QuickStart documentation). This command executes the GMQL query and registers the results in the system. Be aware that, in order to avoid ambiguity, query and user name, as well as day and time of the query execution start, are added to the name of each materialized dataset. The result files are saved in the Hadoop file system if the query is executed in map-reduce mode, or in the folder $.../userName/data/results/$ in the GQML directory if the query is executed in local mode. In any case, it is possible to copy any dataset in a folder of the local file system by using the command:

```
GMQLScriptManager CopyDsToLocal <DatasetName> <DestinationURL>
```

Results are saved in GTF format, which we stress adopts the 1-base notation for the genomic region coordinates (i.e. coordinate counting starts from 0). Thus, if a data format adopting a 0-base notation (e.g. bed, broadPeak, narrowPeak) is used as input to a GMQL query, the left coordinate of each genomic region in the GMQL output result is suitably changed to comply with the 1-base notation of the output GTF format.

When possible, result values are stored in the GTF default columns; for example a value associated with the name "score" will be stored in the *score* GTF column. All other resulting attribute-value pairs are included in the last (ninth) column of the GTF format.