RGMQL: GenoMetric Query Language for R/Bioconductor

Simone Pallotta and Marco Masseroli

2018-04-30

Contents

1 Introduction

Recent years have seen a tremendous increase in the volume of data generated in the life sciences, especially propelled by the rapid progress of Next Generation Sequencing (NGS) technologies. These high-throughput technologies can produce billions of short DNA or RNA fragments in excess of a few terabytes of data in a single run. Next-generation sequencing refers to the deep, in-parallel DNA sequencing technologies providing massively parallel analysis and extremely high-throughput from multiple samples at much reduced cost. Improvement of sequencing technologies and data processing pipelines is rapidly providing sequencing data, with associated high-level features, of many individual genomes in multiple biological and clinical conditions. To make effective use of the produced data, the design of big data algorithms and their efficient implementation on modern high performance computing infrastructures, such as clouds, CPU clusters and network infrastructures, is required in order to achieve scalability and performance. For this purpose, the GenoMetric Query Language (GMQL) has been proposed as high-level, declarative language to process, query, and compare multiple and heterogeneous genomic datasets for biomedical knowledge discovery[1]

1.1 Purpose

A very important emerging problem is to make sense of the enormous amount and variety of NGS data becoming available, i.e., to discover how different genomic regions and their products interact and cooperate with each other. To this aim, the integration of several heterogeneous DNA feature data is required. Such big genomic feature data are collected within numerous and heterogeneous files, usually distributed within different repositories, lacking an attribute-based organization and a systematic description of their metadata. These heterogeneous data can contain the hidden answers to very important biomedical questions. To unveil them, standard tools already available for knowledge extraction are too specialized or present powerful features, but have a rough interface not well-suited for scientists/biologists. GMQL addresses these aspects using cloud-based technologies (including Apache Hadoop, mapReduce, and Spark), and focusing on genomic data operations written as simple queries with implicit iterations over thousands of heterogeneous samples, computed efficiently[2]. This RGMQL package makes easy to take advantage of GMQL functionalities also to scientists and biologists with limited knowledge of query and programming languages, but used to the R/Bioconductor environment. This package is built over a GMQL scalable data management engine written in Scala programming language, released as Scala API[3] providing a set of functions to combine, manipulate, compare, and extract genomic data from different data sources both from local and remote datasets. These functions, built extending functionalities available in the R/Bioconductor framework, allow performing complex GMQL processing and queries without knowledge of GMQL syntax, but leveraging on R idiomatic paradigm and logic.

2 Genomic Data Model

The Genomic Data Model (GDM) is based on the notions of datasets and samples[4] Datasets are collections of samples, and each sample consists of two parts, the region data, which describe portions of the genome, and the metadata, which describe sample general properties and how observations are collected. In contrast to other data models, it clearly divides, and

comprehensively manages, observations about genomic regions and metadata. GDM provides a flat attribute based organization, just requiring that each dataset is associated with a given data schema, which specifies the attributes and their type of region data. The first attributes of such schema are fixed (chr, left, right, strand); they represent the genomic region identifying coordinates. In addition, metadata have free attribute-value pair format.

2.1 Genomic Region

Genomic region data describe a broad variety of biomolecular aspects and are very valuable for biomolecular investigation. A genomic region is a portion of a genome, qualified by a quadruple of values called region coordinates:

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

Regions can have associated an arbitrary number of attributes with their value, according to the processing of DNA, RNA or epigenomic sequencing reads that determined the region.

2.2 Metadata

Metadata describe the biological and clinical properties associated with each sample. They are usually collected in a broad variety of data structures and formats that constitute barriers to their use and comparison. GDM models metadata simply as arbitrary semi-structured attribute-value pairs, where attributes may have multiple values.

2.3 Genomic Sample

Formally, a sample s is a collection of genomic regions modelled as the following triple:

$$< id, < r_i, v_i >, m_j >$$

where:

- id is the sample identifier
- Each region is a pair of coordinates r_i and values v_i
- Metadata m_j are attribute-value pairs $< a_j, v_j >$

Note that the sample id attribute provides a many-to-many connection between regions and metadata of a sample. Through the use of a data type system to express region data, and of arbitrary attribute-value pairs for metadata, GDM provides interoperability across datasets in multiple formats produced by different experimental techniques.

2.4 Dataset

A dataset is a collection of samples uniquely identified, with the same region schema and with each sample consisting of two parts:

- region data: describing characteristics and location of genomic portions
- metadata: expressing general properties of the sample

Each dataset is typically produced within the same project by using the same or equivalent technology and tools, but with different experimental conditions, described by metadata.

Datasets contain large number of information describing regions of a genome, with data encoded in human readable format using plain text files.

GMQL datasets are materialized in a standard layout composed of three types of files:

1. genomic region tab-delimited text files with extension .gdm, or .gtf if in standard GTF format
2. metadata attribute-value tab-delimited text files with the same fullname (name and extension) of the correspondent genomic region file and extension .meta
3. schema XML file containing region attribute names and types

All these files reside in a unique folder called $files$.

In RGMQL package, dataset files are considered read-only. Once read, genomic information is represented in an abstract data structure inside the package, mapped to a R GRanges data structure as needed for optimal use and interoperability with all available R/Bioconductor functions.

3 GenoMetric Query Language

The GenoMetric Query Language name stems from such language ability to deal with genomic distances, which are measured as number of nucleotide bases between genomic regions (aligned to the same reference genome) and computed using arithmetic operations between region coordinates. GMQL is a high-level, declarative language that allows expressing queries easily over genomic regions and their metadata, in a way similar to what can be done with the Structured Query Language (SQL) over a relational database. GMQL approach exhibits two main differences with respect to other tools based on Hadoop, mapReduce framework, and Spark engine technologies to address similar biomedical problems:

- GMQL:
  1. reads from processed datasets
  2. supports metadata management
- Others:
  1. read generally from raw or aligned data from NGS machines
  2. provide no support for metadata management

GMQL is the appropriate tool for querying many genomic datasets and very many samples of numerous processed genomic region data that are becoming available. Note however that GMQL performs worse than some other available systems on a small number of small-scale datasets, but these other systems are not cloud-based; hence, they are not adequate for efficient big data processing and, in some cases, they are inherently limited in their data management capacity, as they only work as RAM memory resident processes.

3.1 GMQL Query Structure

A GMQL query, or script, is expressed as a sequence of GMQL operations with the following structure:

$$< variable >= operator(< parameters >) < variable >;$$

where each $< variable >$ stands for a GDM dataset

This RGMQL package brings GMQL functionalities into R environment, allowing users to build directly a GMQL query without knowing the GMQL syntax, but using R idiomatic expressions and available R functions suitably extended. In RGMQL every GMQL operations is translated into a R function and expressed as:

$$variable = operator(variable, parameters)$$

It is very similar to the GMQL syntax for operation expression, although expressed with the R idiomatic paradigm and logic, with parameters totally built using R native data structures such as lists, matrices, vectors or R logic conditions.

4 Processing Environments

In this section, we show how GMQL processing is built in R, which operations are available in RGMQL, and the difference between local and remote dataset processing.

4.1 Local Processing

RGMQL local processing consumes computational power directly from local CPUs/system while managing datasets (both GMQL or generic text plain datasets).

4.1.1 Initialization

Load and attach the RGMQL package in a R session using library function:

```
library('RGMQL')
## Loading required package: RGMQLlib
## GMQL successfully loaded
```

RGMQL depends on another packaage *RGMQLlib* automatically installed and loaded once install and loaded *RGMQL*; if not, you can install it from Bioconductor and load it using:

```
library('RGMQLlib')
```

Before starting using any GMQL operation we need to initialise the GMQL context with the following code:

```
init_gmql()
```

The function *init_gmql()* initializes the context of scalable data management engine laid upon Spark and Hadoop, and the format of materialized result datasets. Details on this and all other functions are provided in the R documentation for this package (i.e., help(RGMQL)).

4.1.2 Read Dataset

After initialization, we need to read datasets. We already defined above the formal definition of dataset and the power of GMQL to deal with data in a variety of standard tab-delimited text formats. In the following, we show how to get data from different sources.

We distinguish two different cases:

1. Local dataset:
   A local dataset is a folder with sample files (region files and correspondent metadata files) on the user computer. As data are already in the user computer, we simply execute:

```
gmql_dataset_path <- system.file("example", "EXON", package = "RGMQL")
data_out = read_gmql(gmql_dataset_path)
```

In this case we are reading a GMQL dataset specified by the path of its folder "EXON" within the subdirectory "example" of the package "RGMQL". It does not matter what kind of format the data are, *read_gmql()* reads many standard tab-delimited text formats without the need of specifying any additional input parameter.

2. GRangesList:
   For better integration in the R environment and with other R packages, we provide the *read_GRangesList()* function to read directly from R memory using GRangesList as input.

```
library("GenomicRanges")
## Loading required package: stats4
## Loading required package: BiocGenerics
## Loading required package: parallel
##
## Attaching package: 'BiocGenerics'
## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB
## The following objects are masked from 'package:stats':
##
##     IQR, mad, sd, var, xtabs
## The following objects are masked from 'package:base':
##
##     Filter, Find, Map, Position, Reduce, anyDuplicated, append,
##     as.data.frame, basename, cbind, colMeans, colSums, colnames,
##     dirname, do.call, duplicated, eval, evalq, get, grep, grepl,
##     intersect, is.unsorted, lapply, lengths, mapply, match, mget,
##     order, paste, pmax, pmax.int, pmin, pmin.int, rank, rbind,
##     rowMeans, rowSums, rownames, sapply, setdiff, sort, table,
##     tapply, union, unique, unsplit, which, which.max, which.min
## Loading required package: S4Vectors
##
## Attaching package: 'S4Vectors'
## The following object is masked from 'package:base':
##
```

```
##     expand.grid
## Loading required package: IRanges
## Loading required package: GenomeInfoDb

# Granges Object with one region: chr2 and two metadata columns: score = 5
# and GC  = 0.45

gr1 <- GRanges(seqnames = "chr2",
    ranges = IRanges(103, 106), strand = "+", score = 5L, GC = 0.45)

# Granges Object with two regions both chr1 and two metadata columns: score = 3
# for the fist region and score = 4 for the second one, GC  = 0.3 and 0.5
# for the first and second region, respectively

gr2 <- GRanges(seqnames = c("chr1", "chr1"),
    ranges = IRanges(c(107, 113), width = 3), strand = c("+", "-"),
    score = 3:4, GC = c(0.3, 0.5))

grl <- GRangesList("txA" = gr1, "txB" = gr2)
data_out <- read_GRangesList(grl)
## Warning in read_GRangesList(grl): No metadata.
## We provide two metadata for you:
##
## 1.provider = PoliMi
## 2.application = RGMQL
```

In this example we show how versatile the RGMQL package is. As specified above, we can directly read a list of GRanges previously created starting from two GRanges. Both *read_GRangesList()* and *read_gmql()* functions return a result object, in this case *data_out*: an instance of GMQLDataset class used as input for executing the subsequent GMQL operation. NOTE: if *metadata(grl)* is empty, the function provides two default metadata:

- "provider" = "PoliMi"
- "application" = "RGMQL"

4.1.3 Queries

GMQL is not a traditional query language: With "query" we intend a group of operations that together produce a result; in this sense GMQL queries are more similar to SQL scripts. GMQL programming consists of a series of select, union, project, difference (and so on . . .) commands.

Let us see a short example:

Find somatic mutations in exons. Consider mutation data samples of human breast cancer cases. For each sample, quantify the mutations in each exon and select the exons with at least one mutation. Return the list of samples ordered by the number of such exons.

```
# These statements define the paths to the folders "EXON" and "MUT" in the
# subdirectory "example" of the package "RGMQL"
```

```
exon_path <- system.file("example", "EXON", package = "RGMQL")
mut_path <- system.file("example", "MUT", package = "RGMQL")

# Read EXON folder as a GMQL dataset named "exon_ds" containing a single
# sample with exon regions, and MUT folder as a GMQL dataset named "mut_ds"
# containing multiple samples with mutation regions

exon_ds <- read_gmql(exon_path)
mut_ds <- read_gmql(mut_path)

# Filter out mut_ds based on predicate

mut = filter(mut_ds, manually_curated__dataType == 'dnaseq' &
                clinical_patient__tumor_tissue_site == 'breast')

# Filter out exon_ds based on predicate

exon = filter(exon_ds, annotation_type == 'exons' &
                    original_provider == 'RefSeq')

# For each mutation sample, count mutations within each exon while
# mapping the mutations to the exon regions using the map() function

exon1 <- map(exon, mut)

# Remove exons in each sample that do not contain mutations

exon2 <- filter(exon1, count_left_right >= 1)

# Using the extend() function, count how many exons remain in each sample and
# store the result in the sample metadata as a new attribute-value pair,
# with exon_count as attribute name

exon3 <- extend(exon2, exon_count = COUNT())

# Order samples in descending order of the added metadata exon_count

exon_res = arrange(exon3, list(DESC("exon_count")))
```

If you want to store persistently the result, you can materialize it into specific path defined as input parameter.

```
# Materialize the result dataset on disk
collect(exon_res)
```

By default *collect()* has R working directory as storing path and *ds1* as name of resulted dataset folder.

4.1.4 Execution

RGMQL processing does not generate results until you invoke the *execute()* function.

```
execute()
```

execute() can be issued only if at least one *read()* and at least one *collect()* are present in the RGMQL query, otherwise an error is generated. Data are saved in the path specified in every *collect()* present in the query. After the execution, the context of scalable data management engine is stopped and a new invocation of *init_gmql()* is needed.

Beside *execute()* we can use:

```
g <- take(exon_res, rows = 45)
```

to execute all *collect()* commands in the RGMQL query and extract data as GRangesList format, a GRangesList for each *collect()* and a GRanges for each sample. NOTE: GRangesList are contained in the R environment and are not saved on disk.

With the *rows* parameter it is possible to specify how many rows, for each sample inside the input dataset, are extracted; by default, the *rows* parameter value is 0, that means all rows are extracted. Note that, since we are working with big data, to extract all rows could be very time and space consuming.

4.2 Remote Processing

RGMQL remote processing consumes computational power from remote cluster/system while managing GMQL datasets.

Remote processing exists in two flavour:

- REST web services:
  User can write GMQL queries (using original GMQL syntax) to be executed remotely on remote data (or local data previously uploaded).
- Batch execution:
  Similar to local execution; user reads data and the system automatically uploads them on the remote system if they are not already there; once loaded, RGMQL functions can be issued to manage and process remote data.

4.2.1 REST Web Services

This RGMQL package allows invoking REST services that implement the commands specified at link.

4.2.1.1 Initialization

GMQL REST services require login; so, the first step is to perform logon with user and password, or as guest. Upon successful logon, you get a request token that you must use in every subsequent REST call. Login can be performed using the *login_gmql()* function:

```
test_url = "http://genomic.deib.polimi.it/gmql-rest-r/"
login_gmql(test_url)
## [1] "your Token is da20b08e-0680-4f0e-b9e3-d95aa58c8a0e"
```

It saves the token in the Global R environment within the variable named *authToken*. With this token you can call all the functions in the GMQL REST web services suite.

4.2.1.2 Execution

User can write a GMQL query as in the following example, and run it as second parameter of the *run_query()* function:

```
job <- run_query(test_url, "query_1", "DNA = SELECT() Example_Dataset_1;
MATERIALIZE DNA INTO RESULT_DS;", output_gtf = FALSE)
```

Or the user can execute a query reading it directly from file (e.g., the "query1.txt" file in the "example" subdirectory of the RGMQL package):

```
query_path <- system.file("example", "query1.txt", package = "RGMQL")
job <- run_query_fromfile(test_url, query_path, output_gtf = FALSE)
```

Once run, query continues on the remote server while *run_query()* or *run_query_fromfile()* returns immediately. User can extract from result (job) the job_id and status. jod_id can then be used to continuously invoke log and trace calls, both in this RGMQL package, to check for job completed status.

```
jod_id <- job$id
trace_job(test_url, jod_id)
```

Then, results materialized on the remote repository can by downloaded locally and imported in GRangesList using the functions in this RGMQL package (see Import/Export).

The returned *job* contains also the name of the created datasets, one for each materialize in the GMQL query run, in the same order; the first can be downloaded locally with:

```
name_dataset <- job$datasets[[1]]$name
download_dataset(test_url, name_dataset)
```

By default *download_dataset()* has R working directory as local storing path.

Once download is done, we can logout from remote repository using:

```
logout_gmql(test_url)
## [1] "Logout"
```

logout_gmql() deletes the $authToken$ from R environment.

Downloaded result dataset can be then imported and used in the R environment as GRangesList (see Import/Export). Alternatively, the remote result dataset can be directly downloaded and imported in the R environment as GRangesList using the function *download_as_GRangesList()*:

```
name_dataset <- job$datasets[[1]]$name
grl = download_as_GRangesList(remote_url, name_dataset)
```

4.2.2 Batch Execution

This execution type is similar to local processing (syntax, functions, and so on . . .) except that materialized data are stored only on the remote repository, from where they can be downloaded locally and imported in GRangesList using the functions in this RGMQL package (see Import/Export).

Before starting with an example, note that we have to log into remote infrastructure with login function:

```
login_gmql(test_url)
```

Otherwise, we can initialize the data engine with a remote url:

```
init_gmql(url = test_url)
## [1] "your Token is 64785c5f-92a9-4bee-a627-6d1312debdd5"
```

In this way login is automatically performed as specified above.

After initialization we have to change to remote processing:

```
remote_processing(TRUE)
## [1] "Remote processing On"
```

or alternatively, instead of switching mode, we can initilize the data engine setting remote processing as TRUE:

```
init_gmql(url = test_url, remote_processing = TRUE)
```

Once done, we can start building our query:

```
## Read the remote dataset Example_Dataset_1
## Read the remote dataset Example_Dataset_2

TCGA_dnaseq <- read_gmql("public.Example_Dataset_1", is_local = FALSE)
HG19_bed_ann <- read_gmql("public.Example_Dataset_2", is_local = FALSE)

## Filter out TCGA_dnaseq based on predicate

mut = filter(TCGA_dnaseq, manually_curated__dataType == 'dnaseq' &
                clinical_patient__tumor_tissue_site == 'breast')

# Filter out HG19_bed_ann based on predicate

exon = filter(HG19_bed_ann, annotation_type == 'exons' &
                    original_provider == 'RefSeq')

# For each mutation sample, count mutations within each exon while
# mapping the mutations to the exon regions using the map() function

exon1 <- map(exon, mut)

# Remove exons in each sample that do not contain mutations
```

```
exon2 <- filter(exon1, count_left_right >= 1)

# Using the extend() function, count how many exons remain in each sample and
# store the result in the sample metadata as a new attribute-value pair,
# with exon_count as attribute name

exon3 <- extend(exon2, exon_count = COUNT())

# Order samples in descending order of the added metadata exon_count

exon_res = arrange(exon3, list(DESC("exon_count")))
```

N.B. in case of remote processing we have to specify the name of the output dataset (not necessary the same as imput) as the second parameter in order to correctly process the qeury on the remote GMQL system. Here is the R code we use:

```
collect(exon_res,"exon_res")
```

```
execute()
```

In this case R processing continues until remote processing ends. With remote processing, after the execution, the context of scalable data management engine is not stopped, and can be used for further queries. Materialized datasets are stored only remotely, from where they can be downloaded and directly imported into the R environment as GRangesList using the function *download_as_GRangesList()*:

```
name_dataset <- job$datasets[[1]]$name
grl = download_as_GRangesList(remote_url, name_dataset)
```

Alternatively, the remote materialized dataset can be first downloaded and stored locally:

```
name_dataset <- job$datasets[[1]]$name
download_dataset(test_url, name_dataset)
```

and then imported in the R environment as GRangesList (see Import/Export).

In any case, once download is done, we can logout from remote repository using:

```
logout_gmql(test_url)
```

4.3 Mixed Processing

As said before, the processing flavour can be switched using the function:

```
remote_processing(TRUE)
## [1] "Remote processing On"
```

An user can switch processing mode until the first *collect()* has been performed.

This kind of processing comes from the fact that the *read_gmql()* function can accept either a local dataset or a remote repository dataset, even in the same query as in the following example:

```
# This statement defines the path to the folder "MUT" in the subdirectory
# "example" of the package "RGMQL"

mut_path <- system.file("example", "MUT", package = "RGMQL")

# Read MUT folder as a GMQL dataset named "mut_ds" containing a single
# sample with mutation regions

mut_ds <- read_gmql(mut_path, is_local = TRUE)

# Read the remote dataset Example_Dataset_2

HG19_bed_ann <- read_gmql("public.Example_Dataset_2", is_local = FALSE)

# Filter out based on predicate

mut = filter(mut_ds, manually_curated__dataType == 'dnaseq' &
                clinical_patient__tumor_tissue_site == 'breast')

exon = filter(HG19_bed_ann, annotation_type == 'exons' &
                    original_provider == 'RefSeq')

# For each mutation sample, count mutations within each exon while
# mapping the mutations to the exon regions using the map() function

exon1 <- map(exon, mut)

# Remove exons in each sample that do not contain mutations

exon2 <- filter(exon1, count_left_right >= 1)

# Using the extend() function, count how many exons remain in each sample and
# store the result in the sample metadata as a new attribute-value pair,
# with exon_count as attribute name

exon3 <- extend(exon2, exon_count = COUNT())

# Order samples in descending order of the added metadata exon_count

exon_res = arrange(exon3, list(DESC("exon_count")))
```

Materialize result:

```
collect(exon_res,"exon_res")
```

Execute processing:

```
execute()
```

As we can see, the two *read_gmql()* functions above read from different sources: *mut_ds* from local dataset, *HG19_bed_ann* from remote repository.

If we set remote processing to false (*remote_processing(FALSE)*), the execution is performed locally first downloading all needed datasets from remote repositories, otherwise all local datasets are automatically uploaded to the remote GMQL repository associated with the remote system where the processing is performed. In the latter case, materialized datasets are stored on remote repository, from where they can be downloaded and imported into the R environment as in the remote processing scenario (see Remote Processing).

NOTE: The public datasets cannot be downloaded from a remote GMQL repository by design.

5 Utilities

The RGMQL package contains functions that allow the user to interface with other packages available in R/Bioconductor repository, e.g., GenomicRanges, and TFARM. These functions return GRangesList or GRanges with metadata associated, if present, as data structure suitable to further processing in other R/Bioconductor packages.

5.1 Import/Export

We can import a GMQL dataset into R environment as follows:

```
# This statement defines the path to the folder "EXON" in the subdirectory
# "example" of the package "RGMQL"

dataset_path <- system.file("example", "EXON", package = "RGMQL")

# Import the GMQL dataset EXON as GRangesList

imported_data <- import_gmql(dataset_path, is_gtf = FALSE)
imported_data
## GRangesList object of length 1:
## $S_00000
## GRanges object with 571 ranges and 2 metadata columns:
##         seqnames              ranges strand |         name     score
##            <Rle>           <IRanges>  <Rle> |     <factor> <integer>
##     [1]     chr1         11874-12227      + |    NR_046018         0
##     [2]     chr1         12613-12721      + |    NR_046018         0
##     [3]     chr1       917445-917497      - | NM_001291366         0
##     [4]     chr1       934342-934812      - |    NM_021170         0
##     [5]     chr1       934344-934812      - | NM_001142467         0
##     ...      ...                 ...    ... .          ...       ...
##   [567]     chr7 128612480-128612636      - |    NR_034053         0
##   [568]     chr7 128612480-128612636      - | NM_001191028         0
##   [569]     chr7 128612480-128612636      - |    NM_012470         0
##   [570]     chr7 128614922-128615016      - |    NR_034053         0
```

```
##   [571]     chr7 128614922-128615016      - | NM_001191028         0
##
## -------
## seqinfo: 11 sequences from an unspecified genome; no seqlengths

# and its metadata

metadata(imported_data)
## $S_00000
## $S_00000$annotation_type
## [1] "exons"
##
## $S_00000$assembly
## [1] "hg19"
##
## $S_00000$name
## [1] "RefSeqGeneExons"
##
## $S_00000$original_provider
## [1] "RefSeq"
##
## $S_00000$provider
## [1] "POLIMI"
```

The second parameter *is_gtf* must specify the file format: .GTF or .GDM.

We can export a GRangesList as GMQL dataset as follows:

```
# This statement defines the path to the subdirectory "example" of the
# package "RGMQL"

dir_out <- system.file("example", package = "RGMQL")

# Export the GRangesList 'data' as GMQL dataset called 'example' at destination
# path

export_gmql(imported_data, dir_out, is_gtf = TRUE)
## The following `from` values were not present in `x`: type, phase
## [1] "Export to GTF complete"
```

The second parameter *is_gtf* must specify the file format: .GTF or .GDM.

5.2 Filter and Extract

We can also import only a part of a GMQL dataset into R environment, by filtering its content as follows:

```
# This statement defines the path to the folder "TEAD" in the subdirectory
# "example" of the package "RGMQL"

data_in <- system.file("example", "TEAD", package = "RGMQL")
```

```
matrix <- filter_and_extract(data_in, metadata = NULL, region_attributes = c("count"))
matrix
## GRanges object with 5553 ranges and 19 metadata columns:
##          seqnames              ranges strand |  count.EGR1 count.TCF12
##             <Rle>           <IRanges>  <Rle> | <character> <character>
##      [1]     chr1       858942-859396      * |           0           1
##      [2]     chr1       875511-876089      * |           1           1
##      [3]     chr1       877036-877497      * |           1           1
##      [4]     chr1       935245-936485      * |           1           1
##      [5]     chr1       956311-956941      * |           0           1
##      ...      ...                 ...    ... .         ...         ...
##   [5549]     chrX 153236979-153237824      * |           1           1
##   [5550]     chrX 153625328-153625655      * |           0           0
##   [5551]     chrX 153626485-153627435      * |           0           1
##   [5552]     chrX 154254675-154255379      * |           0           1
##   [5553]     chrX 154996072-154996374      * |           0           0
##          count.TEAD4 count.ZBTB7A count.TEAD41 count.TEAD42  count.REST
##          <character>  <character>  <character>  <character> <character>
##      [1]           0            1            0            0           0
##      [2]           1            1            1            0           0
##      [3]           0            1            0            0           0
##      [4]           0            1            1            2           1
##      [5]           0            1            0            0           0
##      ...         ...          ...          ...          ...         ...
##   [5549]           0            0            0            0           0
##   [5550]           0            0            0            0           0
##   [5551]           0            0            0            2           0
##   [5552]           0            0            0            1           0
##   [5553]           0            0            0            0           0
##          count.POLR2A count.TEAD43  count.TAF1 count.TEAD44 count.TEAD45
##           <character>  <character> <character>  <character>  <character>
##      [1]            1            1           0            0            0
##      [2]            0            1           0            1            0
##      [3]            0            1           1            0            0
##      [4]            1            1           1            0            0
##      [5]            1            1           1            1            0
##      ...          ...          ...         ...          ...          ...
##   [5549]            1            1           1            1            0
##   [5550]            1            1           1            1            0
##   [5551]            1            1           1            1            0
##   [5552]            1            1           2            1            0
##   [5553]            0            1           0            0            0
##          count.TEAD46   count.MAX count.TEAD47   count.SRF count.TEAD48
##           <character> <character>  <character> <character>  <character>
##      [1]            1           1            0           0            1
##      [2]            1           1            0           0            1
##      [3]            1           1            0           0            1
##      [4]            1           1            0           1            1
##      [5]            1           1            0           0            1
##      ...          ...         ...          ...         ...          ...
```

```
##   [5549]            0           1           0           0           0
##   [5550]            0           1           0           0           0
##   [5551]            0           1           0           1           1
##   [5552]            0           1           0           1           1
##   [5553]            0           0           0           0           0
##          count.TEAD49 count.FOXM1
##           <character> <character>
##      [1]            0           0
##      [2]            0           0
##      [3]            0           0
##      [4]            1           0
##      [5]            0           0
##      ...          ...         ...
##   [5549]            1           0
##   [5550]            0           0
##   [5551]            1           0
##   [5552]            1           0
##   [5553]            0           0
##   -------
##   seqinfo: 23 sequences from an unspecified genome; no seqlengths
```

filter_and_extract() filters the samples in the input dataset based on their specified *metadata*, and then extracts as metadata columns of GRanges the vector of region attributes you specify to retrieve from each filtered sample from the input dataset. If the $metadata$ argument is NULL, all samples are taken. The number of obtained columns is equal to the number of samples left after filtering, multiplied by the number of specified region attributes. If *region_attributes* is not specified, only the fundamental elements of GRanges are extracted, i.e., the genomic coordinates. Note that this function works only if every sample in dataset includes the same number of regions with the same coordinates. Each metadata column is named using *region_attributes* concatenated with the function input parameter *suffix*. By default *suffix* correspond to a metadata: *antibody_target*.

5.3 Metadata

Each sample of a GMQL dataset has its own metadata associated and generally every metadata attribute has a single value. The case with distinct values for the same metadata attribute is shown in the figure below for the *disease* metadata attribute.

```
annotation_type     exons
assembly            hg19
disease             cancer
disease             neurofibromatosis type 1
disease             pulmonary fibrosis
disease             Tuberous sclerosis
name                RefSeqGeneExons
original_provider   RefSeq
provider            POLIMI
```

In this case GMQL automatically handles this situation. In the Import/export paragraph, we showed that a GMQL dataset can be imported into R environment as a GRangesList, and so its metadata too.

```
# This statement defines the path to the folder "DATASET_META" in the
# subdirectory "example" of the package "RGMQL"

dataset_path <- system.file("example", "DATASET_META", package = "RGMQL")

# Import the GMQL dataset DATASET_META as GRangesList

grl_data <- import_gmql(dataset_path, is_gtf = FALSE)
grl_data
## GRangesList object of length 1:
## $S_00000
## GRanges object with 571 ranges and 2 metadata columns:
##         seqnames              ranges strand |         name     score
##            <Rle>           <IRanges>  <Rle> |     <factor> <integer>
##     [1]     chr1         11874-12227      + |    NR_046018         0
##     [2]     chr1         12613-12721      + |    NR_046018         0
##     [3]     chr1       917445-917497      - | NM_001291366         0
##     [4]     chr1       934342-934812      - |    NM_021170         0
##     [5]     chr1       934344-934812      - | NM_001142467         0
##     ...      ...                 ...    ... .          ...       ...
##   [567]     chr7 128612480-128612636      - |    NR_034053         0
##   [568]     chr7 128612480-128612636      - | NM_001191028         0
##   [569]     chr7 128612480-128612636      - |    NM_012470         0
##   [570]     chr7 128614922-128615016      - |    NR_034053         0
##   [571]     chr7 128614922-128615016      - | NM_001191028         0
##
## -------
## seqinfo: 11 sequences from an unspecified genome; no seqlengths

# and its metadata

metadata(grl_data)
## $S_00000
## $S_00000$annotation_type
## [1] "exons"
##
## $S_00000$assembly
## [1] "hg19"
##
## $S_00000$disease
## [1] "cancer"
##
## $S_00000$disease
## [1] "pulmonary fibrosis"
##
## $S_00000$disease
## [1] "Tuberous sclerosis"
```

```
##
## $S_00000$disease
## [1] "neurofibromatosis type 1"
##
## $S_00000$name
## [1] "RefSeqGeneExons"
##
## $S_00000$original_provider
## [1] "RefSeq"
##
## $S_00000$provider
## [1] "POLIMI"
```

The metadata are stored as simple list in the form key-values and it does not matter if mutiple values for the same metadata attribute are present; all values are stored and shown. Difficulties can arise when we need to get all the metadata values; normally, since the metadata list is in the form key-value, we can extract the metadata values using:

```
# store metadata on variable a

a = metadata(grl_data)

# get disease value of sample S_00000

a$S_00000$disease
## [1] "cancer"
```

Yet, in this case only the first disease value is shown. If we want to retrieve all disease values, we should use instead:

```
# get all disease values of sample S_00000

a$S_00000[which(names(a$S_00000) %in% "disease")]
## $disease
## [1] "cancer"
##
## $disease
## [1] "pulmonary fibrosis"
##
## $disease
## [1] "Tuberous sclerosis"
##
## $disease
## [1] "neurofibromatosis type 1"
```

References

1. Masseroli M, Pinoli P, Venco F, Kaitoua A, Jalili V, Paluzzi F, Muller H, Ceri S. GenoMetric Query Language: A novel approach to large-scale genomic data management. *Bioinformatics*. 2015;31(12):1881-1888.

2. Ceri S, Kaitoua A, Masseroli M, Pinoli P, Venco F. Data management for heterogeneous genomic datasets. *IEEE/ACM Trans Comput Biol Bioinform*. 2017;14(6):1251-1264.

3. GenoMetric Query Language open source software on GitHub repository. 2017. https://github.com/DEIB-GECO/GMQL/.

4. Masseroli M, Kaitoua A, Pinoli P, Ceri S. Modeling and interoperability of heterogeneous genomic big data for integrative processing and querying. *Methods*. 2016;111:3-11.