

GMQL - Introduction to the language (with Federated extensions)

Introduction	1
A. GENOMIC DATA MODEL (GDM)	1
B. BASIC OPERATORS	1
<i>Foreword: Syntactic conventions and other observations</i>	2
1) SELECT	3
2) MATERIALIZE	7
3) PROJECT	8
4) EXTEND	13
5) ORDER	14
6) GROUP	17
7) MERGE	21
8) UNION	22
9) DIFFERENCE	23
10) MAP	25
11) JOIN	28
12) COVER	42
<i>Cover variants</i>	45
C. FEDERATED FEATURES	47

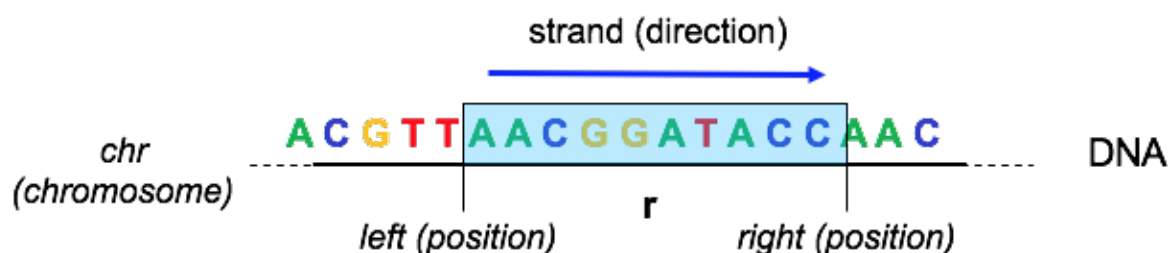
Introduction

This document extends the “GMQL – Introduction to the language” document, available at http://www.bioinformatics.deib.polimi.it/genomic_computing/GMQLsystem/doc/GMQL_introduction_to_the_language.pdf . In particular, it adds the description of the directives and operator parameters that allow the user to run GMQL queries within a federated environment. After a short presentation of the Genomic Data Model adopted by GMQL (Section A), the document contains a description of the basic operators of the language (Section B). After listing syntactic conventions, Section B reports the list of all operators in GMQL along with their parameters, syntax and general usage. For each operator also a list of basic examples is provided, showcasing how to combine different parameters and describing the associated semantics. Finally, Section C, describes how to modify a query in order to make it run within a federation of GMQL instances, by introducing the new directives and operator parameters.

A. GENOMIC DATA MODEL (GDM)

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

1. **Genomic region values** (or simply **regions**), aligned w.r.t. a given reference, with specific left-right ends within a chromosome:



Regions can store different information regarding the “spot” they mark in a particular sample, such as region length or statistical significance. Regions of the model describe processed data, e.g., mutations, expression or bindings; they have a **schema**, with 5 common attributes (*id*, *chr*, *left*, *right*, *strand*) including the id of the region and the region coordinates, along the aligned reference genome, and then arbitrary typed attributes. This provides interoperability across a plethora of genomic data formats;

2. **Metadata**, storing all the knowledge about the particular sample, are arbitrary attribute-value pairs, independent from any standardization attempt; they trace the data provenance, including biological and clinical aspects.

B. BASIC OPERATORS

After illustrating GMQL syntactic conventions, this section reports the list of all operators in GMQL along with their parameters, syntax and general usage. Each operator also contains a list of basic examples, showcasing how to combine different parameters and describing the associated semantics.

Within this document, by convention, operator parameters and conditions which are **mandatory** are written in **bold**.

Foreword: Syntactic conventions and other observations

- **Region** and **metadata attribute names**, as well as **dataset names**, are **case sensitive**: for instance, `pvalue != pValue != PVALUE`. **GMQL keywords**, however, are **not case sensitive**: e.g., `UPSTREAM == upstream == UpStReAm`.
- Assigning different values to an existing variable (i.e., dataset) name is not allowed by the language; a new dataset must be created instead.
- **Logical predicates on region or metadata** consist of concatenations of atomic predicates by means of the OR, AND and NOT logical operators. Atomic predicates have the following general form: `attribute_name (==, or >, or <, or >=, or <=) value`. If the value is a numeric literal, it is automatically parsed to the related numeric format. Standard numeric ordering is used in order to evaluate `>`, `<`; otherwise, if the attribute value is a string literal, a lexicographic order is used.
- The following are all the **aggregate functions** available for GMQL operators:
 - COUNT (requires no argument, counts number of regions, and is computed by default in the MAP operation). Note that this is not available in the GROUP operator, where instead the COUNTSAMP aggregate function is available (it requires no argument, counts number of samples, and is applicable only to create a new metadata with such count);
 - BAG and BAGD (applicable to attributes of any type, create comma-separated strings of attribute values, *distinct* in the case of BAGD);
 - SUM, AVG (average), MIN, MAX, MEDIAN, STD (standard deviation) (applicable to attributes of numeric types).
- In all operators that allow for **metadata comparisons** (i.e., those that include *groupby* or *joinby* options), the language recognizes dot-separated suffix **substrings of metadata (or region) attribute names** (for instance, 'age' in 'DS1.age'). So, if a metadata selection or meta-join is made, the language searches for all metadata which contain the queried attribute name substring with the queried attribute value, e.g., the query `age == '45'` selects samples with metadata 'DS1.age 45' or 'P1.DS1.age 45'.
- In a metajoin condition (i.e., for all operators that include such condition: SELECT has *semijoin*, DIFFERENCE, MAP, and JOIN have *joinby*, GROUP, MERGE, and COVER have *groupby*), different matching options can be used:
 - `metadata_attribute_name`: it matches all attributes that are equal to **OR** end with the dot-separated suffix specified name (regardless additional `metadata_attribute_name` dot-separated prefixes not explicitly specified);
 - `EXACT(metadata_attribute_name)`: it matches all attributes that are equal to the specified name (without any prefixes);
 - `FULL(metadata_attribute_name)`: it matches two attributes if they end with the specified name **AND** their full names are equal;For instance, if we consider the following attributes:
 1. `pref1.pref2.att`
 2. `pref1.att`
 3. `att`
 4. `pref1.att`Then:
 - `att` matches all of the above attributes;
 - `EXACT(att)` matches only attribute 3. (i.e., `att`);
 - `FULL(att)` matches attributes 2. and 4. (i.e., `pref1.att`).

- In GMQL, query **comments** can be introduced, starting with the character #; all parts after the # along the same line are considered comments.
- **Note:** GMQL queries and scripts always require at least one SELECT and one MATERIALIZE statement in order to compile, therefore to execute.
- **Note:** the only operator in the current release which allows to **edit region coordinates** as region attributes is the PROJECT operator.
- **Note:** it is possible to use the PROJECT command to copy region coordinates into separate region attributes and employ them as any other attribute (including use in aggregations).
- When evaluating the effect of every GMQL operator, **all** the following **six characteristics** of the output dataset must be considered:
 1. **Number of samples** (called *dataset cardinality*), based on input dataset(s) cardinality;
 2. **Number of sample regions**;
 3. **Dataset schema**, i.e., region attributes;
 4. **Samples region coordinates**, based on input regions (also when these overlap in a single sample);
 5. **Samples region attribute values**;
 6. **Samples metadata**.

1) SELECT

The SELECT operation creates a new dataset from an existing one (considering also an additional dataset if a semijoin clause is specified, see below) by extracting a subset of samples and/or regions from the input dataset; each sample in the output dataset has the same region attributes, values, and metadata as in the input dataset.

The general syntax for SELECT is:

$DS_{out} = \text{SELECT}(p_m; \text{region: } p_r; \text{semijoin: } p_{sj}(DS_{ext})) DS_{in};$

where:

- DS_{in} is the input dataset;
- DS_{out} is the resulting output dataset;
- p_m is a logical predicate on metadata;
- p_r is a logical predicate on genomic regions within each sample in DS_{in} ;
- $p_{sj}(DS_{ext})$ is a semi-join predicate, with form: $attr_1, attr_2, \dots, attr_N$ **IN** (or **NOT IN**) DS_{ext} , where DS_{ext} is another dataset previously created.

This operation (hereafter called selection) can therefore be based on three kinds of criteria, of which at least one must be specified:

1. Metadata predicates: selection based on the existence and values of certain metadata attributes in each sample. For instance, $antibody_target == 'POLR2A'$ extracts only samples whose metadata contain the attribute $antibody_target$ with associated value $POLR2A$. In predicates, attribute-value conditions can be composed using logical predicates AND, OR and NOT; in the latter case attribute-value conditions must be within parentheses, e.g., $NOT(antibody_target == 'POLR2A')$;

2. Region predicates: selection based on the characteristics of the genomic regions of each sample. For instance, *strand* == + extracts only samples that include regions with *strand* attribute (defined in the dataset schema) *equal* to + and only those regions whose strand is *equal* to +. Notice that the use of metadata attributes in predicates on region attributes is enabled. For instance, a condition such as *AccIndex* == *META(maxCount)* can be used, where *AccIndex* is a region attribute and *maxCount* is a metadata attribute;
3. Semi-join clauses: selection based on the existence of certain metadata attributes and the matching of their values with those associated with at least one sample in an external dataset DS_{ext} . In particular, a semi-join predicate in the form $a_1, a_2, \dots, a_N \text{ IN } DS_{ext}$ is true for a given sample s_{in}^k of DS_{in} if and only if there exists at least one sample in dataset DS_{ext} with metadata attributes named a_1, a_2, \dots, a_N and these attributes of DS_{ext} have at least one value in common with the same attributes a_1, a_2, \dots, a_N in s_{in}^k . For instance, *semijoin: cell, antibody_target IN CTCF_RAW* extracts only those samples of DS_{in} that have both cell and antibody_target values found in at least one sample of CTCF_RAW. NOT IN condition is evaluated accordingly.

Clearly, a user might define complex SELECT statements with more than one selection type at the same time; if this happens, it is intended that clauses of heterogeneous type are connected by an AND condition.

Note 1: SELECT() DS_{in} selects all samples in dataset DS_{in} and copies them in the output.

Note 2: The wildcard character '*' can be used in a SELECT statement to indicate all values of an attribute, e.g., SELECT(NOT(*attribute_name* == '*')) DS_{in} selects all samples in dataset DS_{in} which do not include in their metadata the attribute named *attribute_name* (with any value) and copies such samples in the output.

Note 3: As mentioned in the *Foreword* section, in *semijoin* option (which is one of the possible *metajoin* options of GMQL) different alternatives are available with respect to dot-separated prefixes in case present for metadata attribute names:

- metadata_attribute_name;
- EXACT(metadata_attribute_name);
- FULL(metadata_attribute_name).

Please refer to the [Foreword](#) section of this document for further details.

Example 1:

```
DATA = SELECT(region: (chr == chr1 OR chr == chr2) AND
                NOT(strand == + OR strand == -) AND start >= 500000
                AND stop <= 600000) HG19_ENCODE_NARROW;
```

This GMQL statement restricts the set of selected regions in each sample of the input HG19_ENCODE_NARROW dataset to the ones belonging to either chromosome *chr1* or chromosome *chr2* (note that, since *chr* is a region coordinate attribute, its value must be specified without quotes). The syntax *NOT(strand == + OR strand == -)* can be used to include regions whose strand is unknown, i.e., neither positive nor negative (i.e., unstranded regions). Note that, when specifying particular values for the attribute *strand*, quotes should not be used. The last two conditions restrict the selected regions to the ones that extend along the genomic interval [500000, 600000].

If no regions of a samples are selected, the sample is not included in the output RES dataset.

Example 2:



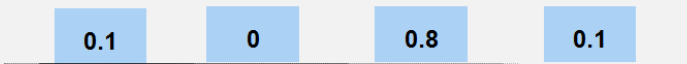
OUTPUT_DATASET = SELECT(Patient_age < 70) INPUT_DATASET;

This GMQL query selects from INPUT_DATASET data samples of patients younger than 70 years old, based on filtering on sample metadata attribute *Patient_age* (see following figure).

INPUT_DATASET:

0.1	0.6	Tumor_type = brca Patient_age = 75		
0.5	0	Tumor_type = brca Patient_age = 63 Sex = Female		
0.1	0	0.8	0.1	Tumor_type = brca Patient_age = 35

OUTPUT_DATASET:

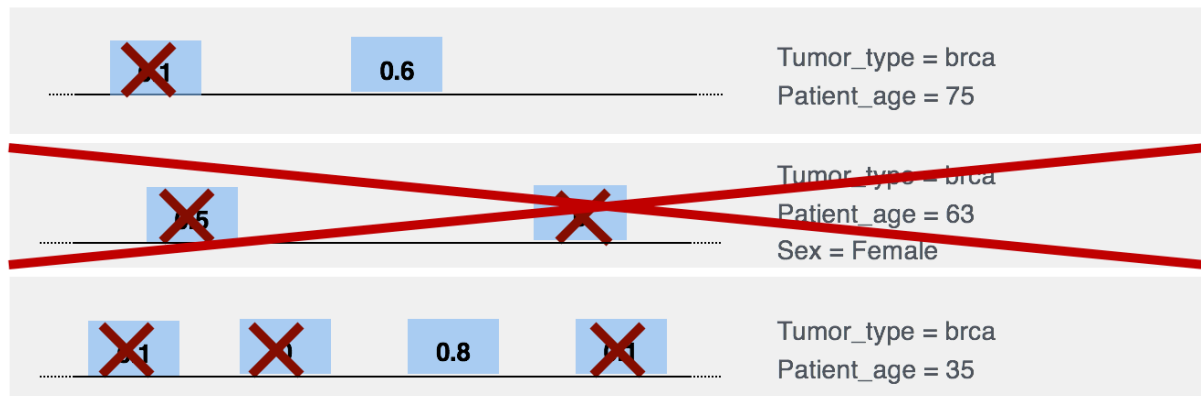
		Tumor_type = brca Patient_age = 75		
		Tumor_type = brca Patient_age = 63 Sex = Female		
				Tumor_type = brca Patient_age = 35

Example 3:

OUTPUT_DATASET = SELECT(region: score > 0.5) INPUT_DATASET;

This query selects, in all samples in INPUT_DATASET, those regions which have a value greater than 0.5 for their attribute *score*. The resulting OUTPUT_DATASET contains a copy of the samples of INPUT-DATASET, with the same metadata, but with only the remaining regions. When, for a specific sample, no regions that satisfy the condition are selected, that sample is not included in the output.

OUTPUT_DATASET:



Example 4:

```
DATA = SELECT(cell == 'Urothelia'; region: left > 100000) HG19_ENCODE_NARROW;
```

This GMQL statement creates a new output dataset DATA which only includes samples from the input dataset HG19_ENCODE_NARROW that present the metadata attribute-value pair (*cell* *Urothelia*). Moreover, in each sample, only the regions whose left coordinate value is greater than 100000 are included in the output dataset DATA. If no regions of a sample are selected, that sample is not included in the output.

Example 5:

```
DATA = SELECT(region: NOT(variant_type == 'SNP')) HG19_TCGA_dnaseq;
```

This statement produces a dataset that contains all the samples of the input dataset (with all their metadata) which have at least one region which is not of *variant_type* 'SNP'. Inside each sample the regions that, for the region attribute *variant_type*, have a value different from "SNP" (Single-Nucleotide Polymorphism) are preserved; the regions that have *variant_type* 'SNP', instead, are excluded.

Note that in case all regions belonging to a sample have been excluded through the NOT condition, that empty sample is not produced in the output dataset.

Example 6:

```
DATA = SELECT(manually_curated__tissue_status == "tumoral" AND
              (manually_curated__tumor_tag == "gbm" OR
               manually_curated__tumor_tag == "brca")) HG19_TCGA_dnaseq;
```

This statement shows how it is possible to combine multiple conditions on the metadata attributes by using the Boolean operators AND and OR. In this particular case, the output dataset contains all the samples that have *manually_curated__tissue_status* = "tumoral" and *manually_curated__tumor_tag* = "gbm", and also all the samples that have *manually_curated__tissue_status* = "tumoral" and *manually_curated__tumor_tag* = "brca". Notice that *gbm* corresponds to data concerning patients with *Glioblastoma multiforme*, instead *brca* refers to *Breast Invasive Carcinoma*.

Example 7:

```
JUN_POLR2A_TF = SELECT(antibody_target == 'JUN'; region: pvalue < 0.01;  
    semijoin: cell NOT IN POLR2A_TF) HG19_ENCODE_NARROW;
```

This statement creates a new dataset called JUN_POLR2A_TF by selecting those samples and their regions from the existing HG19_ENCODE_NARROW dataset (a collection of narrowPeak data from the ENCODE repository) such that:

- A. each output sample has a metadata attribute called *antibody_target* with value *JUN*;
- B. each output sample also has not a metadata attribute called *cell* that has the same value of at least one of the values that a metadata attribute equally called *cell* has in at least one sample of the POLR2A_TF dataset;
- C. for each sample satisfying A and B, only its regions that have a region attribute called *pvalue* with the associated value less than *0.01* are conserved in output; only samples with at least one conserved region are selected in the output.

Example 8:

```
DATA = SELECT(region: score > META(avg_score)) HG19_ENCODE_NARROW;
```

This statement allows to select, in all input sample, all those regions for which the region attribute *score* has a value which is greater than the metadata attribute value *avg_score* in their sample.

2) MATERIALIZE

The MATERIALIZE operation saves the content of a dataset in a file and registers the saved dataset in the system to make it usable in other GMQL queries. The name of the file must be specified and be different from the name of any other file saved in the same query.

The general syntax for MATERIALIZE is the following:

```
MATERIALIZE DS INTO file-name;
```

where:

- *DS* is the dataset (temporary and local to the query) to be saved on the file system;
- *file-name* is the required name of the file into which the dataset *DS* must be saved.

Note 1: In a GMQL script or query a MATERIALIZE statement is always necessary in order to compile/execute it. Only in this way a result of the computation becomes visible and available for download.

Note 2: The actual GMQL implementation materializes *DS* into a file with a name in the form [queryname]_[timestamp]_filename.

All datasets defined in a GMQL query are, by default, temporary; to store and access the content of any dataset generated during a GMQL query such dataset must be materialized. Any dataset can be materialized; however, the operation is time expensive, so for better performance it is suggested to materialize only relevant datasets, such as the final output.

Example:

MATERIALIZE HM_TFS INTO res;

This GMQL statement saves the content of the temporary HM_TFS dataset into a file named *[queryname]_[timestamp]_res*.

3) PROJECT

The PROJECT operator creates, from an existing dataset, a new dataset with all the samples (with their regions and region values) in the input one, but keeping for each sample in the input dataset only those metadata and/or region attributes expressed in the operator parameter list. Region coordinates and values of the remaining metadata and region attributes remain equal to those in the input dataset. Differently from the SELECT operator, PROJECT allows to:

- Remove existing metadata and/or region attributes from a dataset;
- Create new metadata and/or region attributes to be added to the result.

The general syntax for PROJECT is:

```
DSout = PROJECT(RA1, ..., RAm;  
                metadata: MA1, ..., MAn;  
                region_update: NR1 AS g1, ..., NRh AS gh;  
                metadata_update: NM1 AS f1, ..., NMk AS fk) DSin;
```

where:

- DS_{in} is the input dataset;
- DS_{out} is the resulting output dataset;
- RA₁, ..., RA_m are the conserved genomic region attributes;
- MA₁, ..., MA_n are the conserved metadata attributes;
- NR₁, ..., NR_h are new genomic region attributes generated using functions g₁, ..., g_h on existing region or metadata attributes or constant;
- NM₁, ..., NM_k are new metadata attributes generated using functions f₁, ..., f_k on existing metadata attributes or constant.

Note 1: The default form of this operator has no parameter. PROJECT() DS_{in} applies the projection only on the regions. It removes all the region attributes which are not coordinates (i.e., only *chr*, *start*, *stop*, and *strand* are kept).

Note 2: It is possible to use the special keywords ALLBUT to retain all existing genomic region or metadata attributes apart from a specified set.

Note 3: If the names of existing region or metadata attributes are used in place of new region names, the operation updates such region attributes to the new specified values.

To specify the new values, the following options are available:

- All basic mathematical operations (+, -, *, /), including usage of parenthesis;
- The square root mathematical function (i.e., SQRT(attribute_name));
- Whenever possible, the metadata values are cast to numeric. If the cast fails (i.e., the metadata value is a not castable string) the resulting metadata should contain "GMQL Casting Exception: Could not parse".

Note 4: To express which set of region or metadata attributes should be considered, the wildcard "?" can be used in the RA_i place of the syntax (at most one per attribute). For instance, the user can write statements such as:

```
OUTPUT_DATASET = PROJECT(?..score) INPUT_DATASET;  
OUTPUT_DATASET = PROJECT(?..score, ?..name) INPUT_DATASET;  
OUTPUT_DATASET = PROJECT(DS.?) INPUT_DATASET;  
OUTPUT_DATASET = PROJECT(my.?.score) INPUT_DATASET;  
OUTPUT_DATASET = PROJECT(S.?, ?.att, S.?.att) INPUT_DATASET;  
Note that PROJECT(?..S.?) is incorrect.
```

Note 5: It is possible to create a new textual region or metadata attribute with a defined value, e.g., `OUTPUT_DATASET = PROJECT(region_update: label AS "class1") INPUT_DATASET;`

Note 6: It is possible to define a new numeric region attribute with "null" value. The syntax for creating a new attribute with null value is *attribute_name AS NULL(TYPE)*, where type may be INTEGER or DOUBLE. As an example, we can write statements such as `OUTPUT_DATASET = PROJECT(region_update: signal AS NULL(INTEGER), pvalue AS NULL(DOUBLE)) INPUT_DATASET;`

This feature is useful to extend the schema of a dataset before its composition, through the UNION() operator, with another dataset, so that all attributes of the two datasets are included in the UNION() output dataset (which is defined to have the same schema of the UNION() left input dataset). Notice that a null value for the type STRING does not exist, which is instead defined as the empty string (i.e., "").

Note 7: It is possible to define a new region attribute with the value of a metadata attribute using the syntax *region_attribute_name AS META(metadata_attribute, type)*, e.g., `OUTPUT_DATASET = PROJECT(region_update: signal AS META(avg_signal, DOUBLE)) INPUT_DATASET;`

Notice that the type of the new region attribute has to be specified; it can be INTEGER or DOUBLE if the metadata attribute has numeric values, STRING otherwise.

Example 1:

```
OUTPUT_DATASET = PROJECT(region_update: length AS right - left) INPUT_DATASET;
```

This GMQL statement creates a new dataset called OUTPUT_DATASET by preserving all region attributes and creating a new region attribute called *length* with value obtained by subtracting the left coordinate value of a region from its right coordinate value. This simple operation computes the length of the region in terms of number of bases. Notice that the length is always positive regardless of the strand of the region, because *right* and *left* coordinates already take into account the direction. Also notice that, in case the INPUT_DATASET is a dataset containing single base regions (e.g., TSS), all the new length attributes would turn out to be unitary.

Example 2:

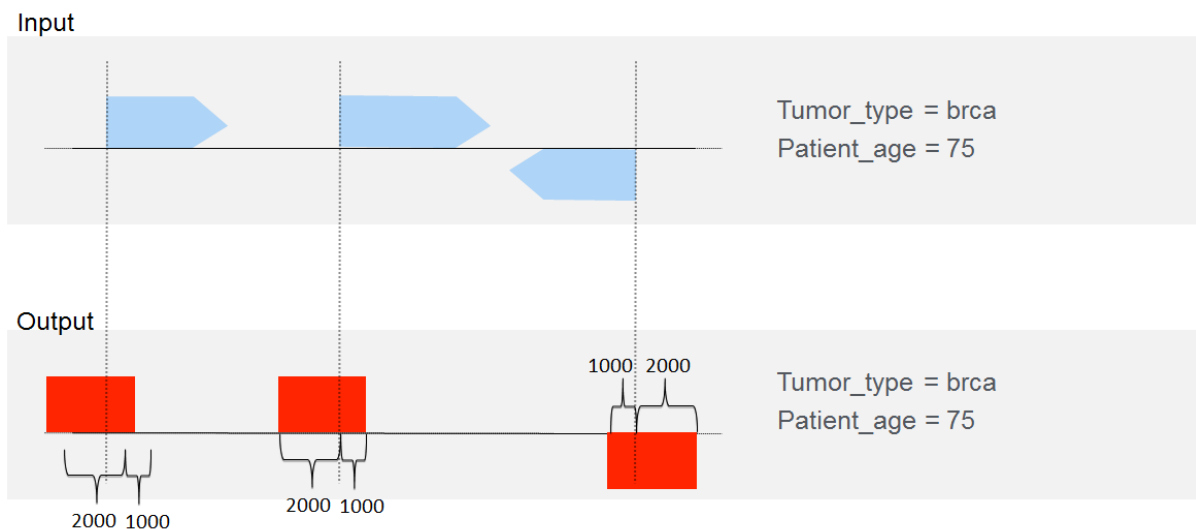
```
OUTPUT_DATASET = PROJECT(region_update: new_right AS right) INPUT_DATASET;
```

This GMQL statement creates a new dataset called `OUTPUT_DATASET` by preserving all region attributes and creating a new region attribute called *new_right* which contains a copy of the value of the coordinate attribute *right*. This allows to subsequently aggregate regions by their right coordinate value using the *new_right* attribute.

Example 3:

```
RES = PROJECT(region_update: start AS start - 2000, stop AS start + 1000) GENES;
```

This `PROJECT` statement considers an input dataset of genes and generates in output a dataset `RES` with the gene promoter regions. To define a promotorial region, it is necessary to start from a transcription start site (TSS) (a single base of DNA at the beginning of the gene transcription region, conventionally taken as a starting point for the gene transcription) and extend it upstream/downstream by a number of given bases. As an example, here, the region coordinate attributes *left* and *right* are redefined by shifting the upstream one of them 2 kbp upstream and the downstream one 1 kbp downstream from the original upstream one, by using the *start/stop* option that takes the region strand into account.



Example 4:

```
OUTPUT_DATASET = PROJECT(metadata: ALLBUT cell, cell_sex) INPUT_DATASET;
```

This example shows how to use the `ALLBUT` option to exclude multiple metadata attributes, retaining all the others not specified. It creates a new dataset `OUTPUT_DATASET` by preserving all region attributes and metadata attributes, apart from the *cell* and *cell_sex* ones, which are excluded from the metadata attributes of all the samples.

Example 5:

```
CTCF_NORM_SCORE = PROJECT(ALLBUT score, value; region_update: new_score AS  
    (score / 1000.0) + 100; metadata_update: normalized AS 1) CTCF_RAW;
```

This GMQL statement creates a new dataset called CTCF_NORM_SCORE by preserving all region attributes apart from *score* and *value*, and creating a new region attribute called *new_score* by dividing the existing score value of each region by 1000.0 and incrementing it by 100.

It also generates, for each sample of the new dataset, a new metadata attribute called *normalized* with value 1, which can be used in future selections.

Example 6:

```
DS_out = PROJECT(variant_classification, variant_type;  
    metadata: manually_curated__tissue_status, manually_curated__tumor_tag) DS_in;
```

This statement produces an output dataset DS_out that contains the same samples as the input dataset DS_in. Each output sample only preserves, as region attributes, the four basic coordinates (*chr*, *left*, *right*, *strand*) and the specified region attributes *variant_classification* and *variant_type*, and as metadata attributes only the specified ones, i.e., *manually_curated__tissue_status* and *manually_curated__tumor_tag*.

Example 7:

```
DS_out1 = PROJECT(metadata_update: age AS age + 10) DS_in;  
DS_out2 = PROJECT(metadata_update: age_plus AS age + 100) DS_in;
```

The first statement produces an output dataset DS_out1 that contains the same samples as the input dataset DS_in. Each output sample contains the same region attributes as the samples in DS_in. As metadata attributes, each sample contains the same ones as in DS_in samples with the exception of the attribute *age*, which is incremented by 10.

The second statement has the same effect, but instead of substituting the value of the *age* attribute, it adds the *age_plus* attribute, which corresponds to the former *age* attribute incremented by 100.

Example 8:

```
DS_out = PROJECT(region_update: qvalue AS NULL(DOUBLE),  
    peak AS NULL(INTEGER), other AS NULL(DOUBLE)) DS_in;
```

This example shows how to write a correct PROJECT statement which adds to all samples of the input dataset DS_in the region attributes *qvalue*, *peak* and *other* of the specified type. Those of these attributes that do not exist in the input dataset are added to the schema of the DS_out dataset and initialized with a NULL value in all regions of all samples of the RES dataset; those of these attributes that already exist in the input dataset are set to the specified type (in case changing the original one) and their values are set to NULL in all regions of all samples of the DS_out dataset.

Example 9:

```
DS_out = PROJECT(region_update: signalSq AS SQRT(signal);  
                  metadata_update: concSq AS SQRT(concentration)) DS_in;
```

This statement allows to build an output dataset `DS_out` such that all the samples from the input dataset `DS_in` are conserved, as well as their region attributes (and their values) and their metadata attributes (and their values). The new region attribute *signalSq* is added to the output schema and to all the output samples with value correspondent to the mathematical squared root of the pre-existing region attribute *signal*. In addition, the new metadata attribute *concSq* is added to all output samples with value correspondent to the mathematical squared root of the pre-existing metadata attribute *concentration*.

Example 10:

```
DS_out = PROJECT(region_update: sampleID AS META(ID, INTEGER),  
                  score AS META(avg_score, DOUBLE), cell AS META(cell, STRING)) DS_in;
```

As Example 9, this statement allows to build an output dataset `DS_out` such that all the samples from the input dataset `DS_in` are conserved, as well as their region attributes (and their values) and their metadata attributes (and their values). The new region attributes *sampleID*, *score*, and *cell* are added to the schema with the specified type (INTEGER, DOUBLE and STRING, respectively); for all regions of a sample their value are equal to the value of the indicated metadata attributes of the sample (respectively: *ID*, *avg_score*, and *cell*).

Example 11:

```
DS_out = PROJECT(region_update: chr1 AS chr, start1 AS start, stop1 AS stop,  
                  strand1 AS strand) DS_in;
```

This GMQL statement creates a new dataset called `DS_out` equal to the input dataset `DS_in`, but in addition with the four new region attributes called *chr1*, *start1*, *stop1*, and *strand1*, which contain respectively copies of the values of the coordinate attributes *chr*, *start*, *stop* and *strand*. This allows to subsequently aggregate regions by their coordinate values using these new attributes (note that aggregating to the original coordinate attributes is not allowed).

Example 12:

```
DS_out = PROJECT() DS_in;
```

This GMQL statement creates a new dataset `DS_out` equal to the input dataset `DS_in`, but with the only difference that it keeps only the coordinates of every region, while all other region attributes are removed from the `DS_out` dataset schema (and their values from all `DS_out` dataset samples).

Example 13:

```
DS_out = PROJECT(metadata_update: newID AS (ID * 100), newInfo AS SQRT(Info)) DS_in;
```

This GMQL statement creates a new dataset `DS_out` equal to the input dataset `DS_in`. In addition, in the metadata of `DS_out` samples, it adds two new metadata: *newID*, which yields the value of the existing metadata attribute *ID* multiplied by a factor of 100 (the new value is of type double), and *newInfo*, which instead contains the special value “GMQL Casting

Exception: Could not parse” since it is derived from a non-numerical field which cannot be casted (in order to perform the numerical operation of computing its squared root).

4) EXTEND

For each sample in an input dataset, the EXTEND operator builds new metadata attributes, assigns their values as the result of arithmetic and/or aggregate functions calculated on sample region attributes, and adds them to the existing metadata attribute-value pairs of the sample. Sample number and their genomic regions, with their attributes and values, remain unchanged in the output dataset.

The general syntax for EXTEND is:

$DS_{out} = \text{EXTEND}(NM_1 \text{ AS } g_1, \dots, NM_k \text{ AS } g_k) DS_{in};$

where:

- DS_{in} is the input dataset whose sample region attribute values are used to compute the new sample metadata;
- DS_{out} is the output dataset, a copy of the input dataset with additional metadata calculated by EXTEND;
- $NM_1, \dots, NM_k;$ are new metadata attributes generated using arithmetic and/or aggregate functions g_1, \dots, g_k on the sample region attributes in DS_{in} . In addition to the usual aggregate functions, additional ones are available: $q1(\text{region_attribute})$, $q2(\text{region_attribute})$, $q3(\text{region_attribute})$, which are respectively the first, second, and third quartile of the values of the specified *region_attribute*.


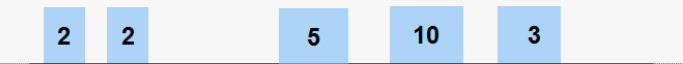

Note: EXTEND does not have a default form (the statement $\text{EXTEND}() DS_{in}$ does not compile); at least one parameter is required.

Example 1:

$\text{RES} = \text{EXTEND}(\text{Region_count AS COUNT}()) \text{EXP};$

This GMQL statement counts the regions in each sample of the input dataset EXP and stores their number as value of the new metadata *Region_count* attribute of the sample in the output dataset RES.

RES:

	Tumor_type = brca Patient_age = 75 Region_count = 3
	Tumor_type = esca Patient_age = 78 Region_count = 5
	Tumor_type = chol Patient_age = 85 Region_count = 2

Example 2:

```
RES = EXTEND(region_count AS COUNT(), min_pvalue AS MIN(pvalue)) EXP;
```

This GMQL statement copies all samples of the EXP dataset into the RES dataset, and then calculates two new metadata attributes for each of them:

1. *Region_count* is the number of sample regions;
2. *min_pvalue* is the minimum *pvalue* of the sample regions.

RES sample regions are the same as the ones in EXP.

Example 3:

```
RES = EXTEND(all_scores AS BAG(score)) EXP;
```

This GMQL statement copies all samples of EXP into RES dataset, and then for each of them adds another metadata attribute, *all_scores*, which is the aggregation comma-separated list of all the values (or only the distinct ones in the case of using BAGD() instead of BAG()) that the attribute *score* takes in the sample regions.

Example 4:

```
RES = EXTEND(quant1 AS q1(score)) EXP;
```

This statement copies all the samples of the EXP dataset into the RES dataset and, for each of them, it adds an additional metadata attribute *quant1*, calculated as the first quartile value of the sample's score distribution.

5) ORDER

The ORDER operator is used to order either samples, sample regions, or both, in a dataset according to a set of metadata and/or region attributes, and/or region coordinates. The number of samples and their regions in the output dataset is as in the input dataset, as well as their metadata and region attributes and values, but a new ordering metadata and/or region attribute is added with the sample or region ordering value, respectively.

The general syntax of ORDER is the following:

```
DSout = ORDER(MA1 DESC, ..., MAn DESC;  
              meta_top: k OR meta_topg: k OR meta_topg: k;  
              region_order: RA1 DESC, ..., RAm DESC;  
              region_top: k OR region_topg: k OR region_topg: k) DSin;
```

where:

- *DS_{in}* is the input dataset;
- *DS_{out}* is the output sorted dataset;
- *MA₁, ..., MA_n* are the ordering metadata attributes;
- *RA₁, ..., RA_m* are the ordering genomic region attributes;
- DESC is an optional parameter to be set after each ordering attribute that reverses the ordering with respect to that attribute (default is ascending, becomes descending);

- k , where specified, is the number (or percentage, in the case of `meta_topp` and `region_topp`) of samples (or regions) to be extracted from the ordered dataset (or from each sample), starting from the top (with respect to the final ascending/descending ordering).

Sorted samples, or sample regions, have a new attribute `_order` added to either metadata, regions, or both of them; the value of `_order` reflects the result of the sorting.

The clauses `meta_top: k` and `region_top: k` extract the first k samples and regions, respectively, according to the final ordering.

The clauses `meta_topp: k` and `region_topp: k` specify that the first $k\%$ elements (either samples or regions) are extracted, out of the total number of elements, according to the final ordering; the integer number of elements extracted is obtained by the floor integer approximation of the $k\%$ elements (where $\text{floor}(x)$ is the greatest integer that is less than or equal to x , i.e., in case of 0.99 elements, 0 elements are extracted).

The clauses `meta_topg: k` and `region_topg: k` implicitly consider the ordering defined by first grouping identical values of the first $n - 1$ ordering attributes, and then sorted by the remaining attributes; they then select the first k samples, or regions, of each group.

Note 1: ORDER does not have a default form (the statement `ORDER() DSin` does not compile); at least an ordering metadata attribute or a `region_order` clause must be specified.

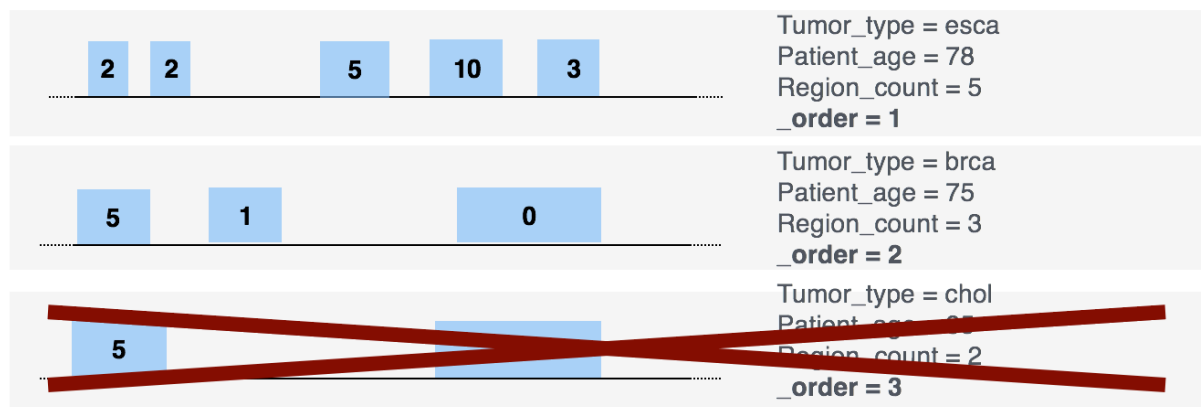
Note 2: When the specified ordering metadata attributes are not present in any of the input samples, an empty output is generated.

Example 1:

```
OUTPUT_DS = ORDER(Region_count DESC; meta_top: 2;) INPUT_DS;
```

This GMQL statement orders the samples in the `INPUT_DS` dataset according to their `Region_count` metadata attribute and takes the two samples that have the highest count. As shown in the following figure, the sample with attribute `_order = 3` is excluded from the output.

OUTPUT_DS:



Example 2:

```
OUTPUT_DS = ORDER(Region_count; meta_top: 5;
                  region_order: Mutation_count DESC; region_top: 7) INPUT_DS;
```


This GMQL statement extracts the first 5 samples on the basis of their region counter (those with the smaller *Region_count*) and then, for each of them, 7 regions on the basis of their mutation counter (those with the higher *Mutation_count*).

Example 3:

```
OUTPUT_DS = ORDER(treatment_type, ID DESC; meta_top: 2) INPUT_DS;
```

This GMQL statement first sorts the samples in INPUT_DS dataset by ascending order with respect to their metadata *treatment_type*, then it sorts them by descending order based on the values of their metadata *ID* attribute (the new metadata attribute *_order* is added to all samples). Finally, only the samples with *_order* = 1 or *_order* = 2 are extracted in the output dataset OUTPUT_DS.

Example 4:

```
OUTPUT_DS = ORDER(target_label DESC; meta_top: 30) INPUT_DS;
```

This GMQL statement first sorts the samples in INPUT_DS dataset by descending order with respect to their *target_label* metadata attribute; then it extracts in the OUTPUT_DS dataset the top 30% of the ordered samples.

By assuming that the INPUT_DS dataset contains 100 samples, the statement returns 30 samples of the INPUT_DS dataset that correspond to the first 30 ones according to the descending order by their *target_label* metadata value (these samples are labelled with the metadata attribute *_order* = 1, *_order* = 2, ..., *_order* = 30).

Example 5:

```
OUTPUT_DS = ORDER(region_order: score; region_top: 50) INPUT_DS;
```

This statement first sorts, in each sample, the regions according to their score. Then, for each sample, it only preserves 50% of the regions (specifically, the first half of the regions, which are ordered according to ascending score).

Example 6:

```
OUTPUT_DS = ORDER(cell, treatment_type; meta_top: 1; region_order: pvalue, length, name; region_top: 2) INPUT_DS;
```

This GMQL statement groups INPUT_DS dataset samples by their metadata attribute *cell* and then, for each group, it selects only the first sample, according to the ascending order of the metadata attribute *treatment_type*.

In addition, inside each sample of the INPUT_DS dataset, regions are ordered according to their ascending *pvalue* and *length* order; then, for each order group, the statement only outputs the first two regions in each sample based on ascending order of the attribute *name*. Note that the region attribute *order* is added to the schema of the output dataset. This new region attribute equals to 1 in the regions that are first (by ascending order of attribute *name*) and 2 in the regions that are second (by the same order).

6) GROUP

The GROUP operator can be used on metadata and on regions.

When used on metadata attributes, GROUP performs the grouping of samples of the input dataset based on one specified metadata attribute. If the metadata attribute is multi-value, i.e., it assumes multiple values for sample (e.g., both `<disease, cancer>` and `<disease, diabetes>`), the grouping identifies different groups of samples for each attribute value combination (e.g., group1 for samples that feature the combination `<disease, cancer>`, group2 for samples that feature the combination `<disease, diabetes>`, and group3 for samples that feature both combinations `<disease, cancer>` and `<disease, diabetes>`).

For each obtained group, it is possible to request the evaluation of aggregate functions on metadata attributes; these functions consider the metadata contained in all samples of the group. The regions, their attributes and their values in output are the same as the ones in input for each sample, and the total number of samples does not change. All metadata in the input samples are conserved with their values in the output samples, with the addition of the `_group` attribute, whose value is the identifier of the group to which the specific sample is assigned; other metadata attributes can be added as aggregate functions computed on specified metadata.

When used on region attributes, GROUP can group regions of each sample individually, based on their coordinates (`chr`, `start`, `stop`, `strand`) and possibly also on other specified grouping region attributes (when these are present in the schema of the input dataset). In each sample, regions found in the same group (i.e., regions with same coordinates and grouping attribute values), are combined into a single region; this allows to merge regions that are duplicated inside the same sample (based on the values of their coordinates and of other possible specified region attributes). For each grouped region, it is possible to request the evaluation of aggregate functions on other region attributes (i.e., which are not coordinates, or grouping region attributes). This use is independent on the possible grouping realised based on metadata. The generated output schema only contains the original region attributes on which the grouping has been based, and additionally the attributes in case calculated as aggregated functions.

If the GROUP is applied only on regions, the output metadata and their values are equal to the ones in input.

Both when applied on metadata and on regions, the GROUP operation returns a number of output samples equal to the number of input ones.

Note that the two possible uses of GROUP, on metadata and on regions, are perfectly orthogonal, therefore they can be used in combination or independently.

The general syntax for GROUP is:

```
DSout = GROUP(MA;  
               meta_aggregates: GM1 AS f1, ..., GMk AS fk;  
               region_keys: RA1, ..., RAm;  
               region_aggregates: GR1 AS g1, ..., GRh AS gh) DSin;
```

where:

- DS_{in} is the input dataset;
- DS_{out} is the output dataset;
- MA is the grouping metadata attribute;
- GM_1, \dots, GM_k are new metadata attributes generated using aggregate functions f_1, \dots, f_k on k metadata attributes in DS_{in} ;
- RA_1, \dots, RA_m are the grouping genomic region attributes, in addition to the implicit default `chr`, `left`, `right`, `strand` attributes;
- GR_1, \dots, GR_h are new region attributes generated using aggregate functions g_1, \dots, g_h on h region attributes in DS_{in} .

Note that `meta_aggregates` can be specified only if a grouping metadata attribute is specified, while `region_aggregates` can be specified even when the `region_keys` parameter is not used, since this is equivalent to grouping only on the four coordinates *chr*, *left*, *right*, *strand*.

Several observations can be made on the effect of GROUP:

- Should a grouping attribute be multi-valued, samples are partitioned by each subset of their distinct values (e.g., samples with a Disease attribute set both to 'Cancer' and 'Diabetes' are within a group which is distinct from the groups of the samples with only one value, either 'Cancer' or 'Diabetes').
- Samples having missing values for any of the grouping metadata attributes are assigned all together to one group, identified by the special value `_group = 0`.
- It is not possible to perform GROUP based on multiple metadata attributes.
- When grouping applies to regions, by default it includes as grouping attributes the region coordinates *chr*, *left*, *right*, *strand*. This choice corresponds to the procedure of merging duplicate regions, i.e., regions with the same coordinates within the same sample, ensuring that the result is a legal GMQL sample.

Note 1: The default form of this operator has no parameter. `GROUP() DSin` applies the grouping only on the region attributes which represent the four genomic coordinates, i.e., *chr*, *start*, *stop*, and *strand*. Inside a single sample, it collapses all regions that have equal values in these four coordinates into a single one, thus eliminating duplicate regions.

Note 2: The option `region_keys` accepts as parameters only region attributes that are not region coordinates. When used, it always implicitly considers, preceding the list of specified attributes, the 4 region coordinates. This means that a grouping is always performed on these coordinates before grouping on additional region attributes.

Note 3: As mentioned in the *Foreword* section, for the grouping metadata attributes in the option (which is one of the possible *metajoin* options of GMQL) different alternatives are available with respect to dot-separated prefixes in case present for metadata attribute names:

- `metadata_attribute_name`;
- `EXACT(metadata_attribute_name)`
- `FULL(metadata_attribute_name)`.

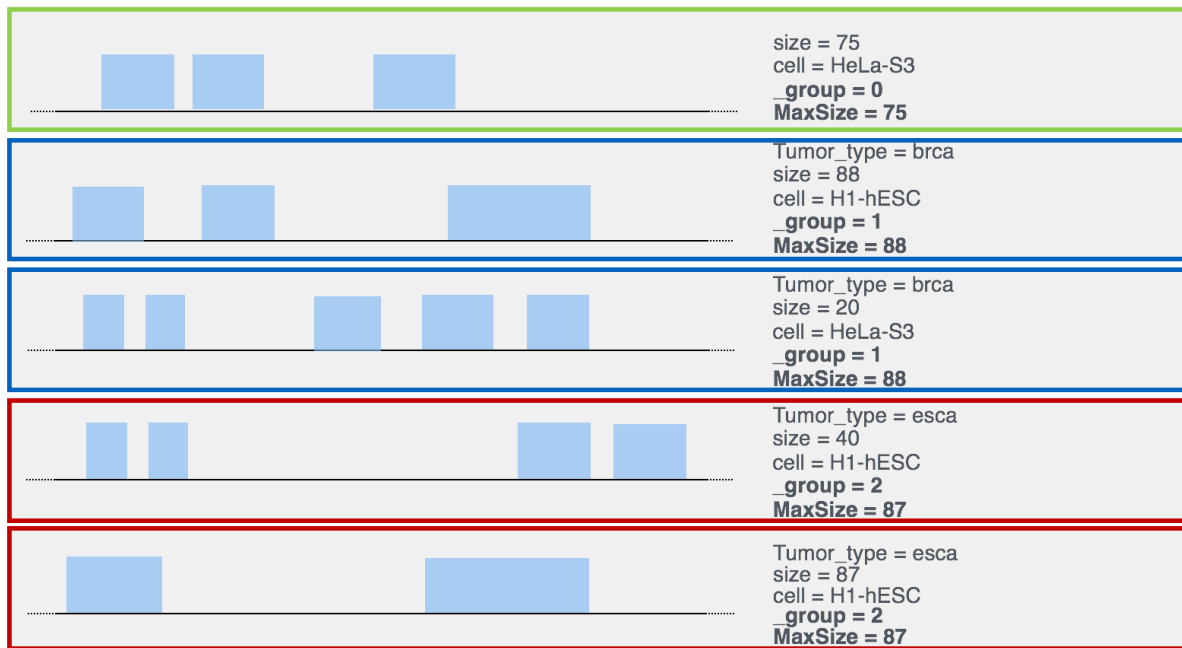
Please refer to the [Foreword](#) section of this document for further details.

Example 1:

```
GROUPS_T = GROUP(tumor_type; meta_aggregates: MaxSize AS MAX(size)) EXP;
```

This GMQL statement groups samples of the input EXP dataset according to their value of the metadata attribute *tumor_type* and computes the maximum value that the metadata attribute *size* takes inside the samples belonging to each group. The samples in the output GROUPS_T dataset have a new `_group` metadata attribute which indicates which group they belong to, based on the grouping on the metadata attribute *tumor_type*. In addition, they present the new metadata aggregate attribute *MaxSize*. Note that the samples without metadata attribute *tumor_type* are assigned to a single group with `_group` value equal 0.

GROUPS_T:

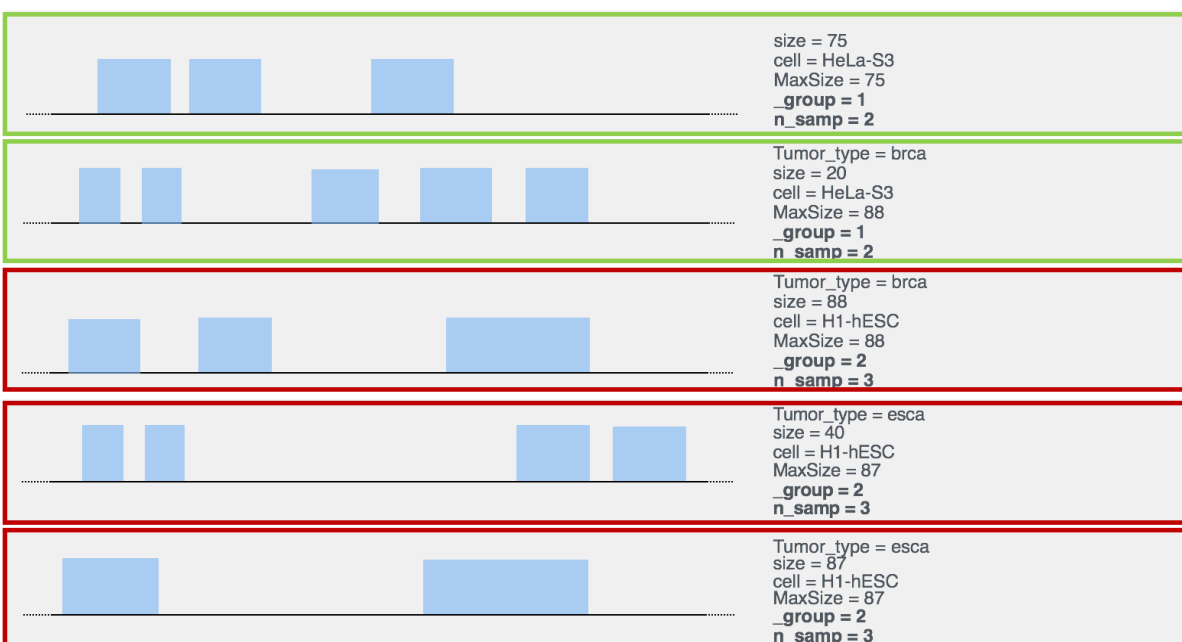


Example 2:

GROUPS_C = GROUP(cell; meta_aggregates: n_samp AS COUNTSAMP()) EXP;

This GMQL statement takes as input dataset the same input dataset as the previous example. Yet, it calculates new *_group* values based on the grouping attribute *cell*, and adds the metadata aggregate attribute *n_samp*, which counts the number of samples belonging to the respective group. It has the following output GROUPS_C dataset samples (note that now no sample has metadata attribute *_group* with value equal 0 since all input samples include the metadata attribute *cell*, with different values, on which the new grouping is based):

GROUPS_C:



Example 3:

GROUPS = GROUP(region_aggregates: regNum AS COUNT()) EXP;

In each individual EXP input dataset sample, this GMQL statement groups the sample regions by their coordinates *chr*, *left*, *right*, *strand* (which are assumed implicitly), and keeps only one region for each group (i.e., a single region with the same coordinates). This behavior corresponds to eliminating duplicated regions in the same sample. In the output dataset schema, the new region attribute *regNum* is added, computed as the number of regions that have the same coordinates for each region group within an individual input sample, and the computed value is assigned to each region of each output sample. All other region attributes, which are not coordinates, are discarded.

Supposing the input dataset contained only one sample which presented the following regions, with schema *<chr, left, right, strand, name, score>*:

chr1	1007265	1009661	-	RNF223	1
chr1	1007265	1009661	-	C1orf159	2
chr1	1076223	1078563	+	RP11-465B22.5	3
chr1	1102494	1102577	+	MIR200B	4
chr1	1102494	1102577	+	MIR200A	5

the output dataset GROUPS would contain, similarly, only one sample, with schema *<chr, left, right, strand, regNum>* and the following regions:

chr1	1007265	1009661	-	2
chr1	1076223	1078563	+	1
chr1	1102494	1102577	+	2

As it can be observed, only the 4 coordinates are preserved (*name* and *score* attributes are discarded). The additional attribute *regNum* is added. Its value is 2 when the relative region has been computed by grouping two regions from the initial sample (e.g., the region chr1, 1007265, 1009661, -), while it is 1 when the relative region only appeared once in the input sample.

Example 4:

GROUPS = GROUP(region_keys: score;
region_aggregates: avg_pvalue AS AVG(pvalue), max_qvalue AS MAX(qvalue)) EXP;

This GMQL statement groups the regions of each EXP dataset sample by region coordinates *chr*, *left*, *right*, *strand* (these are implicitly considered) and the additional region attribute *score* (which is explicitly specified), and keeps only one region for each group. In the output GROUPS dataset schema, the new region attributes *avg_pvalue* and *max_qvalue* are added, respectively computed as the average of the values taken by the *pvalue* and the maximum of the values taken by the *qvalue* region attributes in the regions grouped together, and the computed value is assigned to each region of each output sample. Note that the region attributes which are not coordinates or *score* are discarded.

Example 5:

GROUPS = GROUP(cell_tissue; meta_aggregates: min_tier AS MIN(cell_tier);
region_aggregates: min_signal AS MIN(signal)) EXP;

This GMQL statement shows how the GROUP operator can be used on both metadata and regions at the same time. In this case, it first groups the samples of the EXP dataset by metadata attribute *cell_tissue* values, and adds to each sample the attribute *_group* to indicate which group it belongs to. Then, it calculates the minimum value of the metadata attribute *cell_tier* over the samples that are part of a same group and adds this to all samples as value of the new metadata attribute *min_tier*.

Inside each sample, it groups the regions based on their coordinates (implicitly considered, without the need of using the *region_keys* option); for each region group it keeps only one region, and calculates the new region attribute *min_signal* as the minimum of the values taken by the region attribute *signal* in each region group.

7) MERGE

The MERGE operator builds a new dataset consisting of a single sample having:

- as regions, all the regions of all the input samples, with the same attributes and values;
- as metadata, the union of all the metadata attribute-values of the input samples.

A *groupby* clause can be specified on metadata: the samples are then partitioned in groups, each with a distinct value of the grouping metadata attributes, and the MERGE operation is applied to each group separately, yielding to one sample in the result dataset for each group. Samples without the grouping metadata attributes are disregarded.

The general syntax for MERGE is:

$DS_{out} = \text{MERGE}(\text{groupby: } M_1, \dots, M_n) DS_{in};$

where:

- DS_{in} is the input dataset to be merged;
- DS_{out} is the output dataset;
- M_1, \dots, M_n are the (optional) metadata attributes used in the *groupby* clause (see below).

Note 1: As mentioned in the *Foreword* section, in *groupby* option (which is one of the possible *metajoin* options of GMQL) different alternatives are available with respect to dot-separated prefixes in case present for metadata attribute names:

- *metadata_attribute_name*;
- EXACT(*metadata_attribute_name*);
- FULL(*metadata_attribute_name*).

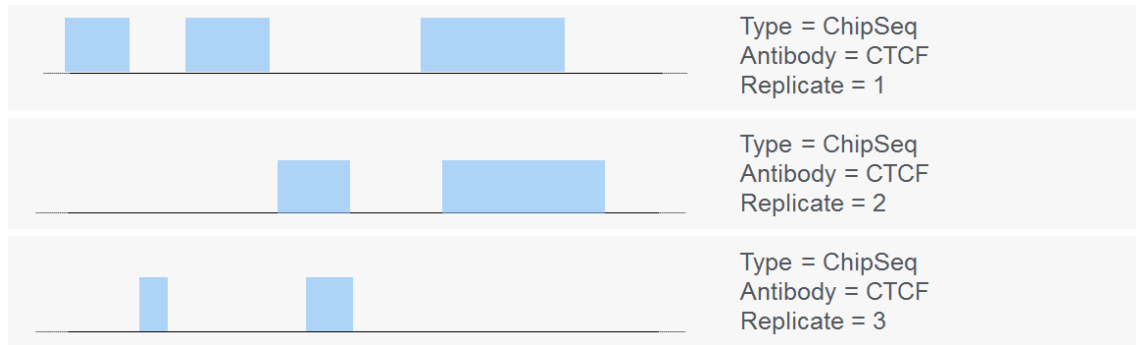
Please refer to the [Foreword](#) section of this document for further details.

Example 1:

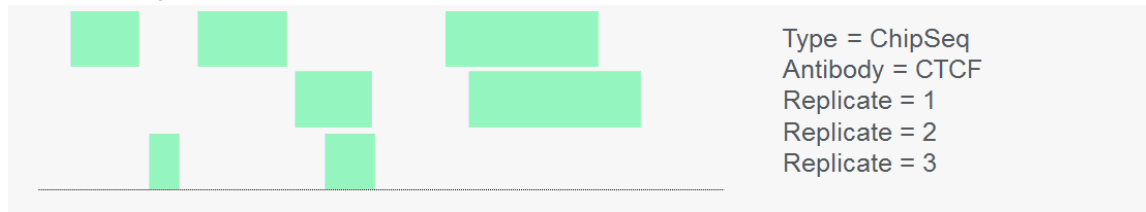
MERGED_ALL = MERGE() INIT_DATA;

This GMQL statement collapses a bunch of samples (both regions and metadata) into a single one. More in detail, it creates a new dataset MERGED_ALL consisting of a single sample having as regions all the regions in the INIT_DATA dataset, with the same attributes and values, and as metadata the union of all the metadata attribute values of the samples of INIT_DATA.

For instance, we may have this INIT_DATA input dataset:



And would get this MERGED_ALL result:



Example 2:

```
MERGED = MERGE(groupby: antibody_target) EXPERIMENT;
```

This GMQL statement creates a dataset called MERGED, which contains one sample for each `antibody_target` value found within the metadata of the EXPERIMENT dataset samples; each created sample contains all regions from all EXPERIMENT samples with the same specific value for their `antibody_target` metadata attribute.

8) UNION

The UNION operation is used to integrate homogeneous or heterogeneous samples of two datasets within a single dataset; for each sample of either one of the two input datasets, a sample is created in the result dataset as follows:

- its metadata attributes and values are the same as in the original sample;
- its schema is the schema of the first (left) input dataset; new identifiers are assigned to each output sample;
- its regions are the same in coordinates and attribute values as in the original sample if it is from the first (left) input dataset; if it is from the second (right) input dataset, its regions are the same in coordinates, but only region attributes identical (in name and type) to those of the first input dataset are retained, with the same values. Region attributes which are missing in the second input dataset sample (w.r.t. the merged schema) are set to null.

The general syntax for UNION is:

```
DSout = UNION() DS1 DS2;
```

where:

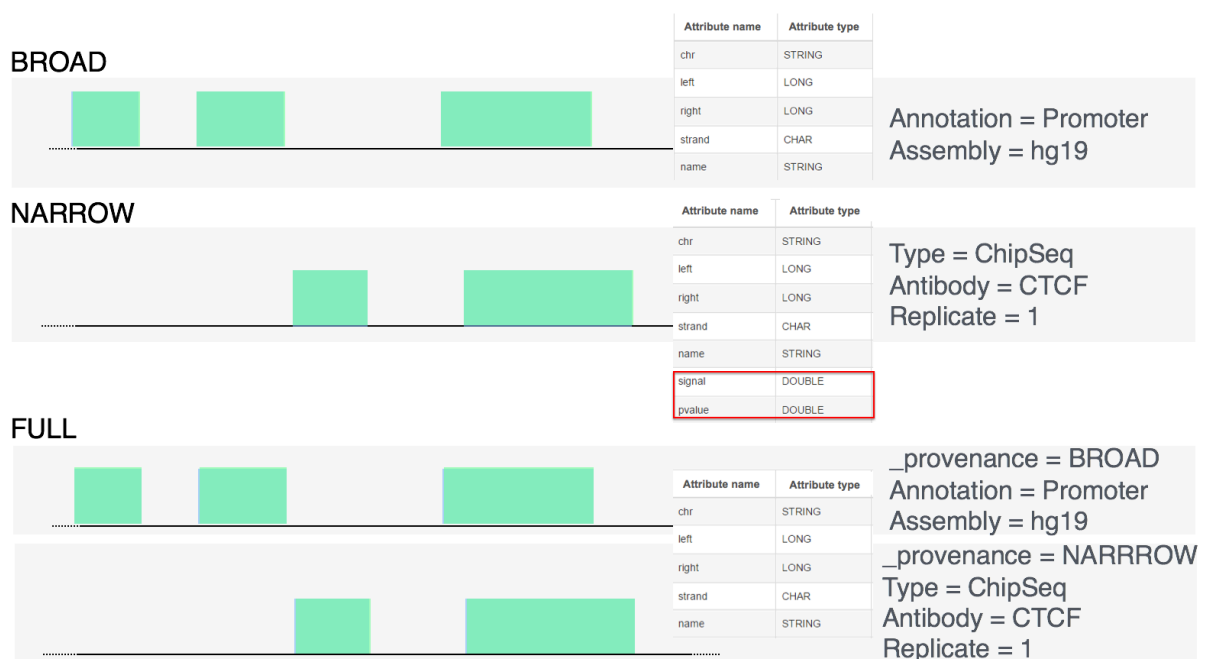
- DS_1 and DS_2 are the input datasets to be unified;
- DS_{out} is the unified output dataset.

The union of the two schemas is performed by taking only the schema of the first dataset, removing the region attributes of the second dataset which are not identical to those of the first dataset; two region attributes are considered identical if they have the same name and type. To avoid losing region attributes from the second dataset, before the union new attributes can be added to the first dataset by using the PROJECT operator; the new attributes must be identical in name and type to those present only in the second dataset, and must be added with null value (in the case of attributes of numerical type) or empty string value (in the case of attributes of type STRING). For what concerns metadata, attributes of samples from either the first or second input dataset are enriched with an additional *_provenance* attribute with value the name of one of the two input datasets, so as to trace the dataset to which they originally belonged.

Example:

FULL = UNION() BROAD NARROW;

This statement creates a dataset called FULL which contains all samples from the datasets BROAD and NARROW (that could include broadPeak and narrowPeaks data samples from ENCODE experiments), whose schema is defined by merging BROAD and NARROW dataset schemas (union of all the attributes present in both the two input datasets).



9) DIFFERENCE

DIFFERENCE is a binary, non-symmetric operator that produces one sample in the result for each sample of the first operand (left input dataset), by keeping the same metadata attributes and values of the first operand sample and only those regions (with their attributes and values) of the first operand sample which do not intersect with any region in the second operand (right

input dataset) sample (also known as *negative regions*). The general syntax for DIFFERENCE is:

$DS_{out} = \text{DIFFERENCE}(\text{exact: true; joinby: } M_1, \dots, M_n) DS_{ref} DS_{neg};$

where:

- DS_{ref} is the *reference* dataset, i.e., the dataset which is copied in the output and from which regions of DS_{neg} are “subtracted”;
- DS_{neg} is the *negative* dataset, i.e., the dataset whose regions are checked for intersection against the reference regions. If any reference region is found to have intersection (or exact coordinate matching, in case of specifying the *exact: true* option) with a region in DS_{neg} , it is removed from the output dataset;
- DS_{out} is the output dataset;
- M_1, \dots, M_n are the (optional) metadata attributes used in the joinby clause.

The optional **joinby** clause is used to extract subsets of samples on which to apply the DIFFERENCE operator: only those samples $s_1 \in DS_{ref}$ and $s_2 \in DS_{neg}$ that have the same value for each attribute M_1 through M_n are considered when performing the DIFFERENCE.

Note 1: DIFFERENCE operates in two different modes based on region intersection: the default behavior (i.e., $\text{DIFFERENCE}() DS_{ref} DS_{neg}$), and the exact matching (i.e., $\text{DIFFERENCE}(\text{exact: true}) DS_{ref} DS_{neg}$). In the second case, only regions in the first dataset whose coordinates do not exactly match the coordinates of any region in the second dataset are kept in the output dataset.

Note 2: If all regions of a sample in the first input dataset intersect (match, if the exact option is used) at least a region in the second input dataset, the sample is not included in the output dataset.

Note 3: As mentioned in the *Foreword* section, in *joinby* option (which is one of the possible *metajoin* options of GMQL) different alternatives are available with respect to dot-separated prefixes in case present for metadata attribute names:

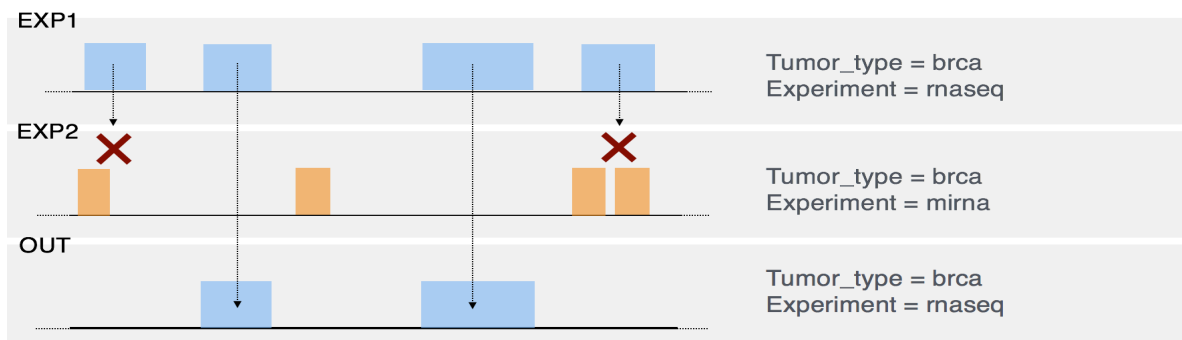
- metadata_attribute_name;
- EXACT(metadata_attribute_name);
- FULL(metadata_attribute_name).

Please refer to the [Foreword](#) section of this document for further details.

Example 1:

$\text{OUT} = \text{DIFFERENCE}() \text{EXP1 EXP2};$

This GMQL statement returns all the regions in the first dataset that do not overlap any region in the second dataset.



Example 2:

```
RES = DIFFERENCE(exact: true) EXP1 EXP2;
```

This statement extracts all regions in the first input dataset EXP1 that do not coincide (exactly from the start to the end coordinate) with at least a region in the second input dataset EXP2.

Example 3:

```
OUT = DIFFERENCE(joinby: antibody_target) EXP1 EXP2;
```

This GMQL statement performs the DIFFERENCE operation considering subsets of samples that have the same value for the metadata attribute *antibody_target*; indeed, only those samples $s_i \in \text{EXP1}$ and $s_j \in \text{EXP2}$ that have the same value of *antibody_target* are compared. For every different value of *antibody_target*, the statement allows to extract all the regions of EXP1 samples, with an *antibody_target* value, that do not intersect any of the regions of EXP2 samples with the same *antibody_target* value.

10) MAP

MAP is a non-symmetric operator over two datasets, respectively called **reference** and **experiment**. The operation computes, for each sample in the experiment dataset, aggregates over the values of the experiment regions that intersect with a region in a reference sample, for each region of each sample in the reference dataset; we say that experiment regions are *mapped* to the reference regions. The number of generated output samples is the Cartesian product of the samples in the two input datasets; each output sample has the same regions as the related input reference sample, with their attributes and values, plus the attributes computed as aggregates over experiment region values. Output sample metadata are the union of the related input sample metadata, whose attribute names are prefixed with their input dataset name.

For each reference sample, the MAP operation produces a matrix like structure, called **genomic space**, where each experiment sample is associated with a row, each reference region with a column, and each matrix row is a vector of numbers - the aggregates computed during MAP execution. When the features of the reference regions are unknown, the MAP helps in extracting the most interesting regions out of many candidates.

The general syntax for MAP is:

```
DSout = MAP(NR1 AS g1, ..., NRh AS gh;  
            count_name: X;  
            joinby: MA1, ..., MAn) DSref DSexp;
```

where:

- DS_{ref} is the *reference* dataset;
- DS_{exp} is the *experiment* dataset;
- DS_{out} is the output dataset;
- NR_1, \dots, NR_h are new genomic region attributes (optionally) generated using functions g_1, \dots, g_h on existing experiment region attributes;

- X is an optional preference name given by the user to the metadata attribute which corresponds to the number of each experiment sample region intersecting a certain reference region;
- MA_1, \dots, MA_n are the (optional) metadata attributes used in the *joinby* clause (see below).

Note 1: In each reference sample, multiple regions with exactly the same coordinates and attribute values are managed as a single region.

Note 2: The COUNT() aggregate (counting the number of each experiment sample region intersecting a certain reference region) is always computed; results are stored, by default, in an attribute named *count_[DSrefName]_[DSexpName]*, where *DSrefName* and *DSexpName* are the names of DS_{ref} and DS_{exp} , respectively. To rename the default name of this attribute to a custom name, e.g., *myCountName*, use the following syntax: $DS_{out} = \text{MAP}(\text{count_name: myCountName}) DS_{ref} DS_{exp}$; (Please note that in case together with *count_name* you like to calculate new region attributes, according to the MAP() syntax you have to specify the latter ones as first predicate of the MAP(), e.g., $DS_{out} = \text{MAP}(\text{avg_score AS AVG(score); count_name: myCountName}) DS_{ref} DS_{exp}$;

Note 3: No parameter is mandatory in the MAP operator. The default behavior with syntax $\text{MAP}() DS_{ref} DS_{exp}$ performs the operation without adding any new region attributes (besides the always computed default one (see Note 1) with the number of each experiment sample region intersecting a given reference region), and in its computation it compares all samples of the reference dataset DS_{ref} with all samples of the experiment dataset DS_{exp} .

Note 4: As mentioned in the *Foreword* section, in *joinby* option (which is one of the possible *metajoin* options of GMQL) different alternatives are available with respect to dot-separated prefixes in case present for metadata attribute names:

- *metadata_attribute_name*;
- EXACT(*metadata_attribute_name*);
- FULL(*metadata_attribute_name*).

Please refer to the [Foreword](#) section of this document for further details.

We first describe the effect of the basic MAP operation (without **joinby** clause). Let:

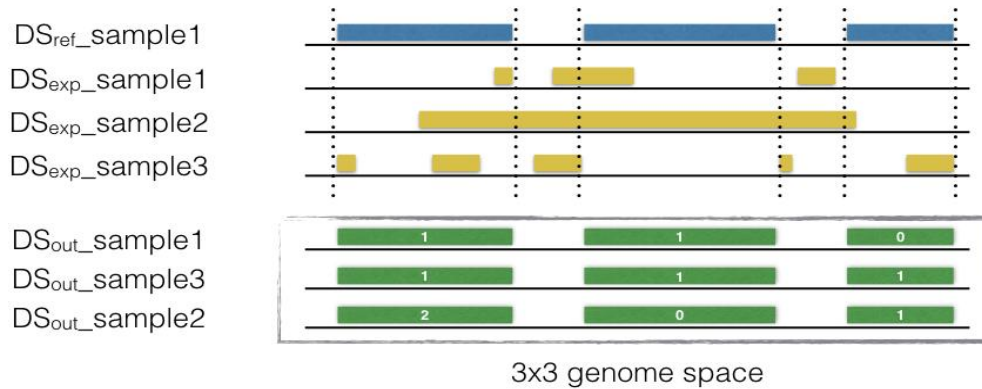
- $s_1 \in DS_{ref}$ be a given reference sample with R_1 the set of its regions and M_1 its metadata;
- s_2 be a generic sample of the experiment dataset DS_{exp} with R_2 the set of its regions and M_2 its metadata.

A new sample s_3 is constructed as follows:

- the metadata M_3 are obtained by merging metadata M_1 and M_2 , taking track of their provenance by prefixing their attribute names with the name of their original dataset;
- the regions R_3 are created such that, for each region $r_1 \in R_1$, there is exactly an equal region $r_3 \in R_3$, with the same coordinates and having as attributes the attributes of r_1 and optionally, in addition, the new attributes computed by the aggregate functions g_i specified in the operation; such aggregate functions are applied to the attributes of all the regions $r_2 \in R_2$ having a non-empty intersection with r_1 .

The operation is iterated for each experiment samples and each reference sample, and it generates a reference sample-specific genomic space at each reference sample iteration.

When the **joinby** clause is present, only pairs of samples s_1 of DS_{ref} and s_2 of DS_{exp} with metadata M_1 and M_2 that satisfy the *joinby* condition are considered. Syntactically, the clause consists of a list of metadata attribute names (or their suffixes) that must be present with equal values in both M_1 and M_2 (attribute names specified in the *joinby* clause can also refer to only the last suffix of actual attribute names in M_1 and M_2 for the $s_1 - s_2$ matches to be considered).

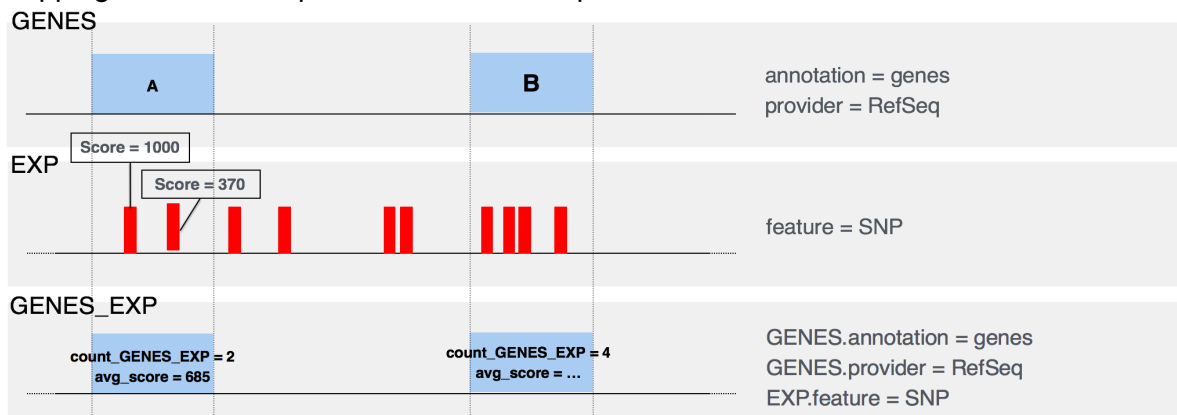


The above figure shows the result of the MAP operation (with no *joinby* clause) on a small portion of the genome. The input consists of one reference sample with 3 regions, in the DS_{ref} dataset, and three experiment samples in the DS_{exp} dataset; the output consists of the DS_{out} dataset with three samples, each with the same regions as in the reference sample, which contain a feature called **count_DS_ref_DS_exp** (if not renamed), where DS_{ref} and DS_{exp} are the input dataset names counting the number of experiment regions which intersect with the specific reference region. The result can be interpreted as a (3×3) genome space.

Example 1:

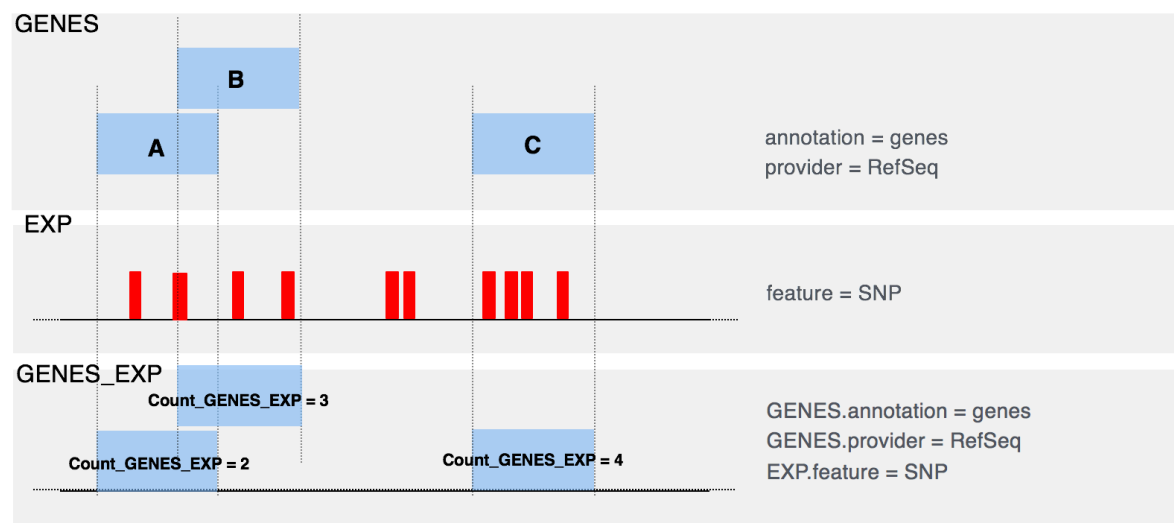
GENES_EXP = MAP(avg_score AS AVG(score)) GENES EXP;

Given a dataset GENES, containing a single sample with a known set of gene regions, and another dataset EXP containing genomic region results from a genomic experiment on the same species, this GMQL statement counts the number of regions in each sample from the experiment which overlap with a known gene, saving results in the output GENES_EXP dataset as a region attribute named *count_GENES_EXP*; it also computes the average (AVG) score value across such regions, saving results in the output GENES_EXP dataset as a region attribute (feature) called *avg_score*. If GENES dataset contains multiple samples, the number of samples in the output GENES_EXP dataset is the Cartesian product of the number of samples in the GENES and EXP datasets, and each of the output samples represents the mapping of a EXP sample on a GENES sample.



Observation:

Notice that when a reference sample include (partially) overlapping regions, all these regions are included in the result regions, as it can be seen in the following figure.



Example 2:

OUT = MAP (minScore AS MIN(score); count_name: reg_num; joinby: cell_tissue) REF EXP;

This GMQL statement counts the number of regions in each sample from EXP that overlap with a REF region, saving results in output as a region attribute *reg_num*, and for each REF region it computes also the minimum *score* of all the regions in each EXP sample that overlap with it. The MAP *joinby* option ensures that only the EXP samples referring to the same *cell_tissue* of a REF sample are mapped on such REF sample; EXP samples with no *cell_tissue* metadata attribute, or with such metadata but with a different value from the one(s) of REF sample(s), are disregarded.

11) JOIN

The JOIN operator takes in input two datasets, respectively known as *anchor* (the first/left one) and *experiment* (the second/right one) and returns a dataset of samples consisting of regions extracted from the operands according to the specified conditions (known as *equi predicate* and *genometric predicate*). The number of generated output samples is the Cartesian product of the number of samples in the anchor and in the experiment dataset (if no *joinby* clause is specified). The region attributes (and their values) in the output dataset are the union of the region attributes (with their values) in the input datasets, unless the RIGHT_DISTINCT, LEFT_DISTINCT, or BOTH option is specified (see later). Homonymous attributes are disambiguated by prefixing their name with their dataset name. The output metadata are the union of the input metadata with their attribute names prefixed with their input dataset name, unless the RIGHT_DISTINCT or LEFT_DISTINCT option is specified (see later); in this latter case the output metadata are equal to the metadata of the input *experiment* (second/right) or *anchor* (first/left) dataset sample, respectively (without prefixing their attribute names).

The general syntax for JOIN is the following:

$DS_{out} = \text{JOIN}(\text{genometric_predicate};$
 on_attributes: RA_1, \dots, RA_m ;
 output: coord-param;
 joinby: MA_1, \dots, MA_n) $DS_{anc} DS_{exp}$;

where:

- DS_{anc} and DS_{exp} are respectively the *anchor* and *experiment* datasets;
- DS_{out} is the output dataset;
- *genometric_predicate* is an optional concatenation of *distal* conditions by means of logical ANDs (see later for details);
- RA_1, \dots, RA_m are the (optional) region attributes used in the *equi predicate* clause, i.e., region attributes which must exist in both input datasets and whose values in the experiment dataset region must be equal to their values in the anchor dataset region for the experiment region to be considered;
- *coord-param* is one of four different values that declare which region is given in output for each input pair of anchor and experiment regions satisfying the equi predicate and the genometric predicate:
 - LEFT outputs the anchor regions from DS_{anc} that satisfy the equi predicate and the genometric predicate. LEFT can be postfixed by the keyword `_DISTINCT` which calls for the duplicate elimination of DS_{anc} output regions with the same values, regardless the DS_{exp} paired region and its values. In this case, the output regions attributes and their values are all those of DS_{anc} , and the output metadata are equal to the DS_{anc} metadata, without additional prefixes;
 - RIGHT outputs the experiment regions from DS_{exp} that satisfy the equi predicate and the genometric predicate. RIGHT can be postfixed by the keyword `_DISTINCT` which calls for the duplicate elimination of DS_{exp} output regions with the same values, regardless the DS_{anc} paired region and its values. In this case, the output regions attributes and their values are all those of DS_{exp} , and the output metadata are equal to the DS_{exp} metadata, without additional prefixes;
 - INT outputs the overlapping part (intersection) of the anchor and experiment regions that satisfy the equi predicate and the genometric predicate; if the intersection is empty, no output is produced;
 - CAT outputs the concatenation between the anchor and experiment regions that satisfy the equi predicate and the genometric predicate, i.e., the output region is defined as having *left (right)* coordinates equal to the minimum (maximum) of the corresponding coordinate values in the anchor and experiment regions satisfying the equi predicate and the genometric predicate;
 - BOTH outputs the same regions as LEFT, but it adds in the output region attributes the coordinates of the DS_{exp} region that, together with the output DS_{anc} region, satisfies the equi predicate and the genometric predicate;
- MA_1, \dots, MA_n are the (optional) metadata attributes used in the *joinby* clause (see below).

The *joinby* condition (also called *meta-join* predicate) is used to select sample pairs satisfying certain conditions on their metadata (e.g., regarding the same cell line or antibody target); syntactically, it is expressed as a list of metadata attributes whose names and values must match between samples in DS_{anc} and DS_{exp} in order for such samples to verify the condition and be considered for the join.

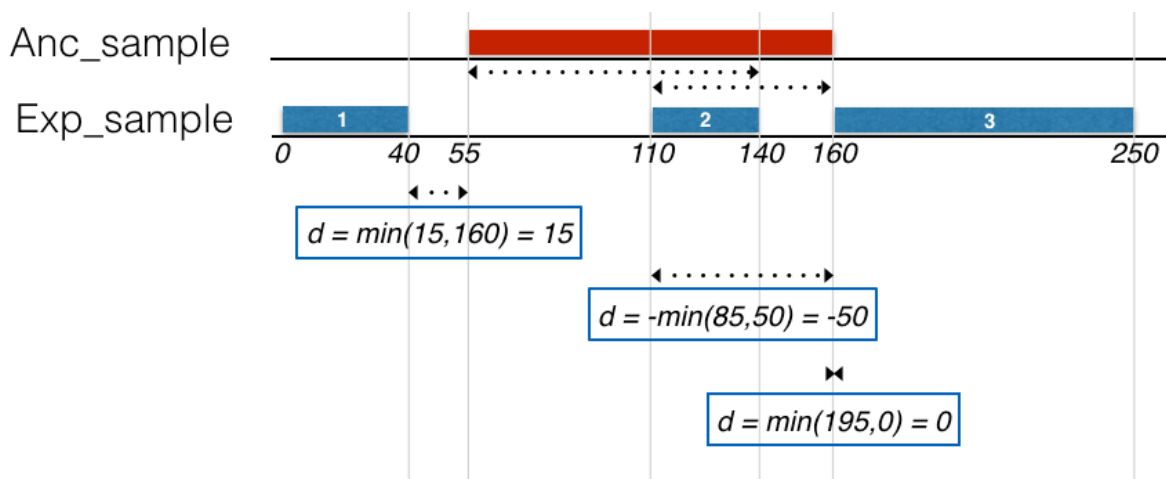
As mentioned in the *Foreword* section, in *joinby* option (which is one of the possible *metajoin* options of GMQL) different alternatives are available with respect to dot-separated prefixes in case present for metadata attribute names:

- `metadata_attribute_name`;
- `EXACT(metadata_attribute_name)`;
- `FULL(metadata_attribute_name)`.

Please refer to the [Foreword](#) section of this document for further details.

Genometric predicates are fundamental for JOIN commands: they allow the expression of a variety of distal conditions all based on the concept of **genomic distance**. The genomic distance is defined as the number of base pairs (i.e., nucleotides) between the closest opposite ends of two regions on the same DNA strand, or when at least one of the two regions has unknown strand, and belonging to the same chromosome (it is not defined for regions on different chromosomes or different DNA strands). More formally, considering r_1 as the anchor region and r_2 as the experiment region, their distance is calculated as $\min(\text{abs}(r_1.\text{left} - r_2.\text{right}), \text{abs}(r_2.\text{left} - r_1.\text{right}))$. If r_1 and r_2 overlap, then it is returned as the negative number of the above definition.

Note: In the GMQL framework, overlapping regions have negative distance while adjacent regions have distance equal to 0. In the following picture three possible cases of distance calculation are shown. In particular, let us consider the experiment blue region marked with 1, with left coordinate 0 and right coordinate 40 and the anchor red region with left coordinate 55 and right coordinate 160; their relative distance is calculated as: $\min(\text{abs}(55 - 40), \text{abs}(0 - 160)) = \min(15, 160) = 15$. Let us consider, instead, the experiment blue region 2, which **overlaps** with the anchor red region. In this case the same genomic distance definition must be applied preceded by a minus sign. Thus, being the coordinates of the anchor region [55, 160] and the coordinates of the experiment region [110, 140], the distance is calculated as: $-\min(\text{abs}(55 - 140), \text{abs}(110 - 160)) = -\min(85, 50) = -50$. The calculation for the blue region 3 is similar to the one for the blue region 1 (not overlapping). In this case, it is adjacent to the anchor red region, from which it has distance equal to 0.



A genomic predicate is a sequence of distal conditions (i.e., evaluated using genomic distance) defined as follows:

- MD(K) (or MINDIST(K), MINDISTANCE(K)) denotes the *minimum distance clause*, which selects the first K regions of an experiment sample at minimal distance from an anchor region of an anchor dataset sample. In case of ties (i.e., regions at the same distance from the anchor region), all tied experiment regions are kept in the result, even if they would exceed the limit of K
- DLE(N) (or DIST \leq N, DISTANCE \leq N) denotes the *less-equal distance clause*, which selects all the regions of the experiment such that their distance from the anchor region is less than, or equal to, N bases. In particular, DLE(0) searches for experiment regions adjacent to, or overlapping, the anchor region. [Please refer carefully to the next paragraph for the cases DLE(N) with $N < 0$, regarding only overlapping regions.]
- DGE(N) (or DIST \geq N, DISTANCE \geq N) denotes the *greater-equal distance clause*, which selects all the regions of the experiment such that their distance from the anchor region is greater than, or equal to, N bases. [Please refer carefully to the next paragraph for the cases DGE(N) with $N < 0$.]
- DL(N) (or DIST $<$ N, DISTANCE $<$ N) and DG(N) (or DIST $>$ N, DISTANCE $>$ N) denote the *less and greater distance clauses*, which select all regions of the experiment such that the distance from the anchor region is less, or greater, than N bases, respectively. In particular, DL(0) selects experiment regions overlapping with an anchor region regardless of the amount of the overlapping. [Please refer carefully to the next paragraph for the cases DL(N) or DG(N) with $N < 0$, the former one regarding only overlapping regions.]

For a correct use of the distal conditions, it is essential to note that comparisons between the distance between two regions and the specified N parameter in the distal conditions are to be intended in the arithmetic meaning.

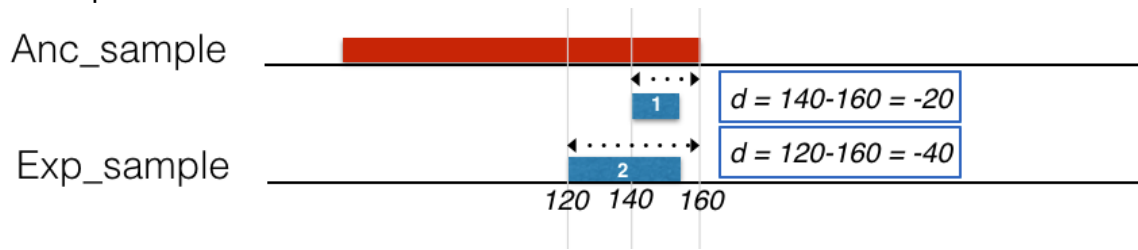
The use of these conditions is intuitive in case $N > 0$: let us suppose two regions have a relative distance of 30 (they do not overlap and the right-end of the left most region is 30 bp distant from the left-end of the right most region). If the user wishes to select them, he/she should either use a DLE(N) with $N \geq 30$ (e.g., DLE(40), which produces the inequality $30 \leq 40$), or DGE(N) with $N \leq 30$ (e.g., DGE(20), which produces the inequality $30 \geq 20$). Similar examples can be written using DL(N) and DG(N) conditions.

Using $N = 0$ in DLE(N) allows to select regions that are either adjacent (i.e., their relative distance is equal to zero) or overlapping in any possible way (i.e., their relative distance is negative). This is since DLE(0) produces the inequality “*some zero or negative distance*” ≤ 0 , which is arithmetically always true.

Conversely, $N = 0$ in DL(N) allows to select regions that are overlapping in any possible way (and are not adjacent). This is because DL(0) produces the inequality “*some negative distance*” < 0 , which is arithmetically always true.

The user must be particularly careful when using distal conditions in case he/she wishes to select only overlapping regions with a certain relative distance N. In this case, a negative N should be used. The idea is that **DLE(N) with $N < 0$** selects regions which are distant **at least** N base pairs one from the other in **absolute value**, i.e., $abs(d) \geq abs(N)$. Consider for example two overlapping regions with relative distance -40 and two other overlapping regions with relative distance -20. DLE(-30) selects the first ones since $abs(-40) \geq abs(-30)$, while

it discards the second ones since $abs(-20) \not\geq abs(-30)$. DGE(-30) would instead select the second pair and not the first one.



If we consider the above image, the evaluation of the conditions is explained in the following table:

Distal condition	Exp region 1	Selected	Exp region 2	Selected
DLE(-30)	$-20 \leq -30$	no	$-40 \leq -30$	yes
DGE(-30)	$-20 \geq -30$	yes	$-40 \geq -30$	no
DL(-20)	$-20 < -20$	no	$-40 < -20$	yes
DL(-40)	$-20 < -40$	no	$-40 < -40$	no
DG(-20)	$-20 > -20$	no	$-40 > -20$	no
DG(-40)	$-20 > -40$	yes	$-40 > -40$	no

An additional clause that can be specified in a genomic predicate is UP/DOWN (or UPSTREAM/DOWNSTREAM), called the *upstream/downstream clause*, which refers to the upstream or downstream directions of the genome. This clause requires that the rest of the predicate holds only on the upstream (downstream) genome with respect to the anchor region. More specifically:

- in the positive strand (or when the strand is unknown), UP is true for those regions of the experiment whose right-end is lower than, or equal to, the left-end of the anchor, and DOWN is true for those regions of the experiment whose left-end is higher than, or equal to, the right-end of the anchor;
- in the negative strand inequalities are exchanged;
- remaining regions of the experiment must be overlapping with the anchor region.

When this clause is not present, distal conditions apply to both directions of the genome indifferently.

Genomic clauses are strings composed of concatenations of at least one and at most four distal conditions including DLE (or DL), DGE (or DG), MD, UPSTREAM or DOWNSTREAM; we say that a genomic clause is *well-formed* if and only if it includes at least one less-equal distance, or one less distance, or a minimum distance clause. Genomic predicates (clauses) used in JOIN statements must be well-formed.

Examples:

The following strings are legal, well-formed, genomic predicates:

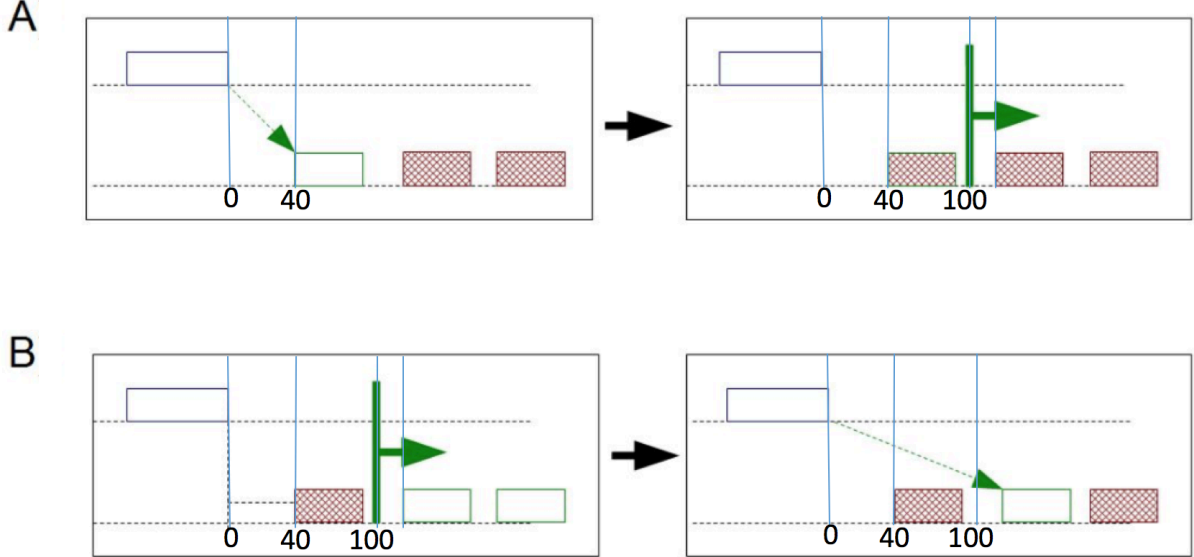
- DGE(500), UP, DLE(1000);
- MD(100), DG(3000);
- DL(2000), DOWN;
- DLE(300).

Note: Different orderings of the same distal clauses may produce different results. In the below figure, we show an evaluation of the following two clauses relative to an anchor region:

A. MD(1), DGE(100);

B. DGE(100), MD(1).

In case A, the MD(1) clause is computed first, producing one region which is next excluded by computing the DGE(100) clause; therefore, no region is produced as result. In case B, the DGE(100) clause is computed first, producing two regions, and then the MD(1) clause is computed, producing one region as result.



Similarly, the clauses:

A. MD(1), UP

B. UP, MD(1)

may produce different results: in case A, the minimum distance region is selected regardless of its up/down stream position to the anchor, and then it is retained if and only if it belongs to the upstream of the anchor, while in case B only upstream regions are considered, and the one at minimum distance is selected.

The join result is constructed as follows:

- The meta-join predicate initially selects all pairs s_i of DS_{anc} and s_j of DS_{exp} that satisfy the *joinby* condition. If the clause is omitted, then the complete Cartesian product between samples of DS_{anc} and of DS_{exp} is selected;
- For each such pair, a new sample s_{ij} is generated in the result, having metadata given by the union of metadata of s_i and s_j (unless the LEFT_DISTINCT, RIGHT_DISTINCT, or BOTH option is specified);
- The genomic predicate is tested for all the pairs (r_i, r_j) of regions, for each $r_i \in s_i$ and $r_j \in s_j$. This is done by giving (in turn) to each $r_i \in s_i$ the role of anchor region and then evaluating the genomic predicate condition with all the regions r_j of s_j .
- The equi predicate is tested for all the pairs (r_i, r_j) of regions, for each $r_i \in s_i$ and $r_j \in s_j$.
- For every pair (r_i, r_j) that satisfies the conjunction of the genomic and equi predicates, a new region is generated in s_{ij} , according to the *coord-param* value; the attributes and their values of the new region are all those of the r_i , and r_j regions (unless the LEFT_DISTINCT, RIGHT_DISTINCT, or BOTH option is specified), and homonymous attributes are disambiguated by prefixing their input dataset name to their name.

Note 1: By construction, the JOIN operation yields results whose number can grow quadratically both in the number of samples and of regions; hence, it is the most computationally intensive of all GMQL operations.

Note 2: The behavior of JOIN() without any argument is not defined; at least one of *genometric_predicate* or *on_attributes* conditions must be provided.

In case the user chooses to specify *genometric_predicate*, it must contain at least one and at most four distal conditions including DLE (or DL), DGE (or DG), MD, UPSTREAM or DOWNSTREAM, and it must include at least one *less-equal distance* or one *less distance*, or a *minimum distance* clause (which can then be combined with other clauses) in order to be well-formed and compile. Then, if no *output* option is specified, the default output option CAT is used.

In case the user chooses to specify only *on_attribute* (without a distance clause), this must be followed by the specification of the *output* option, considering that the values allowed are: *LEFT*, *RIGHT*, *LEFT_DISTINCT*, *RIGHT_DISTINCT*, or *BOTH* (whilst *INT* and *CAT* are not permitted).

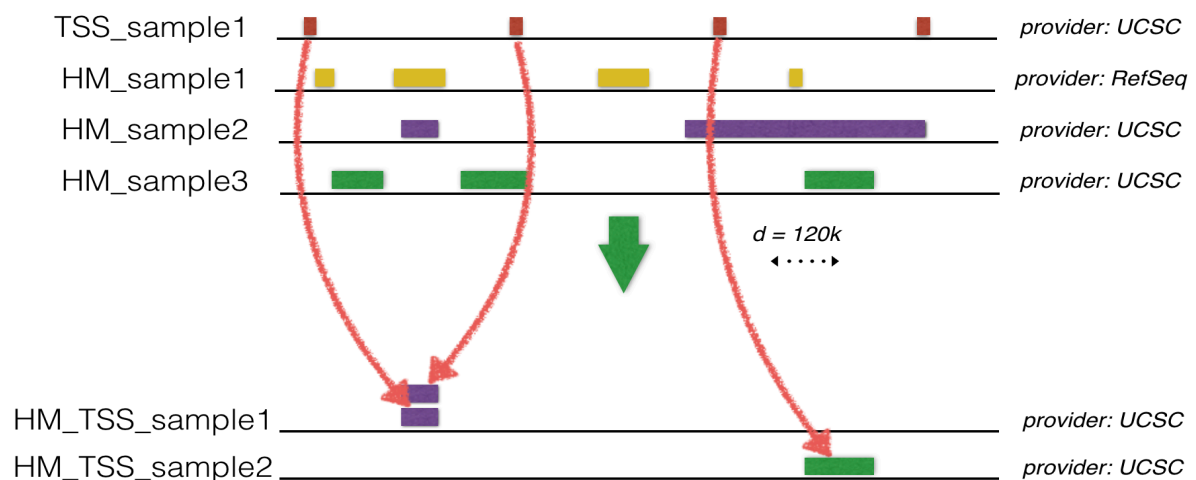
Note that, in case both distance and *on_attribute* conditions are specified, these are considered in conjunction: since the distance condition has to hold, the matching region pairs have to be on the same chr, and therefore in this case it is possible to use the *INT* and *CAT* output options.

Example 1:

HM_TSS = JOIN(MD(1), DGE(120000); output: RIGHT; joinby: provider) TSS HM;

Given a dataset HM of ChIP-seq experiment samples regarding Histone Modifications and a dataset called TSS with a sample including Transcription Start Site annotations, this GMQL statement searches for those regions of HM that are at a minimal distance from a transcription start site (TSS) and takes the first/closest one for each TSS, provided that such distance is greater than or equal to 120K bases and the joined TSS and HM samples are obtained from the same *provider* (*joinby* clause).

Assume that sample metadata are as shown in the following picture.



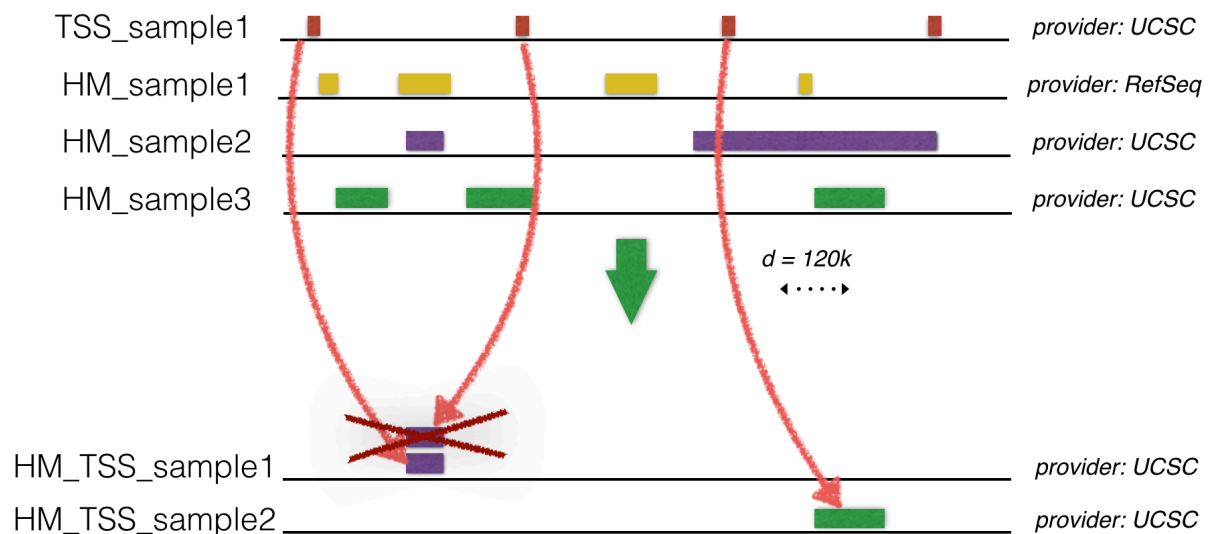
The first sample (HM_sample1) is excluded from JOIN genomic predicate scan because of mismatching metadata: the cardinality of the result dataset is $2 \times 1 = 2$ samples. The result includes only the selected regions of the right input dataset (in this case HM), with their attributes and values together with the attributes and values of the joined region in the other input dataset (in this case the left one TSS).

Region from HM_sample2 (purple) is replicated in HM_TSS_sample1 because it is a valid JOIN results for both the first two TSS (red) regions, respectively; from HM_sample3 only one green region is selected since MD(1) condition is evaluated first.

Example 2:

HM_TSS = JOIN(MD(1), DGE(120000); output: RIGHT_DISTINCT; joinby: provider) TSS HM;

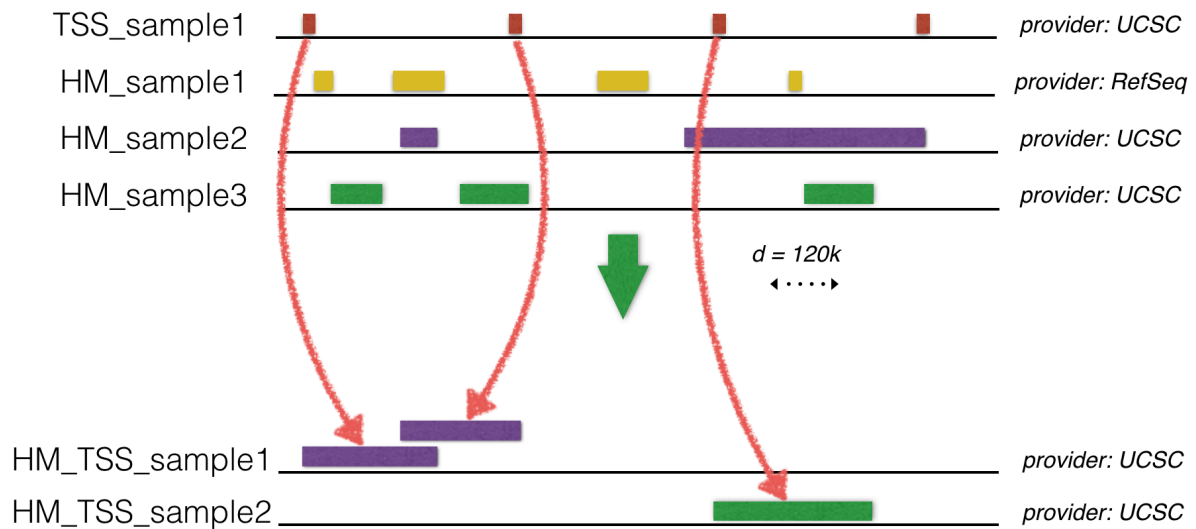
This example replicates Example 1, but uses a different output option: RIGHT_DISTINCT instead of RIGHT. RIGHT_DISTINCT allows to eliminate replicated regions in the output due to joining with multiple regions in the other dataset sample. In this specific case, the left most region from HM_sample2 (purple) is a valid JOIN result for both the two leftmost regions in the TSS_sample1 (red), but in the output dataset TSS_HM only one of these two HM_sample2 regions is included (given the RIGHT_DISTINCT output option used). The output metadata are equal to the metadata of the input *experiment* (HM) dataset, without prefixing their attribute names.



Example 3:

HM_TSS = JOIN(MD(1), DGE(120000); output: CAT; joinby: provider) TSS HM;

This example includes the same input datasets, genomic predicate and joinby condition as in Example 1, but the output is produced as the concatenation of regions selected by the genomic predicate (see CAT description above). In the following picture we report the output JOIN samples in this scenario:



Example 4:

TFBS_TSS = JOIN(DGE(5000), DLE(100000); output: LEFT) TFBS TSS;

Given a dataset TFBS that contains peak regions of transcription factor binding sites (TFBSs), and another dataset named TSS that contains 1 bp-long transcription start sites (TSSs), this GMQL statement returns as output all those TFBSs that are far no more than 100k bp, but no less than 5000 bp from a TSS (i.e., in possible enhancer regions).

If one would instead be interested in the TSSs that have at least one TFBS in such regions, this statement could be changed by using *output: RIGHT* instead of *output: LEFT* as parameter (see Example 1).

Example 5:

TFBS_TSS = JOIN(DL(30000); output: LEFT_DISTINCT) TFBS TSS;

For each pair of samples, one from the TFBS dataset and the other from the TSS dataset, this statement selects all the regions in the TFBS dataset sample such that their distance from a region in the TSS dataset sample is less than 30000 bases. The output samples do not contain any replicate region, in case generated by the join operation, which are eliminated due to the LEFT_DISTINCT output option (differently from the LEFT output option). The output metadata are equal to the metadata of the input TFBS dataset, without prefixing their attribute names.

Example 6:

TFBS_TSS = JOIN(DIST < 100; output: BOTH) TFBS TSS;

For each pair of samples, one from the TFBS dataset and the other from the TSS dataset, this statement selects all the regions of the TFBS dataset sample such that their distance from a region in the TSS dataset sample is less than 100 bases. Output regions include all attributes and values of selected TFBS dataset regions, as well as attributes and values and (differently from other output options) coordinates of the paired TSS dataset region.

Note that Example 4, 5 and 6 show different uses of the output option. In Example 4, using LEFT, the metadata are a union of the metadata attributes from the input TFBS and TSS datasets, prefixed with their dataset name. The schema of the region attributes also corresponds to the union of the attributes used in the two datasets. In Example 5, which uses LEFT_DISTINCT, metadata attributes in the samples of the output dataset are only those belonging to the TFBS (left) dataset, without any prefix. As far as the region attributes schema is concerned, it is the same as the one of the TFBS (left) dataset. In Example 6, using BOTH, the metadata are treated as in the case of the LEFT (or RIGHT) option. As to the region part, the behavior of the output option BOTH is the same of the output option LEFT with the difference that the coordinates of the region from the right dataset are included in the output as additional attributes of the region selected from the left dataset.

Example 7:

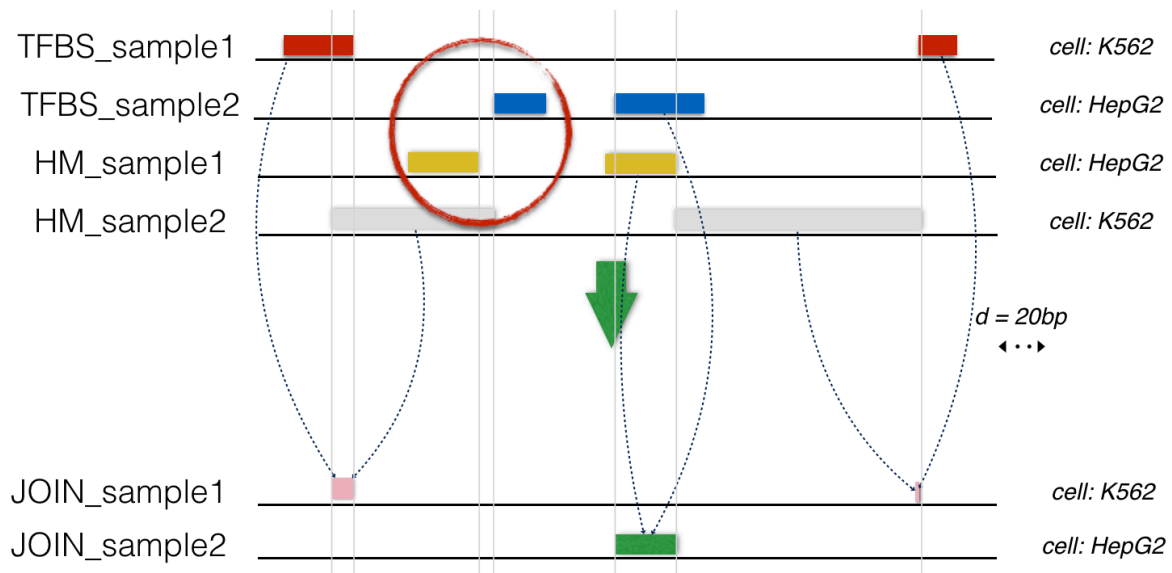
TF_HM_OVERLAP = JOIN(DLE(20); output: INT; joinby: cell) TFBS HM;

The input consists of a dataset TFBS that contains peak regions of transcription factor binding sites (TFBSs) for certain TFs, and another dataset named HM that contains regions resulting from experiments targeting specific histone modifications (for instance methylations).

For each pair of samples, one from the TFBS dataset and the other from the HM dataset, provided that they regard the same cell line (indicated by the *joinby* condition which checks the correspondent metadata attribute *cell*), this statement selects all the intersections (*output* option INT) between regions in a TFBS dataset sample such that their distance from a region in the HM dataset sample is less than or equal to 20 bases.

The metadata of an output sample are the union of the metadata of the two intersected samples which it was originated from.

For suitable histone modifications, this GMQL statement can provide evidence of actual TF bindings to the DNA in open chromatin regions.



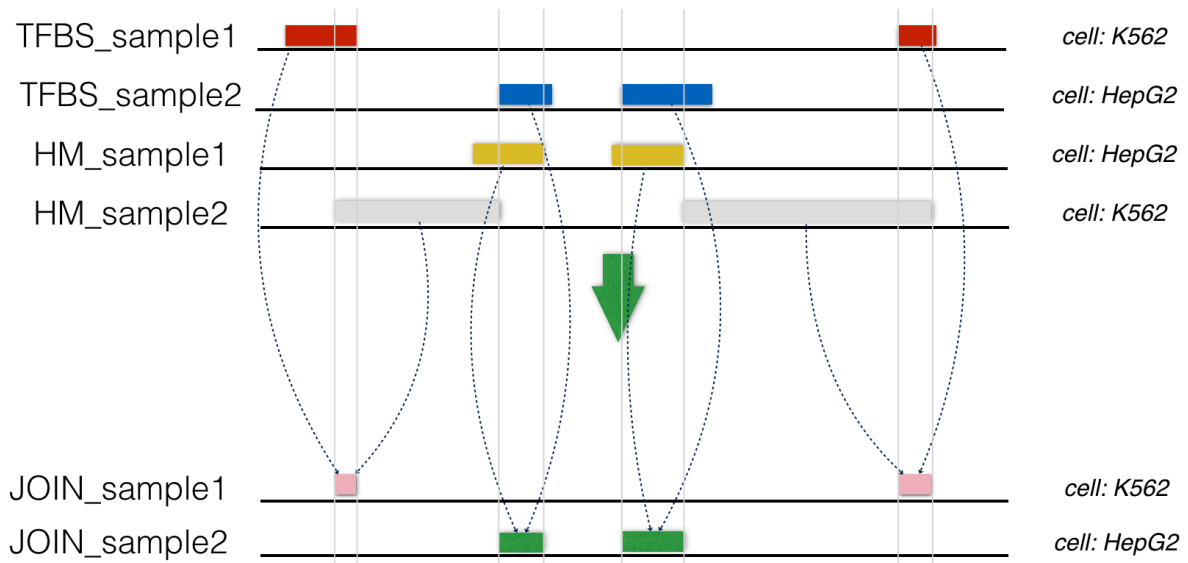
Note in the picture that the first blue region in TFBS_sample2 matches the first yellow region in HM_sample1 since their distance is lower than 20 bp, but no correlated region is output in the JOIN_sample2 since their intersection is empty.

Example 8:

TF_HM_OVERLAP = JOIN(DLE(0); output: INT; joinby: cell) TFBS HM;

Differently from the previous example, this JOIN statement first selects all regions in a TFBS dataset sample such that they are adjacent to or overlap a region in the HM dataset sample (always satisfying the *joinby* condition), then selects for output only the intersections between the mentioned regions.

Note that the effect of the statement in Example 7 and in this example in the specific case depicted below is the same.



Example 9:

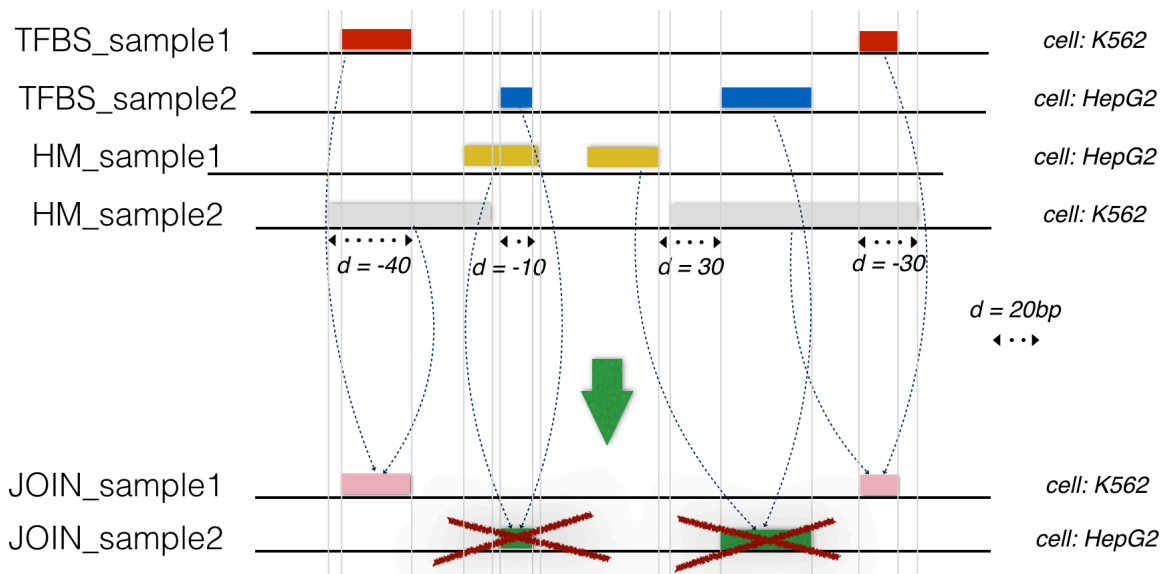
TF_HM_OVERLAP = JOIN(DL(-20); output: LEFT; joinby: cell) TFBS HM;

Differently from Examples 7 and 8, this JOIN statement considers all regions in a TFBS dataset sample such that their distance with a region in the HM dataset sample is lower than -20 bp, i.e., TFBS regions overlapping a HM region and with a distance lower than (at least of) -20 bp from it. This is only computed for samples satisfying the *joinby* condition.

In other words, the output includes only the anchor regions from a TFBS sample such that they have a non-empty intersection with a region of a HM sample and the absolute value of their distance is greater than 20.

As it can be observed in the figure below: the first green region in the JOIN_sample2 is not output because the distance between the first blue region in TFBS_sample2 and the first yellow region in HM_sample1 measures -10, which is not lower than -20; furthermore, it should be observed that the distance between the second blue region in TFBS_sample2 and the second yellow region in HM_sample1 measures 30 bp, which is greater than -20, so the two regions are not matched.

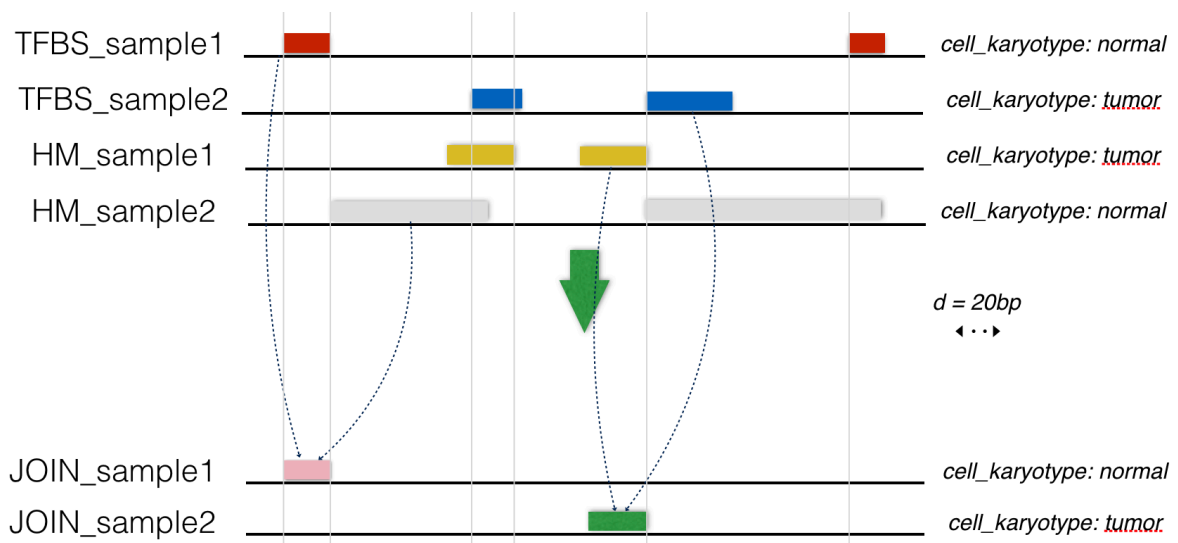
Note that, in the figure, the line for JOIN_sample2 is represented only for illustration purposes of this specific example. Indeed, since it does not contain any region, the JOIN_sample2 is not generated in the output dataset.



Example 10:

TF_HM_ADJACENT = JOIN(DGE(0), DLE(0); output: LEFT; joinby: cell_karyotype) TFBS HM;

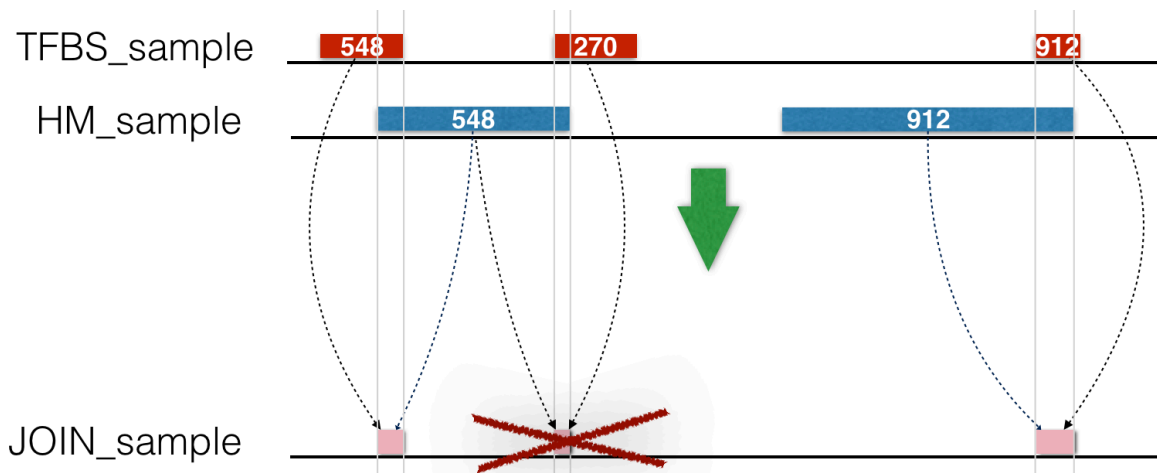
This GMQL statement returns as output only the TFBS dataset sample regions that are adjacent to a HM dataset sample region. As it can be observed in the figure below, TFBS samples are only matched against HM samples with the same value for the metadata attribute *cell_karyotype*. In the output we observe the first pink region as the result of the match between the first red region in the TFBS_sample1 (normal karyotype) and the first grey region in the HM_sample2, which are adjacent (the right coordinate of the first corresponds to the left coordinate of the second). The same applies in the case of the second blue region of the TFBS_sample2 and the second yellow region of the HM_sample1 (both tumor karyotype). All the other regions do not contribute to any output region because they are not adjacent but either overlapping or not intersecting in general.



Example 11:

TF_HM_OVERLAP = JOIN(DL(0); on_attributes: score; output: INT) TFBS HM;

This statement shows the use of the equi predicate option, with syntax *on_attributes*. In case the input datasets TFBS and HM do not present the attribute *score* in their schemata, the output dataset is empty. Assuming that *score* is present as region attribute in both datasets, the statement only matches those regions in the anchor dataset with the regions in the experiment dataset that have the same value in this region attribute. In the following picture we observe that the first red region is matched with the first blue region since they overlap (therefore DL(0) is satisfied) and they have the same score. Same applies to the last red region with the last blue region. Instead, the second red region does not match the first blue region because they have a different score.



Example 12:

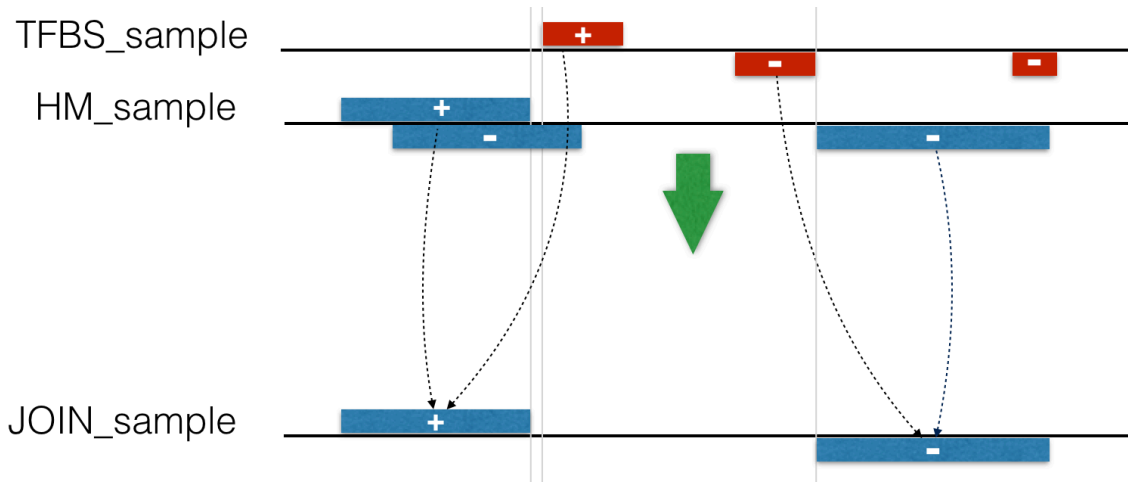
TF_HM_OVERLAP = JOIN(MD(1), UP; output: RIGHT) TFBS HM;

This statement shows the use of the upstream clause, with syntax *UP* (or *UPSTREAM*), in the genomic predicate. This clause requires that the remaining part of the genomic predicate (in this case the minimum distance clause MD(1)) holds only on the upstream genome with respect to the regions of the anchor dataset TFBS, taking into account their strand.

In the positive strand, UP is true for those regions of the HM sample whose right end is lower than, or equal to, the left-end of the TFBS sample regions. In the negative strand, UP is true for those regions of the HM sample whose left end is higher than, or equal to, the right end of the TFBS sample regions.

The first blue region in the output sample is present since it is produced by matching the first region with positive strand in the anchor and the first region with positive strand in the experiment. The second blue region in the output is present since it is produced by matching the first region with negative strand in the TFBS sample and the second region with negative strand in the HM sample. Note that in this case the right end of the TFBS region is equal to the left end of the HM region. Note also that the third red region in the TFBS sample, with negative strand, does not match any HM region.

In the output, accordingly to the specified *output* option, only regions from the experiment dataset are given.



Example 13:

TF_HM_OVERLAP = JOIN(MD(1), DOWNSTREAM; output: RIGHT) TFBS HM;

This statement shows the use of the downstream clause, with syntax *DOWNSTREAM* (or *DOWN*), in the genomic predicate. This clause requires that the remaining part of the genomic predicate (in this case the minimum distance clause MD(1)) holds only on the downstream genome with respect to the regions of the anchor dataset TFBS, taking into account their strand.

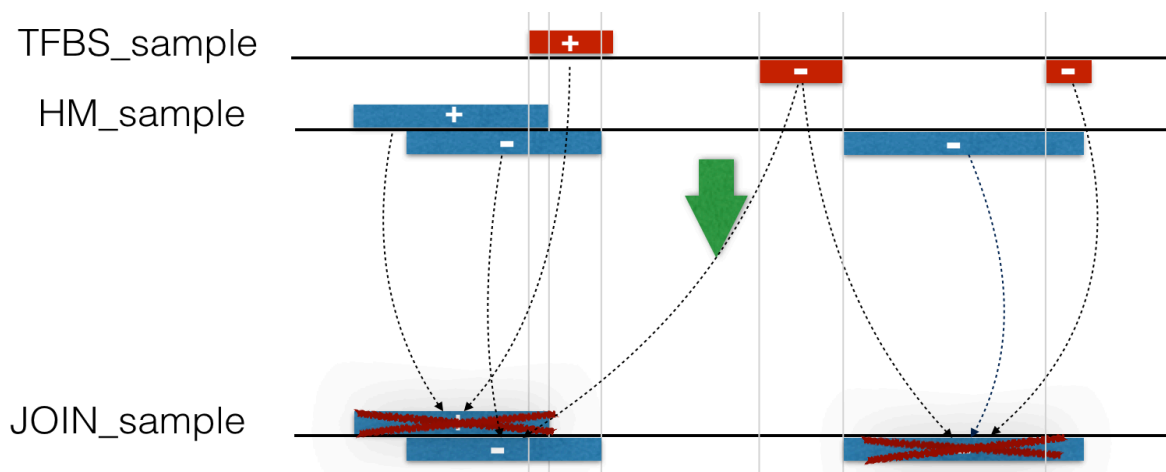
In the positive strand, DOWN is true for those regions of the HM sample whose left end is higher than, or equal to, the right end of the TFBS sample regions. In the negative strand, DOWN is true for those regions of the HM sample whose right end is lower than, or equal to, the left end of the TFBS sample regions.

In the figure below the first blue region in the output is not created because the two regions from which it could be generated are overlapping.

The second blue region in the output is created because it matches the downstream clause: its right end is lower than the left end of its closest anchor region with a negative strand.

The third blue region is not created because the blue HM region does not match the downstream clause with neither the second red TFBS region, nor the third red TFBS region.

In the output, accordingly to the specified *output* option, only regions from the experiment dataset HM are given.



12) COVER

COVER is a GMQL operator that takes as input a dataset (of usually, but not necessarily, multiple samples) and returns another dataset (with a single sample, if no *groupby* option is specified) by “collapsing” the input samples and their regions according to certain rules specified by the COVER parameters. The attributes of the output regions are only the region coordinates, plus in case, when aggregate functions are specified, new attributes with aggregate values over attribute values of the contributing input regions; in addition, the two attributes *JaccardIntersect* and *JaccardResult* are always added, computed as described below. Output metadata are the union of the input ones.

The COVER operation is used to:

- reduce the regions of multiple samples in a single sample (particularly when the samples are replicas of the same experiment);
- deal with overlapping regions;
- compute aggregates on the overlapping regions.

The general syntax of the COVER operator is:

```
DSout = COVER(minAcc, maxAcc;  
              groupby: M1, ..., Mn;  
              aggregate: NR1 AS g1, ..., NRh AS gh) DSin;
```

where:

- DS_{in} is the input dataset;
- minAcc (maxAcc) is the minimum (maximum) accumulation value, i.e., the minimum (maximum) number of overlapping regions to be considered during COVER execution;
- DS_{out} is the output dataset;
- M₁, ..., M_n are the (optional) metadata attributes used in the *groupby* clause (see below);
- NR₁, ..., NR_h are new genomic region attributes (optionally) generated using aggregate functions g₁, ..., g_h on existing region attributes.

The special keywords *ANY* and *ALL* can be used instead of numbers for *minAcc* and *maxAcc*. In particular:

- *ALL* sets the minimum (and/or maximum) to the number of samples in the input dataset;
- *ANY* acts as a wildcard and can be used only as *maxAcc* value; in this case, the COVER extracts all regions with any maximum accumulation value. For instance, COVER(2, ANY) considers all areas defined by a minimum of two overlapping regions in the input dataset, up to any amount of overlapping (3, 4, 5, and so on).

Note 1: COVER and its three variants (FLAT, SUMMIT, and HISTOGRAM), which are described in the following, do not have default arguments (i.e., COVER() DS_{in} does not compile); *minAcc* and *maxAcc* must always be specified.

Note 2: Given any two integer numbers k and n, *minAcc* and *maxAcc* can be optionally expressed as functions of ALL, with the following possible structures:

- ALL / n;
- (ALL + k) / n.

The division is to be considered as an integer division (e.g., $5 / 2 = 2$).

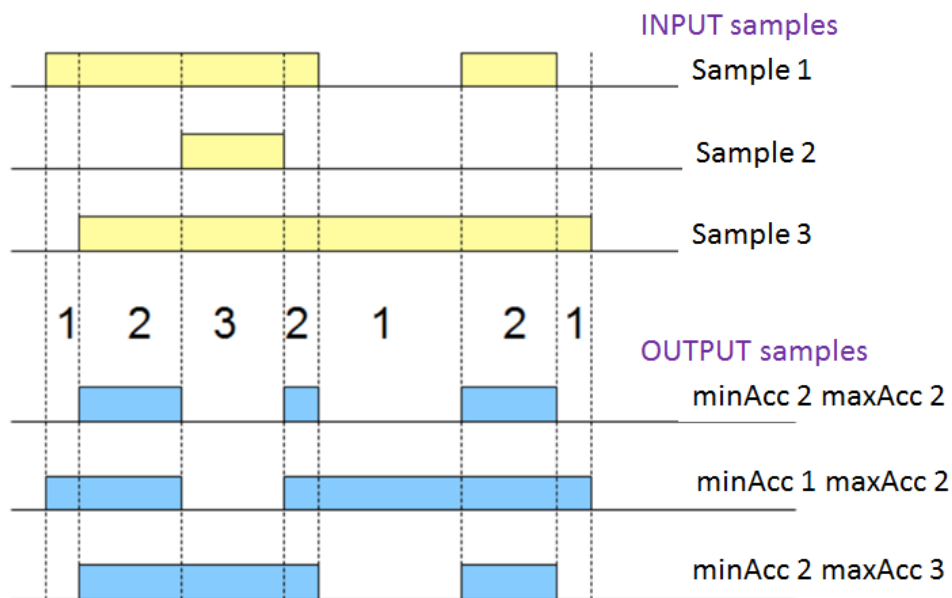
Note 3: As mentioned in the *Foreword* section, in *groupby* option (which is one of the possible *metajoin* options of GMQL) different alternatives are available with respect to dot-separated prefixes in case present for metadata attribute names:

- `metadata_attribute_name`;
- `EXACT(metadata_attribute_name)`;
- `FULL(metadata_attribute_name)`.

Please refer to the [Foreword](#) section of this document for further details.

We first describe the COVER operation with no grouping. In such a case, the operation produces a single output sample, and all the metadata attribute-values of the contributing input samples in DS_{in} are assigned to the resulting single sample in DS_{out} . Regions of the resulting sample are built from DS_{in} in the following way:

- Each resulting region r in DS_{out} is the contiguous intersection of at least $minAcc$ and at most $maxAcc$ contributing regions r_i in the samples of DS_{in} ;
- When regions are stranded, COVER is separately applied to positive and negative strands, unless also unstranded regions are present. In this case, all regions are considered unstranded only;
- Resulting regions may have new attributes NR_i , calculated by means of aggregate expressions over the attributes of the contributing regions; for instance, *Jaccard Indexes* are standard measures of similarity of the contributing regions r_i , added as default region attributes. The *JaccardIntersect* index is calculated as the ratio between the lengths of the intersection and of the union of the contributing regions; the *JaccardResult* index is calculated as the ratio between the lengths of the result and of the union of the contributing regions.



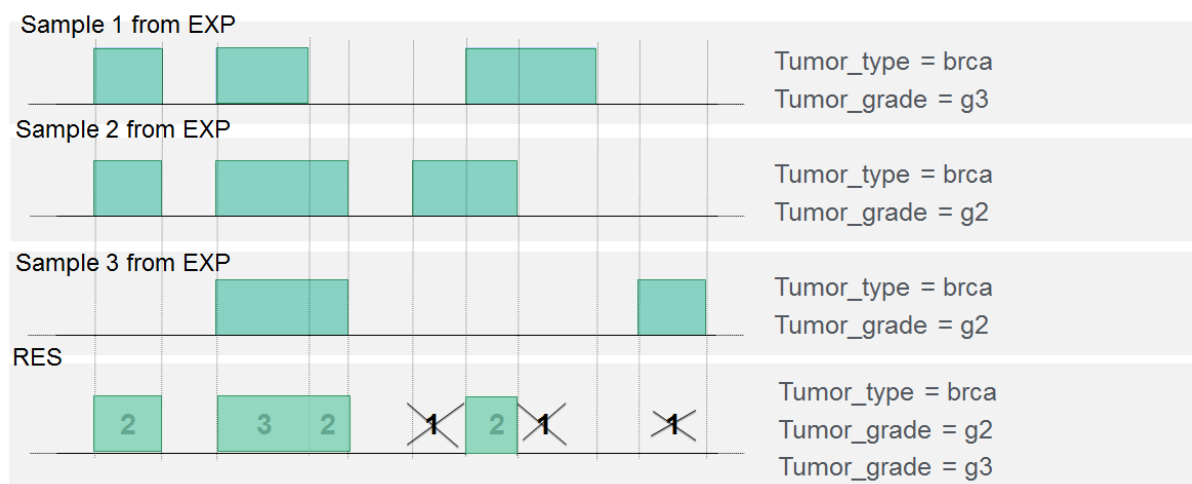
In the above figure we show the results of COVER with *minAcc* and *maxAcc* parameter values set respectively to (2, 2), (1, 2), and (2, 3). Note that in the figure cases $ALL = 3$; so, for instance, COVER(2, 3) provides the same result as COVER(2, ALL).

When a *groupby* clause is specified, the input samples are partitioned in groups, each with distinct values of the grouping metadata attributes, and the COVER operation is separately applied (as described above) to each group, yielding to one sample in the result for each group (input samples that do not satisfy the *groupby* condition are disregarded).

Example 1:

RES = COVER(2, ANY) EXP;

This GMQL statement produces an output dataset with a single output sample. The COVER operation considers all areas defined by a minimum of two overlapping regions in the input samples, up to any amount of overlapping regions. In the figure below we show how no regions are created in the output where only one or no region in the input samples is present. Output region attributes include only region coordinates and Jaccard indexes (*JaccardIntersect* and *JaccardResult*). Metadata are the union of the input metadata, as shown in figure.



Example 2:

RES = COVER(2, 3; groupby: cell; aggregate: min_pvalue AS MIN(pvalue)) EXP;

This GMQL statement computes the result grouping the input EXP samples by the values of their *cell* metadata attribute, thus one output RES sample is generated for each cell type; output regions are produced where at least 2 and at most 3 regions of grouped EXP samples overlap, setting as attributes of the resulting regions the minimum pvalue of the overlapping regions (*min_pvalue*) and their Jaccard indexes (*JaccardIntersect* and *JaccardResult*).

Example 3:

CELL_TF = COVER(1, ANY; groupby: cell, antibody_target) NARROW_PEAK;

Given a dataset NARROW_PEAK, containing transcription factor binding site (TFBS) regions from a repository (e.g., ENCODE), for each antibody target of each cell line, this GMQL statement produces output regions where at least a binding site of the given transcription factor for the given cell exists, grouping cells (first) and antibody targets (then); output regions have only their Jaccard indexes as their attributes. This statement is typically used to extract any possible regions where a TFBS for a given cell line can exist; by rising the *minAcc* parameter (e.g., to 2, 3, or more), the same statement can be used to extract consensus

regions (i.e. regions with higher probability of containing actual signal, in the example case a TFBS for a given cell line).

Cover variants

Three variants are available in GMQL for the COVER operation, which vary the coordinates of the returned regions as follow:

- FLAT returns the union of all the regions which contribute to the COVER. More precisely, it returns the contiguous regions that start from the first end and stop at the last end of the regions which contribute to each region of a COVER with the same parameters;
- SUMMIT returns only those portions of the COVER result with the maximum number of overlapping regions (this is done by returning only regions that start from a position after which the number of overlaps does not increase, and stop at a position where either the number of overlapping regions decreases or violates the maximum accumulation index);
- HISTOGRAM returns all regions contributing to the COVER divided in different (contiguous) parts according to their accumulation index value (one part for each different accumulation value), which is assigned to the additional *AccIndex* region attribute.

The syntax for all variants is the same as for the COVER statement, only replacing COVER with FLAT, HISTOGRAM, or SUMMIT, respectively, as required. Output metadata, as well as region attributes and values, are as for the COVER statement, except for HISTOGRAM which always provides also the *AccIndex* region attribute.

Example 1:

```
RES = FLAT(2, 4; groupby: cell) EXP;
```

This GMQL statement computes the result grouping the input EXP samples by the values of their *cell* metadata attribute, thus one output RES sample is generated for each cell type. Output regions are produced by concatenating all regions which would have been used to construct a COVER(2, 4) statement on the same dataset; Jaccard indexes (*JaccardIntersect* and *JaccardResult*) are set as in the COVER case.

Example 2:

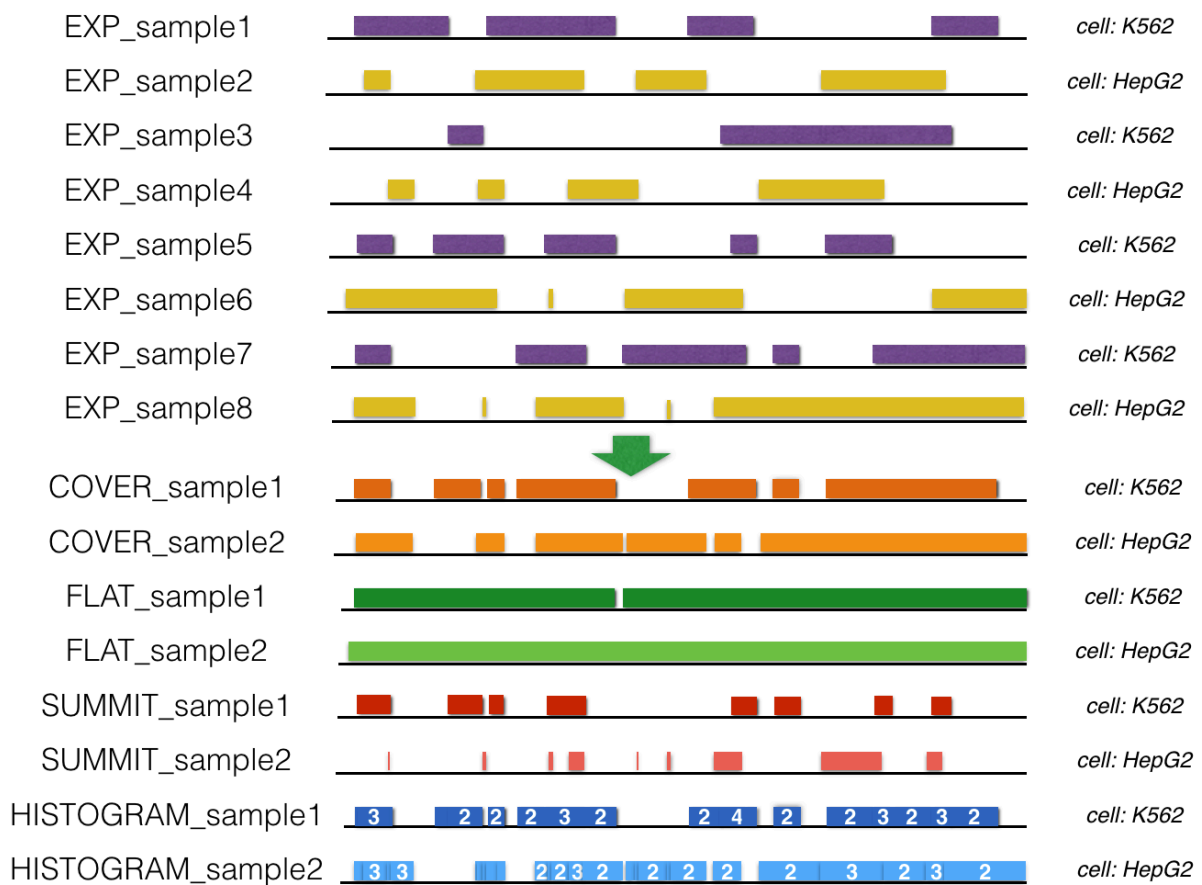
```
RES = SUMMIT(2, 4; groupby: cell) EXP;
```

This GMQL statement computes the result grouping the input EXP samples by the values of their *cell* metadata attribute, thus one output RES sample is generated for each cell type. Output regions are produced by extracting the highest accumulation of overlapping (sub)regions, according to the methodologies described above; Jaccard indexes (*JaccardIntersect* and *JaccardResult*) are set as in the COVER case.

Example 3:

RES = HISTOGRAM(2, 4; groupby: cell) EXP;

This GMQL statement computes the result grouping the input EXP samples by the values of their *cell* metadata attribute, thus one output RES sample is generated for each cell type. Output regions are produced by dividing results from COVER in contiguous sub-regions according to the varying accumulation values (from 2 to 4 in this case): one region for each accumulation value (see figure below for a visual explanation; please note that in the figure HISTOGRAM_sample2 the accumulation values of some regions do not appear due to the complexity of the example and the limited figure resolution). Jaccard indexes (*JaccardIntersect* and *JaccardResult*) are set as in the COVER case.



Example 4:

RES = HISTOGRAM(ALL/2, (ALL+1)/2; groupby: antibody_target) EXP;

This statement computes the result grouping the input EXP samples by the values of their *antibody_target* metadata attribute, thus one output sample is generated for each type of antibody target.

Assuming that the cardinality of the EXP dataset is of 8 samples, then $ALL = 8$. By computing the simple arithmetic operations, we obtain $minAcc = 4$ and $maxAcc = \text{floor}(4.5)$. Therefore, the output regions are produced exactly as if the user had performed an HISTOGRAM(4, 4), after applying the *groupby* option, i.e., HISTOGRAM(4, 4; groupby: antibody_target).

C. FEDERATED EXTENSIONS

A federated GMQL query is executed at two or more communicating servers or clouds (from now on called GMQL INSTANCE). Such execution is launched at a given GMQL instance by a user and terminates at that site with a MATERIALIZE statement that moves the result in the user's private space. Queries are extended by means of directives and operation allocations.

- **Operator allocation:**

A new named optional parameter [**at: instance_name**] is available for all the operators; syntactically this clause is the last one in the operation specification. It indicates that the operation has to be performed on the specified GMQL instance. The clause is optional when a policy is specified, and mandatory otherwise; if a policy is specified, the operation allocation overrides the allocation that would be given by the policy. The SELECT operation that applies directly to a GDM dataset must be allocated at the GMQL instance where the dataset is stored – as this operation is also responsible of loading the dataset from the file system.

- **Directives:**

Directives are used in order to specify properties that hold for the whole query. For this reason, they should be written at the beginning of the query. Directives are introduced by the special character "@". The current system provides two directives:

- 1) **@protected <dataset_name>**

When present, it implies that the user wants to protect the specified dataset from being moved to other GMQL instances. The execution plan of the query must be such that the specified dataset and any of the results obtained from its manipulation are not moved from its original GMQL instance; when instead the execution plan violates such property, execution is terminated with a documented error. Several datasets (on the same GMQL instance) may be protected in the same query; in this case, a directive for each dataset has to be written.

- 2) **@policy <policy_name> [<parameter>]**

When present, it indicates that the allocation of the sub-queries to the GMQL instances is automatically decided by the system, according to the specified policy. Three policies can be specified:

(a) **@policy distributed:** each dataset is processed on the instance of origin until a move is required; this happens when a binary operation (e.g., a JOIN) on two datasets that are in two distinct instances is performed. By default, the right dataset is moved to the location of the left dataset. The choice may change in presence of protected directive so as to satisfy the protection constraint.

(b) **@policy centralized instance_name:** all the operations of the query except the initial select are executed on the specified GMQL instance.

(c) **@policy externalized cluster_name:** all the operations of the query except the initial select are executed on the specified cluster.