Supplementary material

S1 GMQL comparison with STQL

Here, we compare our GenoMetric Query Language (GMQL) with the closest related work published, i.e., the Signal Track Query Language (STQL) (Zhu *et al*., 2017). STQL is a declarative, SQL-like query language, supported by a Web-based tool called Signal Track Analytical Research Tool (START) (http://yiplab.cse.cuhk.edu.hk/start/), which has many common aspects with GMQL, as initially proposed in (Masseroli *et al*., 2015). STQL tracks correspond to GMQL samples, as they share the same organization based on region coordinates. STQL and GMQL have a different style of query writing, as STQL uses the classic Select-From-Where nesting of SQL rather than many distinct algebraic operations as in GMQL.

A main difference is that GMQL queries output an arbitrary number of samples while STQL queries produce a single track as result; in input, GMQL implicitly iterates over many samples of a same dataset, while STQL requires an explicit iterator. Another difference is that STQL is implemented directly on Apache Hive (https://hive.apache.org/) warehousing (optimized for supporting some STQL primitives) and hence depends on a specific cloud engine, with higher upkeep cost, while our approach maps DAG operations to engines, with an engine-independent execution workflow (we implemented GMQL on Spark, Flink and SciDB). Furthermore, in GMQL region data are kept in a GMQL associated repository in their original format, keeping them distinct from metadata, and are not preloaded to a database; this makes them usable for other applications, with no data replication. Conversely, in STQL they are preprocessed and loaded to a database.

The most distinctive feature of GMQL relative to STQL and to all other genomic data management systems proposed up to now is that GMQL processes sample associated metadata during query evaluation and allows tracing the contribution of each input sample to the query result. This aspect is most significant with outputs consisting of multiple samples; besides facilitating biomedical interpretation of results, it additionally allows execution optimizations, for example such as the loading of only the data samples that contribute to the result, based on all the sample metadata predicates computed by the GMQL query. In several cases this significantly reduces the amount of loaded (and then processed) samples, highly decreasing the execution time.

STQL has comparatively more expressive power than GMQL for what concerns supported region calculus. Some relations between regions in the WHERE clause of STQL, specified in Table 1 of (Zhu *et al*., 2017), are hardly translated into GMQL, including: "*contains*, *is within*, *is prefix of*, *is suffix of*, *precedes*, *follows*"; however, their use is harder in many-to-many sample comparisons than in single tracks comparison. Some advanced constructs of STQL currently are not directly supported in GMQL; among them the STQL *exclusivejoin*, that can be expressed as a concatenation of GMQL operations (see for example query SQ4 below) which produce equal results except in the not usual case when one or both tracks/samples in the pairwise comparison contain overlapping regions.

In other cases, GMQL is strictly more powerful than STQL; for instance, in extracting the first k genomic regions (with k specified by the user) at minimum distance from another region (denoted as anchor), which is not supported in STQL. GMQL can also implicitly take the region strand into account with no need of strand-specific encoding, which is conversely required in STQL (see query CQ5 below as an example). Some GMQL operations which take advantage of sample metadata predicates have no counterpart in STQL, in particular those including *groupby* and *joinby* options (see Section S1.2) which allow GMQL to perform implicit iterations even on specific sample pairs matched through their metadata.

In the Supplementary material of (Zhu *et al*., 2017), to demonstrate the use of STQL, authors provided 14 sample queries showing 8 simple and 6 composite STQL queries, and compared their execution with that of equivalent BEDTools (Quinlan and Hall, 2010), Galaxy (Goecks *et al*., 2010) and Python programming language (https://www.python.org/) scripts (when they could implement the latter ones). For better learning the specific aspects of each sample query and to test on real samples (the same used in (Zhu *et al.*, 2017)) the results of executing equivalent queries in GMQL and STQL, we rerun these queries in START (the STQL Web-based tool), and run their GMQL encoded version in the GMQL system Web-interface; they are both open for public access.

Since the hardware is different, it is not meaningful to use the measured time to argue which approach is more efficient; however, an internal comparison of the execution times of simple queries SQ1-SQ8, shown in Table S1, highlights specific simple queries where each system is most effective. For instance, in query SQ4 STQL uses *exclusivejoin*, and the corresponding heavy rewriting of GMQL yields to a query formulation that, comparatively within GMQL queries, is less performing. Queries SQ7 and SQ8, whose expressive power can be reduced to just two or three simple GMQL operations (SELECT on regions and PROJECT or EXTEND, respectively), are comparatively lighter than other queries in GMQL, taking also into account the differences in the input data used in the different queries.

Table S1. Execution time of the eight example simple queries in seconds

Query	STQL (START)	GMQL (Web-interface)
SQ1	345	296
SQ2	160	55
SQ3	40	25
SQ4	90	119
SQ5	62	67
SQ6	40	24
SQ7	63	10
SQ8	29	10

The main purpose of the comparison is to show that, in spite of differences in expressive power of the region calculus, all test queries of STQL can be expressed in GMQL producing equivalent output results; the test implicitly provides an analysis of expressive power also w.r.t. BEDTools, Python and Galaxy, as for several of the 14 sample queries in (Zhu *et al.*, 2017) the authors were unable to find a trivial way to express the same operations with the formalisms used in such other popular approaches.

STQL authors provided also a user study where one biologist easily learnt to program STQL and compared it with PERL programming language, finding the former easy to learn. For a comparison of STQL and GMQL in terms of usability, one can compare the STQL compact and nested form (e.g., of query SQ2 and SQ5) with the progressive style of building results of its translation to GMQL. We agree with (Olston *et al.*, 2008) that “the latter method is much more appealing to programmers than encoding their task as an SQL query.”

In the following, we provide the translation of the 8 simple (SQ1-SQ8) and 6 composite (CQ1-CQ6) STQL sample queries to GMQL. For a full account of the biological use of each query, we refer the reader to the Supplementary material of (Zhu *et al.*, 2017); we provide instead a summary and a description of the mapping from STQL to GMQL. We also show one example of GMQL query that is not supported in STQL; other GMQL examples unsupported in STQL are available in the GMQL documentation at http://www.bioinformatics.deib.polimi.it/GMQLsystem/, particularly in the *GMQL example queries* and in the *Biological examples* documents. The example queries in the former document are also available in the GMQL Web interface (*Example Queries* top menu) at http://www.gmql.eu/, where they can be directly executed.

S1.1 Translation of sample STQL queries into GMQL

SQ1: It computes the average signal of H3K4me1 histone modification at each 100 bp bin across the whole genome, for identifying potential transcriptional enhancers.

STQL:

```
SELECT *
FROM (project
    `wgEncodeBroadHistone`.`wgEncodeBroadHistoneGm12878H3k04me1StdSigV2.bigWig`
on
    generate bins with length 100 with vd_sum using EACH MODEL) Ntint
WHERE Ntint.value > 0;
```

GMQL:

```
HIS = SELECT(tableName == "wgEncodeBroadHistoneGm12878H3k04me1StdSigV2")
    HG19_ENCODE_SIGNAL
# BINS dataset includes a single sample with consecutive regions of fixed size (100 bp)
BIN = SELECT() BINS;
BIN_HIS_MAP = MAP(avg_signal AS AVG(score)) BIN HIS;
RES = SELECT(region: avg_signal > 0) BIN_HIS;
MATERIALIZE RES INTO SQ1;
```

Explanations: In GMQL, genome binning as part of the query language is not supported. In this and other queries, we use an artificially generated bin annotation sample consisting of consecutive regions (i.e., bins) of equal size (100 bases); then, we MAP the signal over such annotation, and SELECT the regions where the average signal is positive, obtaining the same result.

SQ2: It computes the expression level of each gene, defined as the average RNA (cDNA) sequencing (RNA-seq) signals covering the genomic locations of the gene.

STQL:

```
SELECT *
FROM (project
`wgEncodeCshlLongRnaSeq`.`wgEncodeCshlLongRnaSeqGm12878CellTotalPlusRawSigRep1.bigWig`
on (
    SELECT DISTINCT chr, chrstart, chrend
    FROM `wgEncodeGencode`.`gencode.v19.annotation.gtf`
    WHERE feature = 'gene') Ntint1
        with vd_avg using EACH MODEL) Ntint2
WHERE Ntint2.value > 0;
```

GMQL:

```
CSHL = SELECT(tableName == "wgEncodeCshlLongRnaSeqGm12878CellTotalPlusRawSigRep1")
    HG19_ENCODE_SIGNAL;
GENE = SELECT(region: feature == "gene") HG19_GENCODE_ANNOTATION;
GENE_DISTICT = GROUP() GENE;
CSHL_MAP = MAP(avg_signal AS AVG(score)) GENE_DISTINCT CSHL;
RES = SELECT(region: avg_signal > 0) CSHL_MAP;
MATERIALIZE RES INTO SQ2;
```

Explanations: In GMQL, we select genes from GENCODE gene annotations; we obtain the same effect as the DISTINCT option of STQL by GROUPING genes on distinct coordinate values. Then, similarly to the SQ1 example, a MAP operation computes the average RNA sequencing signal value of each distinct gene, and a SELECT operation extracts the regions of the genes with positive expression signal.

SQ3: It finds the genomic regions covered by signal peaks of both H3K4me1 and H3K27ac histone modifications, which are potential active enhancers in a particular context (the HCT-116 human cell line in this case).

STQL:

```
SELECT *
FROM `wgEncodeSydhHistone`.`wgEncodeSydhHistoneHct116H3k04me1UcdPk.narrowPeak`
     intersectjoin
`wgEncodeSydhHistone`.`wgEncodeSydhHistoneHct116H3k27acUcdPk.narrowPeak`;
```

GMQL:

```
H3K4me1 = SELECT(cell == "HCT-116" AND antibody_target == "H3K4me1")
   HG19_ENCODE_NARROW;
H3K27ac = SELECT(cell == "HCT-116" AND antibody_target == "H3K27ac")
   HG19_ENCODE_NARROW;
RES = JOIN(DISTANCE < 0; output: INT) H3K4me1 H3K27ac;
MATERIALIZE RES INTO SQ3;
```

Explanations: GMQL expresses STQL *intersectjoin* with a JOIN operation with the metric clause DISTANCE < 0 and the region construction clause INT (which builds the intersection of matching regions, in this case the intersection region of signal peaks of both H3K4me1 and H3K27ac histone modifications).

SQ4: It identifies expressed regions outside annotated level-1 (experimentally validated) and level-2 (manually curated) GENCODE protein-coding genes, some of which could be non-coding RNAs.

STQL:

```
SELECT *
FROM `wgEncodeCshlLongRnaSeq`.`
wgEncodeCshlLongRnaSeqGm12878CellTotalPlusRawSigRep1.bigWig`
   exclusivejoin (
      SELECT chr, chrstart, chrend
      FROM `wgEncodeGencode`.`gencode.v19.annotation.gtf`
      WHERE feature = 'gene' AND attributes LIKE '%gene_type "protein_coding"%' AND
         (attributes LIKE '%level 1%' OR attributes LIKE '%level 2%')
) Ntint;
```

GMQL:

```
CSHL = SELECT(tableName == "wgEncodeCshlLongRnaSeqGm12878CellTotalPlusRawSigRep1")
   HG19_ENCODE_SIGNAL;
GENE = SELECT(region: feature == "gene" AND gene_type == "protein_coding" AND
   (level == 1 OR level == 2)) HG19_GENCODE_ANNOTATION;
CSHL1 = COVER(1,ANY) CSHL;
U1 = UNION() CSHL1 CSHL1;
GENE1 = COVER(1,ANY) GENE;
U = UNION() U1 GENE1;
```

```
    CSHLnotGENE = COVER(2,2) U;
    MATERIALIZE CSHLnotGENE  INTO SQ4;
```

Explanations: This query shows how GMQL can express the STQL *exclusivejoin* condition; it does so through a complex rewriting, which uses two UNION and three COVER operations. Let T1 be a track to be exclusively joined with a track T2, i.e., from which we want to extract all T1 region parts that do not overlap any T2 region. The intuition behind the rewriting is that, by considering three tracks such that two of them are the COVER(1,ANY) of the first track T1 and the third one is the COVER(1,ANY) of the second track T2, we can extract the exclusive join of T1 with T2 by making the UNION of the three tracks and then taking only the regions extracted by applying in them a COVER(2,2) operation (which extracts the genomic regions covered only by two of the three unified tracks, i.e., only by the T1 regions). We acknowledge that this is an artificial rewriting, which provides same results in the usual case when each track in the pairwise comparison contains no overlapping regions, and that the STQL exclusive join is much more elegant.

SQ5: It identifies contiguous genomic regions with significant expression, which could correspond to transcribed exons.

STQL:

```
    SELECT *
    FROM coalesce (
        SELECT chr, chrstart, chrend, value
        FROM
        `wgEncodeCshlLongRnaSeq`.`wgEncodeCshlLongRnaSeqGm12878CellTotalPlusRawSigRep1.bigWig`
        WHERE value > 2
    ) Ntint with vd_avg using EACH MODEL;
```

GMQL:

```
    CSHL = SELECT(tableName == "wgEncodeCshlLongRnaSeqGm12878CellTotalPlusRawSigRep1";
        region: score > 2) HG19_ENCODE_SIGNAL;
    RES = COVER(1,ANY; aggregate: avg AS AVG(score)) CSHL;
    MATERIALIZE RES INTO SQ5;
```

Explanations: This query highlights that a GMQL COVER(1,ANY) operation translates the STQL *coalesce* operator, and that GMQL can compute the aggregate function simply as part of the COVER operation – the aggregate function applies to the values of all the input regions which contribute to creating a region in the result.

SQ6: It identifies regions bound by a transcription factor that overlap binding sites of another factor, which could indicate co-binding events and provide information for finding functionally related factors.

STQL:

```
    SELECT *
    FROM `wgEncodeSydhTfbs`.`wgEncodeSydhTfbsHelas3CfosStdPk.narrowPeak` Tint1,
        `wgEncodeSydhTfbs`.`wgEncodeSydhTfbsHelas3CjunIggrabPk.narrowPeak` Tint2
    WHERE Tint1 overlaps with Tint2;
```

GMQL:

```
    CFOS = SELECT(cell == "HeLa-S3" AND antibody_tag == "CFOS" AND composite ==
        "wgEncodeSydhTfbs") HG19_ENCODE_NARROW;
    CJUN = SELECT(cell == "HeLa-S3" AND antibody_tag == "CJUN" AND composite ==
        "wgEncodeSydhTfbs") HG19_ENCODE_NARROW;
    RES = JOIN(DISTANCE < 0; output: LEFT_DISTINCT) CFOS CJUN;
    MATERIALIZE RES INTO SQ6;
```

Explanations: The STQL query extracts overlapping regions from two tracks which are prepared *a priori*. GMQL extracts the samples of interest from the relevant dataset and then computes the overlap condition by means of a JOIN operation with DISTANCE < 0. Note that the "*overlap with*" condition of STQL is not symmetric, as it extracts regions of the left operand which overlap with the right operand (hence the use of LEFT in GMQL), and that the _DISTINCT clause of GMQL is used to keep only one copy of the regions of the left operand (instead of the many pairs created for each matching region of the right operand).

SQ7: It identifies all annotated genes longer than a given length (e.g., 1000 bp).

STQL:
```
    SELECT *
    FROM `wgEncodeGencode`.`gencode.v19.annotation.gtf` Tint
    WHERE feature = 'gene' AND length(Tint) > 1000;
```

GMQL:
```
    GENE = SELECT(region: feature == "gene") HG19_GENCODE_ANNOTATION;
    GENE_LENGTH = PROJECT(region_update: length AS right - left) GENE;
    RES = SELECT(region: length > 1000) GENE_LENGTH;
    MATERIALIZE RES INTO SQ7;
```

Explanations: The two queries are very similar, they only differ because STQL provides a *length* function applicable to a region while GMQL explicitly computes the difference of the *right* and *left* coordinates of a region to derive the region length.

SQ8: It counts the number of annotated non-protein-coding genes, which is relatively more variable than the number of protein-coding genes among different annotation sets and different versions of the same annotation set.

STQL:
```
    SELECT COUNT(*)
    FROM `wgEncodeGencode`.`gencode.v19.annotation.gtf`
    WHERE feature = 'gene' AND attributes NOT LIKE '%gene_type "protein_coding"%';
```

GMQL:
```
    GENE = SELECT(region: feature == "gene" AND NOT(gene_type == "protein_coding"))
        HG19_GENCODE_ANNOTATION;
    RES = EXTEND(region_count AS COUNT()) GENE;
    MATERIALIZE RES INTO SQ8;
```

Explanations: GMQL does not provide text pattern matching (LIKE predicate); so, the STQL LIKE test is translated to standard equality in GMQL. The result of the query is a counter in STQL, while in GMQL it is a single metadata attribute named "*region_count*" that is added with its value to the single result sample. Note that this query applies on a single sample input dataset; thanks to GMQL implicit iterations, exactly the same GMQL query can be used with no changes also in the case the input dataset includes any number of samples, whereas the STQL query must be modified by explicitly express the required iterations.

<u>CQ1</u>: It counts the number of transcription factors with a binding peak overlapping each genomic location. Neighboring locations with the same count are grouped into one single interval in the results. This query can be used as one step in identifying regions with high occupancy of transcription-related factors (HOT) (Yip *et al.*, 2012).

STQL:

```
FOR TRACK T IN (category = 'SYDH TFBS', cell = 'GM12878' and fname LIKE '%Pk%')
SELECT chr, chrstart, chrend, value
FROM T
COMBINED WITH UNION ALL AS Step1Results;

SELECT *
FROM discretize Step1Results with vd_sum using EACH MODEL;
```

Explanations: The first sub-query demonstrates the use of the *FOR TRACK IN ()* construct in selecting all files corresponding to transcription factor binding peaks in a particular cell line. The union of all these peaks is stored in a temporary track called "Step1Results". Each of these peaks has a value of 1. In the second sub-query, the discretize operation is used to cut the overlapping peaks into non-overlapping regions. The number of different transcription factors having binding peaks overlapping each such region is counted by using the *vd sum* operation with the *EACH MODEL* of interval values. The final results are stored in a signal track called "Step2Results" using the second form of *CREATE TRACK*.

GMQL:

```
TF = SELECT(composite == "wgEncodeSydhTfbs" AND cell == "GM12878" AND view == "Peaks")
    HG19_ENCODE_NARROW;  # 53 samples

RES = HISTOGRAM(1,ANY) TF;
MATERIALIZE RES INTO CQ1;
```

Explanations: The same effect as the complex STQL query is obtained in GQML by a single HISTOGRAM operation (with parameters 1,ANY), counting the number of transcription factor binding peaks that overlap in each histogram interval.

Note that GMQL implicitly iterates over all 53 samples selected in the input dataset with no need of additional iterative construct, which conversely STQL requires.

<u>CQ2</u>: It identifies regions that 1) have active transcription factor binding, 2) are not within predefined promoter-proximal regulatory modules and 3) are at least 10 kb away from high-confidence annotated genes. These regions are potentially gene-distal regulatory regions.

STQL:

```
CREATE TRACK Step1Results AS
SELECT NtintA.chr, NtintA.chrstart, NtintA.chrend
FROM (`HumanMetaTracks`.`BAR_Gm12878_merged.bed`  exclusivejoin
    `HumanMetaTracks`.`PRM_Gm12878_merged.bed`) NtintA;

CREATE TRACK Step2Results AS
SELECT NtintB.chr, NtintB.chrstart, NtintB.chrend
FROM Step1Results NtintB, `wgEncodeGencode`.`gencode.v19.annotation.gtf` Tint3
```

```
WHERE Tint3.feature = 'gene' AND (Tint3.attributes LIKE '%level 1%' OR Tint3.attributes LIKE
    '%level 2%') AND distance(NtintB, Tint3) < 10000;

SELECT * FROM Step1Results exclusivejoin Step2Results;
```

Explanations: The first sub-query uses *exclusivejoin* to select regions with active transcription factor binding that are not within the pre-defined promoter-proximal regulatory regions, as provided by (Yip *et al.*, 2012). The second sub-query takes these regions and identifies those that are within 10,000 bp from any level-1 or level-2 annotated genes in “GENCODE”. The third sub-query removes the gene-proximal regions obtained in sub-query 2 from the regions obtained in sub-query 1 to get the final results.

GMQL:

```
BAR = SELECT(tableName == "BAR_Gm12878_merged") Yip_HumanMetaTracks;
PRM = SELECT(tableName == "PRM_Gm12878_merged") Yip_HumanMetaTracks;
BAR1 = COVER(1,ANY) BAR;
PRM1 = COVER(1,ANY) PRM;
U1 = UNION() BAR1 BAR1;
U = UNION() U1 PRM1;
BAR_notPRM = COVER(2,2) U;

GENE = SELECT(region: feature == "gene" AND (level == 1 OR level == 2))
    HG19_ANNOTATION_GENCODE;
BAR_notPRM_close = JOIN(DISTANCE < 10000; output: RIGHT_DISTINCT) GENE
BAR_notPRM;

UU1 = UNION() BAR_notPRM BAR_notPRM;
UU = UNION() UU1 BAR_notPRM_close;
BAR_notPRM_far = COVER(2,2) UU;
MATERIALIZE BAR_notPRM_far INTO CQ2;
```

Explanations: The repeated use of *exclusivejoin* in STQL forces the complex GMQL rewriting discussed in SQ4. Note that in Step 2 GMQL uses the output option RIGHT of the JOIN operation to reproduce the STQL SELECT clause, which takes only the first operand “NtintB.chr”, “NtintB.chrstart” and “NtintB.chrend” attribute values of the result, and that the _DISTINCT clause of GMQL output option is used to keep only one copy of the regions of the right operand (instead of the possible multiple copies created by the join operation, each one for each matching region of the left operand).

CQ3: It identifies transcription factor binding regions, in the form of 100 bp bins, that are at least 10 kb from any high-confidence annotated genes. This is another way to identify potential gene-distal regulatory regions when the binding-active regions and the promoter-proximal regulatory modules are not pre-defined and it is desirable to give 100 bp bins as outputs for further analyses.

STQL:

```
FOR TRACK T IN (category = `SYDH TFBS`, cell = 'GM12878' and fname LIKE '%Pk%')
SELECT chr, chrstart, chrend, value
FROM T
COMBINED WITH UNION ALL AS Step1Results;

CREATE TRACK Step2Results AS
SELECT NtintA.chr, NtintA.chrstart, NtintA.chrend
```

```
    FROM (project Step1Results on generate bins with length 100 with vd_sum using EACH MODEL)
        NtintA
    WHERE NtintA.value > 0;

    CREATE TRACK Step3Results as
    SELECT NtintB.chr, NtintB.chrstart, NtintB.chrend
    FROM wgencodegencode.gencode_v19_annotation Tint1, Step2Results NtintB
    WHERE Tint1.feature = 'gene' AND (Tint1.attributes LIKE '%level 1%' OR Tint1.attributes LIKE
        '%level 2%') AND distance(Tint1, NtintB) < 10000;

    SELECT *
    FROM coalesce (SELECT NtintC.chr, NtintC.chrstart, NtintC.chrend
        FROM (Step2Results exclusivejoin Step3Results) NtintC ) NtintD;
```

Explanations: The first sub-query stores all transcription factor binding peaks in a temporary track. The second sub-query maps these regions to 100 bp bins, and counts the number of transcription factors with a peak overlapping each bin. By using the .value > 0 condition, only bins with at least one binding transcription factor are kept. The third sub-query identifies the bins that are close to level-1 or level-2 “GENCODE” genes. Finally, the fourth sub-query uses an exclusive join to find bins far away from these genes, and join those that are adjacent into larger regions.

GMQL:

```
    TF = SELECT(composite == "wgEncodeSydhTfbs" AND cell == "GM12878" AND view == "Peaks")
        HG19_ENCODE_NARROW;  # 53 samples

    # BINS dataset includes a single sample with consecutive regions of fixed size (100 bp)
    BIN = SELECT(size == 100) BINS;
    BIN_TF = MAP() BIN TF;
    BIN_TF1 = SELECT(region: count_BIN_TF > 0) BIN_TF;

    GENE = SELECT(region: feature == "gene" AND (level == 1 OR level == 2))
        HG19_ANNOTATION_GENCODE;
    BIN_close = JOIN(DISTANCE < 10000; output: RIGHT_DISTINCT) GENE BIN_TF1;

    BIN_TF11 = COVER(1,ANY) BIN_TF1;
    BIN_ TF_close1 = COVER(1,ANY) BIN_close;
    U1 = UNION() BIN_TF11 BIN_TF11;
    U = UNION() U1 BIN_TF_close1;
    BIN_TF_far = COVER(2,2) U;
    RES = FLAT(1,ANY) BIN_TF_far;
    MATERIALIZE RES INTO CQ3;
```

Explanations: As in query SQ1, in GMQL we use an artificially produced annotation dataset consisting of consecutive regions (i.e., bins), each of 100 bp size, and a MAP operation to map the transcription factor binding peaks of each of the 53 selected samples of the input dataset over such annotation bins, counting the number of binding peaks of each transcription factor that overlap each bin. Note that in GMQL the MAP operation implicitly counts the regions in each sample of the second operand that overlap a region of the first operand.

Then, in GMQL we use the SELECT operation to extract, for each transcription factor, the bins with at least one overlapping binding peak. The GMQL translations of Step 3 and Step 4 were discussed in previous example queries (i.e., in queries CQ2 and SQ4; particularly, in Step 4 we use the same GMQL rewriting for STQL *exclusivejoin* as in example query SQ4). In Step 4, the STQL *coalesce* construct is translated in

GMQL by a FLAT(1,ANY) operation (FLAT is a variant of the COVER operation that returns the union of all the regions which contribute to the result of the COVER operation, see the GMQL documentation available at http://www.bioinformatics.deib.polimi.it/GMQLsystem/, particularly the *GMQL introduction to the language* document).

CQ4: It identifies genomic regions, in the form of 2000 bp bins, that overlap the binding peaks of at least 2 transcription factors. The average H3K27ac histone modification signal at each of the identified regions is then computed. Thresholding the resulting signals gives a list of regions with exceptionally strong H3K27ac signals, which could be potential super enhancers.

STQL:

```
FOR TRACK T IN (category = 'SYDH TFBS', cell = 'K562' and fname LIKE '%Pk%')
SELECT NtintA.chr, NtintA.chrstart, NtintA.chrend, NtintA.value
FROM (project T on generate bins with length 2000
    with vd_sum using EACH MODEL) NtintA
WHERE NtintA.value > 0
COMBINED WITH UNION ALL AS Step1Results;

CREATE TRACK Step2Results AS
SELECT chr, chrstart, chrend, COUNT(*) AS value
FROM Step1Results
GROUP BY chr, chrstart, chrend;

CREATE TRACK Step3Results AS
SELECT chr, chrstart, chrend
FROM Step2Results
WHERE value > 2;

CREATE TRACK Step4Results AS
SELECT NtintB.chr, NtintB.chrstart, NtintB.chrend, NtintB.value
FROM
(project `wgEncodeBroadHistone`.`wgEncodeBroadHistoneK562H3k27acStdSig.bigWig` on
    Step3Results with vd_sum using EACH MODEL) NtintB;

SELECT *
FROM Step4Results
WHERE value > 3;
```

Explanations: In the first sub-query, all peak files of transcription factor binding from a particular cell line are selected. Each of them is projected onto 2000 bp bins, so that a bin has value 1 if it overlaps with a binding peak, or value 0 if it does not. Only bins that overlap with at least one binding peak are kept. In the second sub-query, the number of transcription factors with a binding peak overlapping a bin is counted by using the COUNT() function and the GROUP BY clause. In the third sub-query, only bins that overlap with at least the binding peaks of a certain number (e.g., 2) of different transcription factors are kept. In the fourth sub-query, H3K27ac signals are mapped onto these remaining bins. Finally, in the fifth sub-query, only bins with an H3K27ac signal level larger than a threshold (e.g., 3) are kept in the output. Again, it is possible to write the STQL statements in a more compact form, but separating them into sub-queries makes each one easy to write and to understand.

GMQL:

```
K562 = SELECT(composite == "wgEncodeSydhTfbs" AND cell == "K562" AND view == "Peaks")
    HG19_ENCODE_NARROW;  # 99 samples
# BINS dataset includes a single sample with consecutive regions of fixed size (2000 bp)
BIN = SELECT(size == 2000) BINS;
BIN_K562 = MAP() BIN K562;
BIN_K562_1 = SELECT(region: count_BIN_K562 > 0) BIN_K562;

BIN_K562_2 = MERGE() BIN_K562_1;
BIN_K562_3 = GROUP(region_aggregates: bin_num AS COUNT()) BIN_K562_2;

BIN_K562_4 = SELECT(region: bin_num > 2) BIN_K562_3;

H3K27ac = SELECT(tableName == "wgEncodeBroadHistoneK562H3k27acStdSig")
    HG19_ENCODE_SIGNAL;
BIN_K562_H3K27ac = MAP(value AS AVG(score)) BIN_K562_4 H3K27ac;

RES = SELECT(region: value > 3) BIN_K562_H3K27ac;
MATERIALIZE RES INTO CQ4;
```

Explanations: For translating the query first step in GMQL, as in query SQ1, we use an artificially produced annotation dataset consisting of consecutive regions (i.e., bins) of 2000 bp each, and we use the MAP operation of GMQL to map each of the transcription factor binding peak samples over such bins; then, as in query CQ3, we use a SELECT operation to keep, for each transcription factor, the bins that overlap with at least a binding peak. At this point, in GMQL we use the MERGE operation to collect in a single sample all the retained binding peak overlapping bins in the 99 samples initially selected, and a GROUP operation on regions (in this case bins) with COUNT aggregation option to collapse all bins with the same genomic location in a single region and count, for each of such bin regions (i.e., each bin genomic location), the number of transcription factors with at least a binding peak overlapping the region; then, a SELECT operation keeps only those of such bin regions where more than two different transcription factors bind, which is the same result obtained by STQL at the end of Step3. In Step 4, similarly as in query SQ1, GMQL uses a MAP operation to map the H3K27ac histone modification signal, selected from an input dataset, on the selected bins and compute the average H3K27ac signal in each of such bins. Finally, again a SELECT operation keeps only those bins with an H3K27ac signal level higher than 3 in the output.

CQ5: It identifies genes with significant differential binding signals at their promoters in two different contexts. In each context, the binding signals are computed by subtracting the ChIP-seq signals by the corresponding background signals obtained from a control experiment.

STQL:

```
CREATE TRACK Step1Results AS
SELECT chr, chrstart, chrend, strand
FROM `wgEncodeGencode`.`gencode.v19.annotation.gtf`
WHERE feature = 'gene' and attributes LIKE '%gene_type "protein_coding"%';

CREATE TRACK Step2Results AS
SELECT DISTINCT NtintA.chr, NtintA.chrstart, NtintA.chrend
FROM (SELECT chr, chrstart - 1500 AS chrstart, chrstart + 500 AS chrend
    FROM Step1Results
    WHERE strand = '+'
    UNION ALL
```

```
        SELECT chr, chrend - 500 AS chrstart, chrend + 1500 AS chrend
        FROM Step1Results
        WHERE strand = '-') NtintA;

    CREATE TRACK Step3Results AS
    SELECT NtintB.chr, NtintB.chrstart, NtintB.chrend, NtintB.value - NtintC.value as value
    FROM (project `wgEncodeSydhTfbs`.`wgEncodeSydhTfbsGm12878JundIggrabSig.bigWig`
        on Step2Results with vd_sum using EACH MODEL) NtintB,
        (project `wgEncodeSydhTfbs`.`wgEncodeSydhTfbsGm12878InputStdSig.bigWig`
            on Step2Results with vd_sum using EACH MODEL) NtintC
    WHERE NtintB coincides with NtintC;

    CREATE TRACK Step4Results AS
    SELECT NtintD.chr, NtintD.chrstart, NtintD.chrend, NtintD.value - NtintE.value as value
    FROM (project `wgEncodeSydhTfbs`.`wgEncodeSydhTfbsK562JundIggrabSig.bigWig`
        on Step2Results with vd_sum using EACH MODEL) NtintD,
        (project `wgEncodeSydhTfbs`.`wgEncodeSydhTfbsK562InputStdSig.bigWig`
            on Step2Results with vd_sum using EACH MODEL) NtintE
    WHERE NtintD coincides with NtintE;

    CREATE TRACK Step5Results AS
    SELECT NtintF.chr, NtintF.chrstart, NtintF.chrend, NtintF.value / NtintG.value as value
    FROM Step3Results NtintF,
        (SELECT chr, chrstart, chrend, value
         FROM Step4Results
         WHERE value != 0) NtintG
    WHERE NtintF coincides with NtintG;

    CREATE TRACK Step6Results AS
    SELECT chr, chrstart, chrend
    FROM Step5Results
    WHERE value > 2;

    SELECT *
    FROM (SELECT NtintH.chr, NtintH.chrstart, NtintH.chrend, NtintH.strand
        FROM Step1Results NtintH,
            (SELECT chr, chrstart + 1500 AS chrstart, chrstart + 1500 AS chrend
             FROM Step6Results) NtintI
        WHERE NtintH.strand = '+' and NtintI is prefix of NtintH
        UNION ALL
        SELECT NtintJ.chr, NtintJ.chrstart, NtintJ.chrend, NtintJ.strand
        FROM Step1Results NtintJ,
            (SELECT chr, chrend - 1500 AS chrstart, chrend - 1500 AS chrend
             FROM Step6Results) NtintK
        WHERE NtintJ.strand = '-' and NtintK is suffix of NtintJ) NtintL;
```

Explanations: The first sub-query identifies all protein-coding genes. The second sub-query defines the promoter of each gene as the region from 1500 bp upstream of the transcription start site to 500 bp downstream of it. The two strands need to be handled in different ways. The third and fourth sub-queries compute the background-subtracted binding signals of a transcription factor at the promoters in two different cell lines. The fifth sub-query computes the fold change of the binding signal, given that the signal is non-zero in the second cell line. The sixth sub-query selects the promoters with at least a 2-fold higher binding signal in the first cell line as compared to the second one. Finally, the seventh sub-query gets back the information of the genes of these promoters.

GMQL:

```
    Step1Results = SELECT(region: feature == "gene" AND gene_type == "protein_coding")
        HG19_ANNOTATION_GENCODE;

    Step2Results = PROJECT(region_update: original_left AS left, original_right AS right,
        start AS start -1500, stop AS start + 500) Step1Results;

    T2 = SELECT(tableName == "wgEncodeSydhTfbsGm12878JundIggrabSig")
        HG19_ENCODE_SIGNAL;
    T3 = SELECT(tableName == "wgEncodeSydhTfbsGm12878InputStdSig")
        HG19_ENCODE_SIGNAL;
    T4 = SELECT(tableName == "wgEncodeSydhTfbsK562JundIggrabSig") HG19_ENCODE_SIGNAL;
    T5 = SELECT(tableName == "wgEncodeSydhTfbsK562InputStdSig") HG19_ENCODE_SIGNAL;

    T2onStep2 = MAP(value_t2 AS AVG(score)) Step2Results T2;
    T3onStep2 = MAP(value_t3 AS AVG(score)) T2onStep2 T3;
    T4onStep2 = MAP(value_t4 AS AVG(score)) T3onStep2 T4;
    T5onStep2 = MAP(value_t5 AS AVG(score)) T4onStep2 T5;

    Step6Results_0 = PROJECT(region_update: left AS original_left, right AS original_right,
        folding_value AS (value_t2 - value_t3) / (value_t4 - value_t5)) T5onStep2;
    Step6Result = SELECT(region: folding_value > 2) Step6Results_0;
    MATERIALIZE Step6Result INTO Step6Result;
```

Explanations: In steps 1 and 2, we select the protein coding genes and from those we build their promoter regions using the SELECT and PROJECT operations of GMQL, respectively. Note that in GMQL instead of “left” and “right” genomic coordinates we can use “start” and “stop” to implicitly take the region strand into account, with therefore no need of strand-specific encoding to handle the two strands in different ways as needed in STQL; conversely, the newly created “original_left” and “original_right” region attributes are used to save the original left and right coordinates of each gene, so that at the end they can be easily restored. In T2, T3, T4, and T5 we load the files of a transcription factor binding signal and of the background signal of two considered cell lines; we then use the MAP operation of GMQL to incrementally map each of such files on the built promoters, computing for each of them the average of the signal values (score) in each promoter. In so doing, the values computed in each MAP operation are incrementally added as a new attribute of each promoter, keeping always the same promoter regions.

Finally, in step 6 first, using a single PROJECT operation of GMQL, for each promoter we compute the ratio of the differences of the aggregate signal values (i.e., the fold change of the background-subtracted binding signal in the two cell lines), and we set back promoter regions to their original “left” and “right” gene positions; then, using a SELECT operation of GMQL we select only those regions with a ratio greater than 2 (i.e., those genes which have in their promoter more than a 2-fold binding signal in the first cell line as compared to the second one – cases with null signal in the second cell line are implicitly managed).

Note that the GMQL management carries on the initial gene region attributes and annotation metadata during the entire processing, providing them within the final result dataset. Furthermore, all materialized results are available within the user private repository of the GMQL system and can be used as input to other GMQL queries.

CQ6: It identifies genomic regions with bi-directional transcription at their flanking regions (Figure S1) which could be potential enhancers producing enhancer RNAs (eRNAs) (Zhu *et al.*, 2017).

STQL:

```
CREATE TRACK Step1Results AS
SELECT chr, chrstart - 200 AS chrstart, chrend - 200 AS chrend
FROM
  `wgEncodeCshlLongRnaSeq`.`wgEncodeCshlLongRnaSeqK562CellPapPlusRawSigRep1.bigWig`
WHERE value > 2;

CREATE TRACK Step2Results AS
SELECT chr, chrstart + 200 AS chrstart, chrend + 200 AS chrend
FROM
  `wgEncodeCshlLongRnaSeq`.`wgEncodeCshlLongRnaSeqK562CellPapMinusRawSigRep1.bigWig`
WHERE value > 2;

SELECT *
FROM Step1Results intersectjoin Step2Results;
```

Explanations: In the first sub-query, genomic regions on the positive strand with an expression level higher than a given value (e.g., 2) are selected. The regions are shifted 200 bp to the left, which will make the last step easy. Likewise, the second sub-query identifies regions on the negative strand with significant expression, and the regions are shifted to the right by 200 bp. Finally, in the third sub-query, the results from the first two sub-queries are intersected. For each region in the final signal track, every constituent genomic position has significant expression level 200 bp downstream on the positive strand and 200 bp upstream on the negative strand, which forms a bi-directional pattern indicative of eRNA (Zhu *et al.*, 2017).

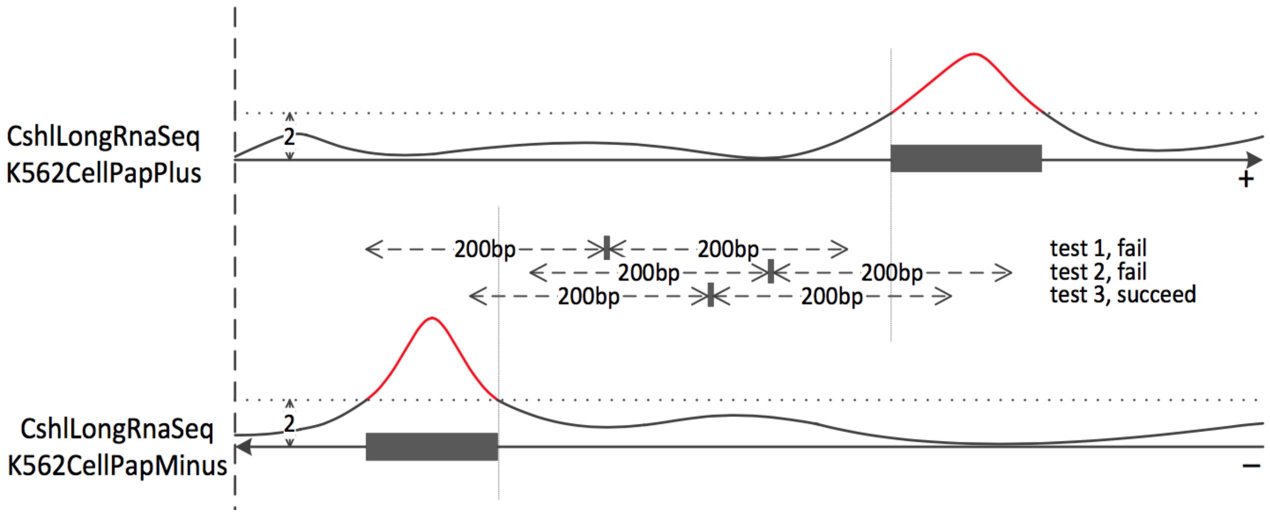

Fig. S1: Illustration of the regions to be identified in this query. RNA-seq signals on the two strands are shown in the two tracks. In the middle, three genomic regions are shown in the form of vertical bars. The first and second regions do not satisfy the query requirements since they miss significant RNA-seq signals on either side, while the third region satisfies them by having significant signals on both sides. This query identifies all regions that satisfy the requirements in the whole genome (Zhu *et al.*, 2017).

GMQL:

```
K562_PLUS = SELECT(tableName == "wgEncodeCshlLongRnaSeqK562CellPapPlusRawSigRep1";
    region: score > 2) HG19_ENCODE_SIGNAL;
K562_PLUS_UPDATE = PROJECT(region_update: left AS left - 200, right AS right - 200)
    K562_PLUS;
```

```
K562_MINUS = SELECT(tableName == "wgEncodeCshlLongRnaSeqK562CellPapMinusRawSigRep1";
    region: score > 2) HG19_ENCODE_SIGNAL;
K562_MINUS_UPDATE = PROJECT(region_update: left AS left + 200, right AS right + 200)
    K562_MINUS;

RES = JOIN(DISTANCE < 0; output: INT) K562_PLUS_UPDATE K562_MINUS_UPDATE;
MATERIALIZE RES INTO CQ6;
```

Explanations: The translation of this query into GMQL is straightforward, as in GMQL the PROJECT operation allows performing the intended region shifts and the JOIN(DISTANCE < 0) operation identifies intersecting regions, with the INT output region construction option that builds the region intersections.

S1.2 Example of GMQL query that cannot be expressed in STQL

Consider the following biological problem: *"From the entire ENCODE BROADPEAK dataset, combine all available H3K4me1 and H3k27ac histone modifications to identify, for each cell line in ENCODE, putative active enhancers, i.e., regions of the genome where both a peak of H3K4me1 and a peak of H3k27ac are present. Next, combine the result for the various cell lines to identify those enhancers that are specifically present in only one of them."*

To provide answer to this problem, the following GMQL query can be built. First, using a SELECT operation it selects all the ChIP-seq experiment samples in the ENCODE BROADPEAK dataset that target the H3K4me1 histone modification and, by means of a COVER operation, combines their replicas in case available for each cell line. The same operation is then performed for the ChIP-seq experiment samples targeting the H3K27ac histone modification. Then, the query identifies putative active enhancers as the genomic regions where a H3K4me1 peak overlaps with a H3K27ac peak in the same cell line; in order to do so, first we use a JOIN operation of GMQL with the "cell" sample metadata attribute as *joinby* key. The result is a set of samples, one for each cell line, where each sample contains the list of putative active enhancer regions for a cell line; a PROJECT operation copies the value of the sample metadata attribute "*cell*", describing the cell line, within a new attribute "*cell*" of every region, so that each region in the output is labeled with the name of the cell line of origin. Finally, we select only those putative active enhancer regions of each cell line that do not overlap any other putative active enhancer of any cell line. We do this using a COVER(1,1) operation of GMQL, with parameters min and max accumulation equal to 1; it outputs a single sample with all the cell line specific putative active enhancers extracted, each one labeled with its cell line name.

```
me1 = SELECT(antibody_target == "H3K4me1") HG19_ENCODE_BROAD;
me1_c = COVER(1,ANY; groupby: cell) me1;
ac = SELECT(antibody_target == "H3K27ac") HG19_ENCODE_BROAD;
ac_c = COVER(1,ANY; groupby: cell) ac;

active = JOIN(DISTANCE < 0; output: INT; joinby: cell) me1_c ac_c;
labeled = PROJECT(region_update: cell AS META(me1_c.cell, STRING)) active;

cell_specific = COVER(1,1; aggregate: cell_line AS BAG(cell)) labeled;
MATERIALIZE cell_specific INTO cell_specific;
```

The above query cannot be expressed in STQL, as it uses sample metadata predicates in the various "*joinby*" and "*groupby*" clauses, which partition the COVER and JOIN operands by distinct values of the specified sample metadata attributes, in this case the "cell" attribute. Thus, they enable GMQL to take full advantage of parallel processing and to apply on big data performing implicit iterations on specific sample pairs matched through their metadata. Two other features of GMQL that cannot be translated to STQL are: the inclusion of a sample metadata attribute in each region of the sample as a region attribute (which in GMQL is possible using the *region_update* option of the PROJECT operation together with the META() function, which takes the value of the specified sample metadata attribute), and the construction of a BAG of region attribute values of the regions that contribute to a result region of the COVER operation. Unsupported features in STQL are highlighted in bold in the above GMQL query. Other similar examples are also available in the GMQL documentation at http://www.bioinformatics.deib.polimi.it/GMQLsystem/, in the *GMQL example queries* and in the *Biological examples* documents (particularly in the latter document, queries Q4, Q6 and Q8; the latter one is the example use case GMQL query in Section 4 of our article, which includes sample metadata predicates not supported in STQL).

References

Goecks,J. *et al*. (2010) Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biol., **11**, 86.

Masseroli,M. *et al*. (2015) GenoMetric Query Language: A novel approach to large-scale genomic data management. *Bioinformatics*, **31**(12), 1881-1888.

Olston,C. *et al*. (2008) Pig Latin: A not-so-foreign language for data processing. In Lakshmanan L.V.S. *et al*. (ed.), *Proc. 2008 ACM SIGMOD Int. Conf. on Management of Data*. ACM, New York, NY, pp. 1099-1110.

Quinlan,A.R. and Hall,I.M. (2010) BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, **26**(6), 841-842.

Zhu,X. *et al*. (2017) START: a system for flexible analysis of hundreds of genomic signal tracks in few lines of SQL-like queries. *BMC Genomics*, **18**(1), 749.

Yip,K.Y. *et al*. (2012) Classification of human genomic regions based on experimentally-determined binding sites of more than 100 transcription-related factors. *Genome Biol*., **13**, 48.

S2 Example use case output metadata

In Table S2, we report an excerpt of the result metadata of the GMQL example use case query (as from Section 4 of our article). Note that these attributes and their values describe the patients who contribute to the result, each originally described by a different sample; metadata are transferred from the source datasets to the result datasets thanks to GMQL operations. Data interpretation is much facilitated by the availability of region data and metadata samples, where the latter include relevant information either computed by the query (e.g., the order and count attributes) or present in the source (e.g., the age and therapy attributes).

Table S2. Metadata excerpt of the top 5% patients finally selected with the example use case GMQL query

order	mutation count	age at initial pathologic diagnosis	radiation therapy	estrogen receptor	progesterone receptor
1	120	90	no	positive	negative
2	42	63	no	positive	positive
3	36	68	no	positive	positive
4	26	83		positive	negative
5	25	61	yes	negative	negative
6	25	47	yes	positive	positive
7	24	60	yes	negative	negative
8	24	81	yes	negative	negative
9	22	55	yes	negative	negative
10	22	76	no	positive	negative
11	20	69	no	negative	negative
12	20	74	no	positive	positive
13	19	77	yes	positive	
14	19	50	yes	positive	positive
15	19	77	no	positive	positive
16	16	90	no	negative	negative
17	14	59	no	positive	positive
18	14	41	yes	positive	positive
19	13	68	yes	positive	positive
20	13	64	yes	positive	positive
21	12	59	no	negative	negative
22	11	69	yes	positive	positive
23	11	75	no	positive	positive
24	10	40	yes	negative	negative
25	9	44	yes	positive	positive
26	8	57	no	positive	negative
27	8	43		positive	positive
28	8	63	no	positive	positive
29	8	45	no	positive	positive
30	8	69	yes	positive	negative

S3 Scalable performance evaluation of the example use case

To demonstrate the scalable performance of our GMQL system, we performed the analysis of scalability and performance of the GMQL example use case in manuscript Section *4 Example use case*. First we broke down the input *METHYLATION* dataset (the biggest one of the three input datasets in the GMQL example use case, including 1,234 samples of DNA methylation data for a total of 358,803,211 methylation regions and 122.7 GB of data); we decomposed it in subparts of increasing sizes (i.e., of 1/16, 1/8, 1/4, 1/2, and 1/1 of the total number of samples of the entire *METHYLATION* dataset). Then, we used each of such subparts as input methylation data in subsequent runs of the same GMQL example use case. The obtained results are shown in Figure S2, showing the GMQL example use case performance as a function of the number of samples of the input methylation dataset. The figure shows a very good scale-up of the system performance with the increase of samples, particularly evident for higher data sizes (very similar are the performance trends as a function of the number of regions or of the size of the input methylation dataset, as in the input methylation data the number of regions and size grow almost proportionally to the number of samples). Table S3 reports the detailed quantitative values of each evaluated input methylation data case and of the related running time.

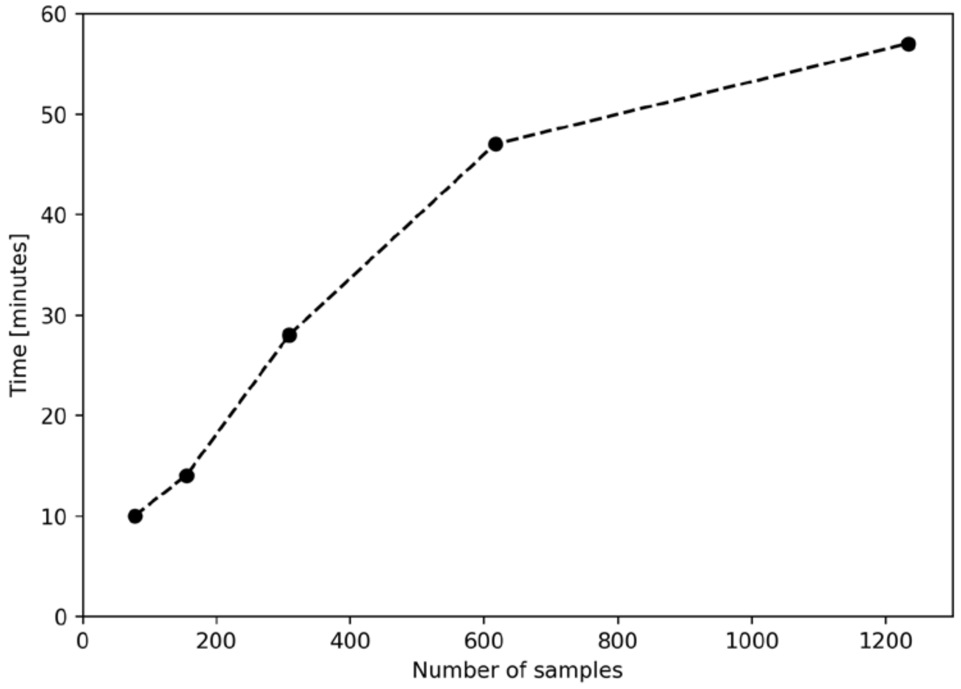

Fig. S2: Execution time of GMQL example use case as a function of number of input DNA methylation samples.

Table S3. Execution time of GMQL example use case vs. quantitative features of the input methylation dataset

Methylation Dataset	Number of Samples	Number of Regions	Size (GB)	Time (minutes)
METHYLATION 1/1	1234	358,803,211	122.7	57
METHYLATION 1/2	617	177,172,167	60.6	47
METHYLATION 1/4	309	87,119,413	29.8	28
METHYLATION 1/8	155	43,203,139	14.8	14
METHYLATION 1/16	78	22,540,997	7.7	10