SOFTWARE

START: A system for flexible analysis of hundreds of genomic signal tracks in few lines of SQL-like queries

Xinjie Zhu[1], Qiang Zhang[2], Eric Dun Ho[3], Ken Hung-On Yu[3,4], Chris Liu[3], Tim H. Huang[5], Alfred Sze-Lok Cheng[6], Ben Kao[1], Eric Lo[3] and Kevin Y. Yip[3,7,8,9*]

[*]Correspondence: kevinyip@cse.cuhk.edu.hk
[3]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong
Full list of author information is available at the end of the article

Abstract

Background: A genomic signal track is a set of genomic intervals associated with values of various types, such as measurements from high-throughput experiments. Analysis of signal tracks requires complex computational methods, which often make the analysts focus too much on the detailed computational steps rather than on their biological questions.

Results: Here we propose Signal Track Query Language (STQL) for simple analysis of signal tracks. It is an Structured Query Language (SQL)-like declarative language, which means one only specifies *what* computations need to be done but not *how* these computations are to be carried out. STQL provides a rich set of constructs for manipulating genomic intervals and their values. To run STQL queries, we have developed the Signal Track Analytical Research Tool (START, `http://yiplab.cse.cuhk.edu.hk/start/`), a system that includes a Web-based user interface and a back-end execution system. The user interface helps users select data from our database of around 10,000 commonly-used public signal tracks, manage their own tracks, and construct, store and share STQL queries. The back-end system automatically translates STQL queries into optimized low-level programs and runs them on a computer cluster in parallel. We use STQL to perform 14 representative analytical tasks. By repeating these analyses using bedtools, Galaxy and custom Python scripts, we show that the STQL solution is usually the simplest, and the parallel execution achieves significant speed-up with large data files. Finally, we describe how a biologist with minimal formal training in computer programming self-learned STQL to analyze DNA methylation data we produced from 60 pairs of hepatocellular carcinoma (HCC) samples.

Conclusions: Overall, STQL and START provide a generic way for analyzing a large number of genomic signal tracks in parallel easily.

Keywords: Human genomics; Signal tracks; Data analysis

Background

The rapid development of new applications of high-throughput sequencing and the sharp reduction of cost have made it common to produce large amounts of sequencing data that measure a variety of biological signals in a single study. For instance, large-scale disease studies can involve the sequencing of hundreds or even thousands of disease and control samples [1]. Major collaborative projects such as Encyclopedia of DNA Elements (ENCODE) [2] and Roadmap Epigenomics [3] have performed tens of thousands of high-throughput sequencing experiments that survey the genomes, transcriptomes and epigenomes of a large number of samples, creating rich and complex sets of data.

After standard data processing, sequencing data are commonly represented as signal tracks. A signal track is a set of genomic intervals each associated with a signal value. Depending on the analytical needs, the intervals can be defined in various ways. For example, when the data from a ChIP-seq (chromatin immunoprecipitation followed by high-throughput sequencing) experiment are represented as a signal track, at the basic level, each interval corresponds to a single genomic location and the associated value is the number of aligned reads that cover the location. At the next level, one could use the distribution of signal values to define signal peaks, and consider each peak as an interval with a fixed value of one (which means "present") or a value that indicates the enrichment score of the peak as compared to control. One could also use a gene annotation set to define intervals of interest (e.g., promoters), and compute the average number of covering reads at each interval as its signal value. In each of these three cases, the ChIP-seq data are represented by a signal track. The generality of representing high-throughput sequencing data by signal tracks is exemplified by its prevalent use in genome browsers for displaying many types of sequencing data.

Analysis of signal tracks usually involves multiple steps. Typical operations at each step include selection of intervals based on certain criteria, comparison of intervals from the same or different tracks, and aggregation of multiple intervals to form new intervals. There are software tools for particular types of operation, and pipelines can be set up by writing scripts that invoke the different tools and convert the outputs of one tool into the inputs of another [4–6].

As the volume and complexity of signal track data have both increased dramatically in recent years, this paradigm of data analysis is facing several challenges. First, many existing tools have a fixed set of functions. When they do not exactly match the needs of an analytical pipeline, one would need to modify a tool or implement a new one. Second, pipelines are usually developed in an imperative language. Researchers are required to specify the detailed computational steps, which could distract him/her from focusing on the biological questions. Third, in order to perform analysis efficiently, a researcher needs to decide on proper data structures, algorithms and parallel execution environments, which impose a strong requirement on his/her computational backgrounds.

With a goal of providing a single platform that can support a large variety of analytical needs, here we describe the Signal Track Query Language (STQL) that we specifically designed for signal track data analysis. It is a declarative language with a syntax similar to the Structured Query Language (SQL) commonly used in relational database systems, which makes STQL easy to learn. Users only need to specify what operations they want to perform using some high-level constructs, but not the detailed steps of how these operations are to be performed, thereby allowing them to focus on the analytical goals rather than the technical details.

To demonstrate the broad applications of STQL, we have implemented a system for executing STQL queries called Signal Track Analytical Research Tool (START, `http://yiplab.cse.cuhk.edu.hk/start/`). It contains a Web interface that guides users to construct STQL queries, and provides example queries for various types of data analysis. At the back end, the submitted queries are automatically translated into executable programs, which are then run on a cluster of machines in parallel. START provides a variety of pre-loaded public data that facilitate integrated analysis of both public and private data, including data from ChromHMM [7], dbSNP [8], ENCODE, FANTOM5 (Functional Annotation of The Mammalian Genome Phase 5) [9], RoadMap Epigenomics, UCSC Genome Browser[10] and Yip et al.[11]. START also provides storage for both users' data files and executed queries, allows sharing of queries among users, and contains features for protecting security and data privacy. Users who want to execute STQL queries locally on their own machines can download our installable package, with a detailed installation guide at `https://github.com/stql/start/wiki/Install-START-in-your-own-cluster` describing the steps for installing the package, pre-processing data, loading signal tracks, organizing the tracks and using the system.

The flexibility of analyzing genomic data with a query language has been clearly demonstrated in a number of recent studies [4, 12–16]. Most of these languages were designed for raw sequencing reads and cannot be used for analyzing signal tracks. GenoMetric Query Language [4] is a language designed for signal track analysis. Compared to this language, STQL provides a large set of interval comparison relations that help simplify queries, constructs for manipulating signal values (based on the **EACH MODEL** and **TOTAL MODEL**), several types of loop statements, complex queries such as those involving sub-queries, the **discretize** operation for creating non-overlapping intervals, and has an SQL-like design that makes it easier to learn for people with SQL experience.

In the followings we describe the different components of STQL and how it can be used to analyze genomic signal tracks. We present illustrative example queries that correspond to commonly performed analytical operations. These example queries include both simple ones that show individual language features of STQL, as well as composite ones that involve multiple steps.

To evaluate the correctness, simplicity and execution efficiency of STQL, we used several other popular approaches to carry out the same analytical tasks, including bedtools [5], Galaxy [6] and custom Python scripts we specifically wrote for these tasks. We show that many of these tasks are most easily carried out by using STQL, and for tasks involving large data files, the transparent parallelization of STQL provided by START leads to significant speed-ups.

We further demonstrate the usability of STQL by describing how a biologist with minimal training in computer programming self-learned STQL to identify genes affected by differential promoter methylation by integrating private sequencing data from 60 pairs of hepatocellular carcinoma (HCC) case-control samples and public signal tracks. The STQL queries written serve as a succinct log of the analyses taken, allowing anyone to reproduce the same results and apply the pipeline to other data sets easily.

Results

Data model of STQL

In STQL, each track is composed of a set of intervals all with the same attributes (possibly with null values). Each interval contains four mandatory attributes, namely its chromosome ('.chr'), starting position ('.chrstart', one-based inclusive), ending position ('.chrend', inclusive), and value ('.value'). Each signal track can define any number of additional attributes for its intervals. For example, a .strand attribute can be defined to contain the strand of each interval, with values '+', '−' and '.' for the positive strand, negative stand, and don't care/not available, respectively. Each interval is therefore equivalent to a tuple in a relational table with a list of single-valued attributes.

Basic constructs in STQL

The formal grammar of STQL is given in the Methods section. Basically, each STQL query contains three main parts, namely a SELECT clause for specifying interval attributes to be included in the results, a FROM clause for the signal tracks to query from, and an optional WHERE clause for criteria for filtering intervals. For example, the following query returns all attributes of the intervals on chromosome 1 from a signal track T:

```
SELECT    *
FROM      T
WHERE     T.chr = 'chr1';
```

If the query result includes all the four mandatory attributes, it will be considered a signal track itself and can be used as an input track of another query.

The SELECT clause

The SELECT clause includes a comma-separated list of attributes of the queried intervals to be returned, which can include both the four mandatory attributes and any of the additional attributes defined for the tracks involved. STQL also supports other syntactic constructs commonly used in the SELECT clause of SQL, such as the DISTINCT keyword for removing duplicates, standard arithmetic operations (addition, subtraction, multiplication and division), and the AS keyword for renaming attributes. As in SQL, if the signal track from which an attribute comes is unambiguous, the attribute can be listed without stating the track name. For example, the following query returns the set of distinct interval lengths for the intervals in a track T:

```
SELECT    DISTINCT chrend − chrstart + 1 AS len
FROM      T;
```

Since interval lengths are commonly queried in analysis tasks, STQL also defines a short-hand ("syntactic sugar") for it, allowing the above query to be written in a simpler form:

```
SELECT    DISTINCT length(T) AS len
FROM      T;
```

The FROM clause

The FROM clause contains a comma-separated list of signal tracks to query from. Each listed track can be an existing signal track in the database, a nested query (described below), or a track dynamically generated using one of the track operations to be described in the section on advanced constructs.

In STQL, conceptually a Cartesian product of the listed tracks is performed in a chromosome-by-chromosome manner, since intervals from different chromosomes are seldom directly compared. For example, suppose we have the following two tracks T_1 and T_2:

T_1			
chr	chrstart	chrend	value
chr1	101	200	10
chr1	201	300	20
chr2	301	400	30

T_2			
chr	chrstart	chrend	value
chr1	401	500	40
chr2	501	600	50
chr3	601	700	60

Suppose the following query is issued to identify all pairs of intervals on the same chromosome from the two tracks:

SELECT T_1.chr, T_1.chrstart, T_1.chrend, T_1.value,
T_2.chr AS chr2, T_2.chrstart AS chrstart2,
T_2.chrend AS chrend2, T_2.value AS value2
FROM T_1, T_2;

The query results will be as follows:

chr	chrstart	chrend	value	chr2	chrstart2	chrend2	value2
chr1	101	200	10	chr1	401	500	40
chr1	201	300	20	chr1	401	500	40
chr2	301	400	30	chr2	501	600	50

The results do not involve any intervals from chromosome 3, because T_1 does not contain any interval on this chromosome. One could also use LEFT JOIN, RIGHT JOIN and OUTER JOIN to include intervals on chromosomes that appear only in the first, second or either of the two joining tracks. For example, suppose RIGHT JOIN is used in the previous query:

SELECT T_1.chr, T_1.chrstart, T_1.chrend, T_1.value,
T_2.chr AS chr2, T_2.chrstart AS chrstart2,
T_2.chrend AS chrend2, T_2.value AS value2
FROM T_1 RIGHT JOIN T_2 ON T_1.chr=T_2.chr;

Then the query results will be as follows:

chr	chrstart	chrend	value	chr2	chrstart2	chrend2	value2
chr1	101	200	10	chr1	401	500	40
chr1	201	300	20	chr1	401	500	40
chr2	301	400	30	chr2	501	600	50
NULL	NULL	NULL	NULL	chr3	601	700	60

In our actual implementation, instead of performing the costly Cartesian product followed by filtering the pairs that satisfy the conditions specified in the WHERE clause, the intervals in each track are sorted and compared directly to produce the list of pairs that satisfy the conditions.

As in SQL, if a signal track T appears in the FROM clause, writing T.chr means the chromosome of an instance (i.e., an interval) on track T. To make the meaning of the query clearer, one could give an alias to each track by appending the alias after the track name in the FROM clause. For instance, the interval length example given above can also be written as follows:

SELECT DISTINCT **length(**TInt**)** AS len
FROM T TInt;

By using the alias TInt, it is clear that the query returns the lengths of the intervals in the signal track as its results. We recommend adding aliases in this way since the resulting queries are easier to understand, but syntactically the aliases are not mandatory.

The WHERE clause

The WHERE clause contains a logical expression that specifies which intervals should be kept in the results. The logical expression can be composed of primitive expressions joined together by standard logical operators AND, OR and NOT. As in SQL, each primitive expression can involve a mathematical equality or inequality (e.g., $\textbf{length}(\mathrm{TInt}) < 1000$). In addition, since in many analysis tasks, different genomic intervals are compared to determine the ones to be included in the final results, a list of common relations are defined in STQL to express the positional relationships among intervals. Table 1 lists the formal definitions of these interval relations, and provides an example use of each relation. If additional relations are needed in a certain task, they can be constructed in STQL queries using the primitive constructs.

The input intervals of these relations can be intervals selected from a signal track or constant intervals specified in the format "[<chr>, <chrstart>, <chrend>]" such as "[chr1, 100, 200]".

Among these interval relations, **is upstream of** and **is downstream of** have the most complex definitions since they involve strand information. As in the usual sense, one can define an interval I_1 to be upstream/downstream of another interval I_2 only if the strand of I_2 is known and the strand of I_1 is either the same as I_2 or is not available.

Since it is common to analyze genomic distances, there is also a function **distance()** defined in STQL for computing the distance between two genomic intervals in the WHERE clause:

$$\textbf{distance}(\mathrm{I}_1, \mathrm{I}_2) = \begin{cases} \mathrm{I}_2.\mathrm{chrstart} - \mathrm{I}_1.\mathrm{chrend} & \text{if } \mathrm{I}_1 \textbf{ precedes } \mathrm{I}_2 \\ 0 & \text{if } \mathrm{I}_1 \textbf{ overlaps with } \mathrm{I}_2 \\ \mathrm{I}_1.\mathrm{chrstart} - \mathrm{I}_2.\mathrm{chrend} & \text{if } \mathrm{I}_1 \textbf{ follows } \mathrm{I}_2 \\ \mathrm{NaN} & \text{if } \mathrm{I}_1.\mathrm{chr} \neq \mathrm{I}_2.\mathrm{chr} \end{cases}$$

One frequently used operation more difficult to define using the primitive constructs is finding out the interval(s) closest to a given interval. In STQL, the **is closest to each** relation is defined for this purpose, as shown in the following example:

SELECT $*$
FROM T_1 TInt_1, T_2 TInt_2
WHERE TInt_1 **is closest to each** TInt_2;

In this example, for each interval in T_2, we find its closest interval among all intervals in T_1. The result can contain zero intervals (if no intervals in T_1 are on that chromosome), one interval, or more than one interval (if multiple intervals in T_1 are of exactly the same closest distance from it).

Other optional clauses

Similar to SQL, STQL provides a GROUP BY clause for grouping intervals and performing aggregations (**COUNT()**, **SUM()**, **AVG()**, **MIN()** and **MAX()**) for each group, and an ORDER BY clause for ordering the selected intervals. For example, the following query counts the number of intervals with a value larger than 10 on each chromosome, with the resulting counts sorted in ascending order:

```
SELECT     TInt.chr, COUNT(*) AS intervalcount
FROM       T TInt
WHERE      TInt.value > 10
GROUP BY TInt.chr
ORDER BY intervalcount;
```

The basic constructs described above are sufficient for many simple analyses. On the other hand, some analyses can be more easily performed with the help of additional constructs. We next describe these advanced constructs defined in STQL.

Advanced constructs in STQL

Creating a new track from an existing track

In an analysis pipeline, it is common for an intermediate step to create small intervals that can overlap or be adjacent to each other. These small regions are subsequently merged into longer regions in later steps. For example, suppose in an analysis step individual transcriptional enhancers are identified, and in the next step the overlapping or adjacent enhancers are to be merged to form potential super enhancers [17]. This type of interval merging can be performed by using the **coalesce** construct, which groups each set of transitively overlapping/adjacent intervals into a single interval, where the starting and ending positions of this resulting interval are respectively the smallest starting position and largest ending position of this group of intervals. The **coalesce** operator can be used in the FROM clause with the following syntax:

FROM **coalesce** T [**with** <vd> **using** <value-model>]

where T is the input track (the individual enhancers), and the optional "**with** <vd> **using** <value-model>" part is for deriving the value of each resulting interval based on the mathematical operation <vd> and value model <value-model>. STQL has a highly flexible design for value derivation that distinguishes itself from other existing languages, the details of which will be discussed shortly. The output of this operation is a new track that contains the merged intervals. An illustration of the **coalesce** operator is given in Figure 1. Complete query examples using **coalesce** and other advanced constructs will be given later.

Another common operation for processing overlapping regions is to use their boundary locations to define discrete intervals that can be adjacent to each other (Figure 2). This is useful when the next analysis step requires all intervals to be non-overlapping, for example when each genomic location should be classified as either within an interval (such as a protein binding site) or not. In STQL, this type of operations can be performed by using the **discretize** operator in the FROM clause:

FROM **discretize** T [**with** <vd> **using** <value-model>]

Creating a new track from two existing tracks

The FROM and WHERE clauses together allow for some basic joins of multiple signal tracks. To make more advanced types of track joins easy to perform, STQL provides convenient constructs for them.

In the first type of advanced track joins, a track T_2 defines the positional information of the resulting intervals and another track T_1 defines their values (Figure 3).

This is most typically used when T_2 corresponds to gene annotations, T_1 is a signal track of experimental values, and the goal is to compute an aggregated signal value for each gene based on the experimental data. In STQL, this type of operations is described as projecting T_1 on T_2 in the FROM clause:

FROM **project** T_1 **on** T_2 [**with** <vd> **using** <value-model>[,**metadata**]]

where the optional "**metadata**" part is for specifying whether non-default attributes of the input intervals are to be inherited by the resulting intervals, which will be explained later.

It is often useful to partition the whole genome into bins of a fixed size, and compute an aggregated signal value for each bin. By choosing a suitable bin size, the signals are smoothed locally and some downstream tasks can be carried out more efficiently due to the reduced data resolution and easily computable bin locations. This binning operation can be performed in STQL by projecting a signal track on a bin track dynamically created using the **generate bins with length** construct in the FROM clause:

FROM **project** T **on generate bins with length** <bin-size> [**with** <vd>
using <value-model>[,**metadata**]]

where <bin-size> is the size of each bin in base pairs. The output intervals are adjacent bins of this size covering the whole genome.

Two different signal tracks are usually compared to find out genomic locations covered by both tracks, one track but not the other, or either track. STQL supports these operations by the **intersectjoin** , **exclusivejoin** and UNION ALL constructs.

intersectjoin considers every pair of overlapping intervals from the two input tracks, and takes their intersection as a resulting interval (Figure 4). It can be used in the FROM clause:

FROM T_1 **intersectjoin** T_2 [**with** <vd> **using** <value-model>[,**metadata**]]

exclusivejoin considers every interval from the first input track, and removes all parts of it that overlap any intervals in the second input track (Figure 5):

FROM T_1 **exclusivejoin** T_2 [**with** <vd> **using** <value-model>[,**metadata**]]

Finally, UNION ALL forms a new track that keeps all intervals from the two input tracks without removing duplicates. The tracks involved must have the same schema. It can be used to join the resulting tracks of two queries. Since the result of UNION ALL is also a signal track, it can be repeatedly applied to join the resulting track with another signal track. For example, the following query takes the union of three signal tracks to form a new track (where the alias NtInt stands for "new track interval"):

```
SELECT    *
FROM      (
          SELECT * FROM T1
          UNION ALL
          SELECT * FROM T2
          UNION ALL
          SELECT * FROM T3) NtInt;
```

Value derivation and inheritance of metadata

All advanced constructs described above allow the derivation of values for the resulting intervals. Having a flexible way to manipulate interval values is crucial to many types of analysis. In STQL, two value models are used for interpreting and deriving signal values. In the **EACH MODEL**, each genomic location within an interval is considered to individually own the signal value of the interval. For example, if signal values represent normalized read counts, an interval having a certain value means that every genomic location in the interval is covered by that number of reads on average. On the other hand, in the **TOTAL MODEL**, all genomic locations within an interval is considered to collectively own the signal value of the interval. For example, if the signal value indicates the total number of MBDCap-seq [18] reads aligned to an interval, all genomic locations of the interval collectively own the signal value. The following query projects these intervals onto 100bp bins, and computes the raw signal of each bin based on the number of bases overlapping with the bin as a fraction of the interval length:

```
SELECT   *
FROM     (project T on generate bins with length 100
with vd_sum using TOTAL MODEL) NtInt
WHERE    NtInt.value > 0;
```

For STQL operations that involve the creation of intervals described above, the value of each resulting interval is determined by the specified value model and mathematical operation. In general, the value of each resulting interval is derived in three steps:

1 For each interval in the input tracks, the signal value at each of its genomic locations is determined.

2 For each interval in the resulting track, the signal value at each of its genomic locations is computed based on the values at the same location of the input intervals computed in Step 1.

3 For each interval in the resulting track, a final value is computed by aggregating the values of its genomic locations computed in Step 2.

For Step 1, if the **EACH MODEL** is used, the value at each genomic location is simply the value of the corresponding interval. On the other hand, if the **TOTAL MODEL** is used, each genomic location is given an equal share of the value of the interval.

Step 2 depends on the exact STQL operation being performed, the details of which will be explained next.

Step 3 computes the average over all values of the genomic locations within the resulting interval.

For example, suppose in Figure 4 every interval in the two input tracks has value 1, and the two tracks are joined using the **intersectjoin** construct with the **vd_sum** operation, which adds up values from different intervals location by location in Step 2 of value derivation. If the **EACH MODEL** is used, the values of I_{r1}, I_{r2}, I_{r3} and I_{r4} will all be 2. This is because in Step 1, every genomic location of the input intervals receives a value of 1; In step 2, every genomic location of the resulting intervals is given a value of 1+1=2; In Step 3, since every location in each resulting interval has the same value, taking the average will give the same value of 2.

On the other hand, if the **TOTAL MODEL** is used, the values of the resulting intervals will depend on the lengths of the intervals. For example, the value of I_{r1} will be I_{11}.value/**length(**I_{11}**)** + I_{21}.value/**length(**I_{21}**)**, since the two fractional values are respectively given to each genomic location of I_{11} and I_{21} in Step 1, and Steps 2 and 3 are similar to the case for the **EACH MODEL**.

Table 2 shows the full list of mathematical operations in STQL and how the value of each genomic location of the resulting interval is computed in Step 2. The operations provided include 1) arithmetic operations (summation, averaging, subtraction, multiplication and division), 2) maximum and minimum function, and 3) direct copying of values from the interval from input track 1 or track 2.

For **intersectjoin** , each resulting interval is formed by exactly two intervals one from each input track, and thus all nine types of operation are well-defined. For **exclusivejoin** , each resulting interval is formed by one interval from the first input track and zero, one or more intervals from the second track. Only the unary operator **vd_left** is applicable. For **coalesce** and **discretize** , only one track is involved, while for **project on** , all values come from track 1. For these three constructs, each resulting interval can be formed by one, two or more than two input intervals. Without a defined order of these intervals, the **vd_diff**, **vd_quotient**, **vd_left** and **vd_right** operations cannot be defined and are thus not allowed.

If the value model and mathematical operation are not specified, the resulting intervals will be given the value NULL.

Each interval may contain additional attributes that are called metadata, such as the name of a gene and the confidence score of a signal peak. For some of the interval-creating constructs, these metadata can be inherited from the input intervals to the resulting intervals using "**metadata**". For **project on** , the metadata are inherited from the input intervals in the second track, the track that defines the positional information of the resulting intervals. For **intersectjoin** and **exclusivejoin** , the metadata are inherited from input intervals in the first track.

Using dynamically created tracks

In the FROM clause, in addition to using existing tracks in the database, one could also create new tracks dynamically using either a nested query or one of the above track operations. For example, the following query first takes the **intersectjoin** of two tracks, and then selects out the resulting intervals with a value larger than 2:

```
SELECT    *
FROM      (T1 intersectjoin T2
          with vd_sum using EACH MODEL) NtInt
WHERE     NtInt.value > 2
```

An alias is given to the intervals of the dynamically created track, which can then be referred to in the SELECT and WHERE clauses.

Data definition and manipulation statements

STQL also contains statements for creating and deleting signal tracks, and loading data into a signal track from a local file.

The CREATE TRACK statement is used to create a new track and add it to the database. It has two different forms:

```
CREATE TRACK <track-name> (<attribute-name1>
<data-type1> [,...]);
   CREATE TRACK <track-name> AS <query>;
```

In the first form, a new empty track is created with the name specified at the placeholder <track-name>. The list of attributes and their data types (string, int or float) are then listed within the brackets. In the second form, an STQL query is executed and the result is stored as a new track with the name specified at <track-name>. If the query results do not form a valid signal track, i.e., it does not have all the required attributes for a signal track, an error will be produced when a query tries to use the query results as a track. This second form of CREATE TRACK is particularly useful when multiple STQL statements are submitted in the same block on the START Web site, where the intermediate results produced by a step are stored in a temporary signal track using a CREATE TRACK statement, which can then be accessed by the queries in the subsequent steps.

Conceptually tracks created by a CREATE TRACK statements persist in the database, but the ones created through the START Web interface (described below) are automatically removed after a certain amount of time to control the space used by each user.

The DROP TRACK statement deletes a track in the database:

```
DROP TRACK <track-name>;
```

Execution of this statement requires the user to have the corresponding permission. There are other security measures in STQL that will be explained when we describe START in detail. The DROP TRACK statement is commonly used to remove intermediate tracks created by the CREATE TRACK statement that are no longer needed.

STQL also allows loading data into a track by using the LOAD DATA LOCAL INPATH INTO TRACK statement, for example after a new track is created using the first form of the CREATE TRACK statement:

```
LOAD DATA LOCAL INPATH <file-path> [OVERWRITE]
INTO TRACK <track-name>;
```

where <file-path> is the path of the data file, <track-name> is the name of the track into which the data are to be loaded, and the OVERWRITE option is for specifying whether any existing data in the track are to be removed.

Selection and looping over signal tracks

A final feature of STQL, which is very useful when analyzing a large number of signal tracks, is selecting tracks based on their attributes, and looping over the selected tracks for repeating some operations. This feature is provided by the FOR TRACK IN () statement with two forms:

```
FOR TRACK <track-variable> IN (category=<track-category>,
<track-selection-conditions>)
          <STQL-query>
COMBINED WITH UNION ALL AS <output-track-name>;
FOR TRACK <track-variable> IN (category=<track-category>,
<track-selection-conditions>)
CREATE TRACK <output-track-name> AS <STQL-query>;
```

In both forms, <track-variable> is a variable for the intervals of a selected track in the STQL query, <track-category> is the category of signal tracks to be selected, <track-selection-conditions> states extra conditions for track selection, <STQL-query> is the query to be performed on each selected track, and <output-track-name> is the name of the track to store the results.

Specifically, <track-selection-conditions> is a list of attribute names and values delimited by "and". For example, if one wants to select all ChIP-seq binding peaks in the GM12878 cell line produced by the ENCODE Stanford/Yale/Davis/Harvard (SYDH) sub-group, and stores the union of all these peaks into an output signal track, the following statement can be used:

```
FOR TRACK TInt IN (category='SYDH TFBS',
cell='GM12878' and fname LIKE '%Pk%')
SELECT     TInt.chr, TInt.chrstart, TInt.chrend
FROM       TInt
COMBINED WITH UNION ALL AS AllPeaks;
```

In this statement, a track is selected if it belongs to the ENCODE SYDH transcription factor binding sites (SYDH TFBS) category, contains data from GM12878 cells, and has "Pk" (peak) as part of its track name. The "LIKE" syntax of SQL for string matching with wildcards can be used in specifying track selection conditions. For each selected track, its intervals are represented by the variable TInt, and the union of the intervals from these tracks are stored in the output track "AllPeaks".

As shown in this example, the first form of the FOR TRACK IN () statement combines the results from all the selected tracks by a UNION ALL operation. The second form, on the other hand, allows the query result from each selected track to be stored in a separate output track (with track name <output-track-name> concatenated with the name of the selected track), which can then be post-processed by using other STQL queries. Currently STQL does not support nested FOR TRACK IN () statements.

To demonstrate the use of STQL, in the Supplementary Materials we provide 14 sample queries, including both simple and complex ones.

Signal Track Analytical Research Tool (START)

We developed a system called Signal Track Analytical Research Tool (START) for running STQL queries on multiple machines in parallel. START involves a front-end Web-based user interface and a backend execution system (Figure 6). The purpose of the Web-based user interface is to provide a simple way for users to test out STQL. We have pre-loaded around 10,000 signal tracks from ENCODE, Roadmap Epigenomics, FANTOM5 [9] and other sources into our database for users to integrate these data into their analyses. In additional to the standard file formats supported by START, we also imported some commonly used data in other formats (such as gene annotation in .gtf format) using our custom scripts.

We encourage users who want to use STQL to analyze large amounts of private data to install START locally on their own machines. We provide an installation package at `https://github.com/stql/start/wiki/Install-START-in-your-own-cluster`. START can be run on either a single machine or a cluster of machines. All source code of START can be found at `https://github.com/stql/start`, distributed under Apache License v2.0.

Front-end: Web-based user interface

START provides a Web-based user interface at http://yiplab.cse.cuhk.edu.hk/start/ (Figure 7). It provides a main input box for entering STQL queries. Multiple queries can be entered at the same time, in which case each query should store its results in a temporary track, and the results of the last query will be returned by the system as the final results.

Four features are provided to help users construct their queries. First, a user can use his/her previous queries or queries shared by other users as template to perform new analyses by changing only the parts that differ. Second, signal tracks stored in the backend database are listed in categories. A user can select signal tracks using the built-in searching function based on text matches in all track attributes. The names and data types of the attributes of the intervals in a signal track can be shown by clicking the "track schema" link. Third, in the main input box, STQL keywords are highlighted in different colors to help users spot syntax errors. Finally, an extensive help system is provided on the START Web site with detailed documentations and example queries.

A user can use all the functions described above and submit STQL queries with or without logging in. Users logged in (after a free registration) can additionally store their own executed queries, data files, and query results on START. Each user is given a different database name such that files of different users are completely separated. Data files can be uploaded in a number of standard file formats (.bed, .bedGraph and .wig), and multiple files can be uploaded at the same time in a zip package. All the supported formats have the chr, chrstart and chrend attributes defined. The value attribute is defined in .bedGraph and .wig, while for the .bed format it is left as NULL. The schema of the uploaded data is automatically generated based on this mapping. A user can also share or unshare queries with other users. START ensures that only queries explicitly shared by the owner can be seen by other users, and data files uploaded by a user cannot be accessed by other users.

A user submits a query by entering a name of the query and pressing the "Submit" button. A checker module at the backend is then invoked immediately. If any syntax error or permission problem is detected, the query is rejected and an error message is returned to the user without executing the query. Otherwise, a query job will be created at the backend and the actual processing of it will be carried out when the execution system becomes available.

When a query has been executed, the user can preview the first few rows of the results on START, or download all the results in a file. Users are not required to wait for a query to complete by keeping the browser open, because when a user returns to the START Web site, he/she can find all executed queries from the menu and the result files can be downloaded from the corresponding page linked from the list of executed queries for recently executed queries.

Back-end: Parallel-execution system

In the back-end of START, STQL queries are translated into optimized executable programs that are run on a cluster of machines in parallel. The parallelization is powered by the Hadoop [19] distributed data storage and MapReduce framework for big data processing. Intervals on each chromosome are mapped to the same computing node. The translation of STQL into executable programs is assisted by Hive [20], a warehousing infrastructure built on top of MapReduce. It provides an SQL-like query language called HiveQL, and it translates HiveQL queries into Hadoop programs. We used Hive to execute parts of STQL queries that can be directly translated into HiveQL queries, and handled some signal track-specific constructs (**is closest to each** , **intersectjoin** , **exclusivejoin** , **project on** , **coalesce** and **discretize**) by our own programs. An advantage of Hive is that it can work on raw data files directly, without requiring a long processing time of converting the raw data files into a particular format before the corresponding tracks can be used in the queries. This feature makes it very efficient for users to use their own signal tracks in the queries.

More details are given in the Methods section.

Comparison with other approaches

To evaluate the simplicity of STQL and the correctness and efficiency of START in executing STQL queries, we compared STQL with three other approaches in performing the same analysis tasks.

First, we used the Web-based user interface to submit the 14 example STQL queries to START, and downloaded the resulting output files. For each query, we measured the time required, from submitting the query to getting the final result file. We also used bedtools [5], Galaxy [6] and custom Python scripts to perform the same tasks. We then checked if the output files produced by the different approaches were the same, and compared the time required.

The source code of these three implementations is available at `https://github.com/stql/start/wiki/Website-User-Manual#source-code-for-other-tools`. For some queries, we were unable to find a trivial way to perform exactly the same operations using one or more of these approaches. We note that this does not mean it is impossible to carry out the corresponding analyses using these approaches, but the solutions could be non-trivial. On the other hand, it was fairly easy to write STQL queries to perform the tasks, and the STQL queries involved fewer tokens than both the bedtools and Python scripts for all the 14 tasks (Table 3).

Based on the execution results, START was able to produce identical output files as those produced by the Python scripts for all 14 queries. In some cases, bedtools and Galaxy produced results different from STQL. For example, for SQ5, bedtools could produce the same intervals as STQL but could not derive the required values. In general, STQL was found to be very expressive, and its value derivation capability was particularly flexible.

Table 4 shows the execution time of the different approaches. For START, we used a Hadoop cluster to execute the queries. The cluster contained 22 machines, each with an Intel Core i7-3770 CPU at 3.40GHz, 16GB main memory, and disks with I/O speed of 133.75 MB/s. For bedtools and python scripts, we used a single machine to execute the queries, with an Intel Core i7-3770 CPU at 3.40GHz, 16GB main memory, and disks with I/O speed of 156 MB/s. For Galaxy, we used its online version (`https://usegalaxy.org/`). Since the hardware used for each approach was different, it is not meaningful to use the measured time to argue which approach is more efficient. Instead, the main purpose of this time comparison is threefold. First, it shows that for some of the tasks that STQL could easily handle, we could not find a way to perform the same tasks using bedtools or Galaxy (marked as N/A in Table 4), suggesting that it is more difficult or even impossible to perform these tasks using these tools. Second, in general, START could finish each task within reasonable time even without using algorithms and data structures specially designed for each task as we did with the Python scripts. Third, when the data files were large, the implicit parallel execution of START made it easy to speed up the analysis, without requiring the user to write anything about parallelization in the STQL queries. For example, in SQ1 and SQ2, the data files involved were larger than 1GB, and START was able to finish the task faster than the other approaches due to its parallel computations.

We have also compared STQL with SQL, and found that some operations are much more difficult to perform using SQL than STQL. The details are provided in the Supplementary Materials.

Case study

To test if STQL is easy to learn and to use, we asked one of us (KH-OY), who was trained as a biologist and had received minimal formal training in computer programming (including SQL), to analyze some sequencing data using two different approaches. The data involved were DNA methylation data we produced by MBDCap-seq [18] on 60 pairs of human hepatocellular carcinoma (HCC) tumor and matched non-tumor tissues. The goal was to compute DNA methylation levels at gene promoters, and identify promoters with significant differential methylation between the tumor and non-tumor groups.

The first analysis approach was to implement the analysis pipeline by writing custom Perl scripts. The second approach was to write STQL queries and submit them through the START Web interface, to perform exactly the same analysis.

Specifically, for each protein-coding gene in Gencode [21] v19, the promoter region was defined as the +/-500bp around the transcription start site. The average methylation signal at each promoter was computed separately for the tumor and non-tumor samples. Finally, the full list of genes and their promoter differential methylation fold change values were reported. The Perl scripts and the STQL queries written, as well as the resulting output files, are all available at `https://github.com/stql/start/raw/master/for-download/STQL_HCC_Diff_Methyl_files.zip`.

The STQL queries are found to be simpler than the Perl scripts. For instance, the Perl scripts involve 253 lines of code in total, while the STQL queries involve only 55 lines.

The two approaches led to identical results. Among the top five most hypermethylated promoters, FGF19 is related to HCC tumor promotion [22], FGF4 is related to HCC drug response [23], and HLX is involved in normal liver development [24]. Although the other two genes have yet to link with HCC, their roles in cancer development have been reported. MYEOV deregulation contributes to malignant transformation of different cell types [25], while LRR1 is involved in cell growth control [26]. These results suggest that it is indeed fairly easy for someone without very strong computer science background to learn and use STQL to produce biologically meaningful results.

Discussion

The main purpose of START is to demonstrate the use of STQL. If it is to be used for routine large-scale data analysis, the signal tracks stored in its database should be frequently updated. To achieve that, we are exploring the possibility to hook up START with major genome databases for automatic data updates, or to setup a local copy of START at these sites.

Currently START supports only signal tracks based on the hg19 human reference genome. Conceptually, STQL can also support other versions of the human reference genome, as well as other species. They will be supported in future versions of START.

In the current implementation of the track operators, parallelization is achieved by sending all intervals on one chromosome to one computing node, which is not very efficient due to the very different sizes of the human chromosomes. We are developing new algorithms to provide sub-chromosome level parallelization [27].

There are other emerging distributed computing frameworks. For instance, Spark [28] is a successor of Hadoop that keeps data files in memory such that a user issuing multiple queries on the same data files could enjoy significant speed up. We will explore the possibility of using Spark as the underlying framework to further improve the efficiency of START.

Since in most applications joining of different tracks does not involve pairing of intervals from different chromosomes, by default STQL only considers pairing of intervals from the same chromosome to avoid the unnecessary computational overhead. If it is necessary to pair intervals from different chromosomes, one way is to save chromosome names in a new attribute and replace the chr attribute by a common fake chromosome name before the join operation. After the join, the actual chromosome names can be copied back. We will consider adding an operation that allows across-chromosome comparisons if many applications find it useful.

Implementation

Back-end system of START

The architecture of START is shown in Figure 6. The Web-based front-end has been described in the main text. Here we provide some high-level descriptions of the interface between the front-end and the back-end, and the back-end system.

Interface between front-end and back-end: Metastore

In order for the front-end user interface to obtain information about the stored signal tracks in the database, it has to obtain the information from the back-end. The metastore provides such information and acts as an interface between the front-end and back-end systems. The metastore records three main types of information, namely 1) the schema of each signal track, i.e., the exact names and data types of the attributes of the intervals in each signal track, 2) the physical locations of the corresponding data files in the backend system, which is stored in a Hadoop file system (HDFS), and 3) the organization of the signal tracks into categories, and the attributes of the signal tracks in each category. When any of these three types of information is updated at the back-end, the Web-based user interface always displays the most updated information by retrieving it from the metastore in real time.

Back-end: Translation

At the back-end, we use Hadoop [19] for distributed data storage, which includes a MapReduce framework for big data processing. High-level STQL queries are translated into executable programs (MapReduce jobs) that can be executed by Hadoop. This translation is facilitated by Hive [20], a warehousing infrastructure built on top of MapReduce. It provides an SQL-like query language called HiveQL, and it translates HiveQL queries into Hadoop programs. We extended HiveQL to include syntactic constructs specific to STQL. An advantage of Hive is that it can work on raw data files directly, without requiring a long processing time of converting the raw data files into a particular format before the corresponding tracks can be used in the queries. This feature makes it very efficient for users to use their own signal tracks in the queries.

To execute an STQL query, the first step is to translate it to a sequence of operations. It involves four sub-steps, namely 1) parsing the STQL statement and producing an abstract syntax tree (AST), 2) traversing the AST to create a query block (QB) and record necessary parsing information in the QB, 3) interacting with the metastore to retrieve metadata of the involved signal tracks, and 4) generating a query plan in the form of a directed acyclic graph (DAG) of logical operations based on the QB.

Back-end: Execution

The DAG of logical operations are then converted into executable jobs in Hadoop. Figure 8 shows a simple example illustrating the typical steps in such a MapReduce job. In the Map phase, the TableScan operator fetches one interval from a signal track at a time, and forwards all attributes of the interval to the Filter operator. Upon receiving an interval, the Filter operator judges whether the interval satisfies the predicate in the WHERE clause. If the predicate holds true for the interval, the Filter operator forwards the interval to the Select Operator. The Select operator selects the attributes of the interval necessary for the calculations. It then forwards the results to the ReduceSink operator, which creates a key-value pair for the interval it receives. This finishes the Map phase. Based on the keys, the intervals are sent to different machine nodes for further processing.

In the Reduce phase, the Intersectjoin operator maintains buffers for caching the intervals it receives. When all intervals have been received, it proceeds with the actual computations. Whenever a resulting interval is produced, it forwards the interval to the Select operator, which supplies all attributes that need to be returned in the final outputs.

Back-end: Optimization

Together, the compiler and executor described above are sufficient for turning STQL statements into executable programs. However, the straight-forward way of translating the queries into executable programs could make the programs inefficient. The goal of the optimizer is to find ways to perform the queries more efficiently.

The optimizer makes use of several key ideas. First, it removes interval attributes that are not needed as early as possible, to reduce the amount of data transfer between computing nodes. Second, when a join is performed between two tracks, instead of producing the Cartesian product, the optimizer tries to use more efficient algorithms to reduce both the computation and the amount of intermediate results. For example, by pre-sorting both signal tracks involved, sometimes it is possible to perform a single linear scan of the resulting sorted tracks to produce the join result. Finally, if the **generate bins with length** construct is used, instead of creating the actual bins, the optimizer computes the overlapping bins of each interval, so that projection can be done efficiently without considering the bins that do not overlap any intervals.

STQL grammar rules

In the Results section we have explained the syntax of STQL using high-level terms and examples. Here we present the complete set of grammar rules that define STQL.

STQL_STATEMENT := DDL | DML | QUERY

DDL := CREATE_TRACK | CTAS | DROP_TRACK

CREATE_TRACK := create track TRACKALIAS *LBracket* SCHEMA *RBracket*

CTAS := create track TRACKALIAS as REG_QUERY

SCHEMA := ATTRNAME DATA_TYPE (, ATTRNAME DATA_TYPE)*

DROP_TRACK := drop track TRACKALIAS

DML := LOAD_DATA

LOAD_DATA := load data local inpath *Filepath* (overwrite)? into track TRACKALIAS

DATA_TYPE := string | int | float

QUERY := REG_QUERY | FOR_LOOP

REG_QUERY := SELECT_STAT FROM_STAT (WHERE_STAT)? (GROUPBY_STAT)? (ORDERBY_STAT)?

FOR_LOOP := for track TRACK_VAR in *LBracket TrackProperty RBracket* (REG_QUERY combined with UNION as TRACKALIAS | CTAS)

TRACK_VAR := *Identifier*

FROM_STAT := from FROM_SOURCE

FROM_SOURCE := MULTIPLETRACK

TRACK := RAW_TRACK | TRANSFORM_RES | OVERLAPJOIN_RES | SUBQUERY | UNION_RES

MULTIPLETRACK := TRACK (, TRACK)*

UNION_RES := *LBracket* TRACK UNION TRACK (UNION TRACK)* *RBracket* TRACKALIAS

UNION := union all

RAW_TRACK := (CATEGORY.)?TRACKNAME ((as)? TRACKALIAS)?

CATEGORY := *Identifier*

TRACKALIAS := *Identifier*

TRANSFORM_RES := TRANSFORM_OP | *LBracket* TRANSFORM_OP *RBracket* TRACKALIAS

TRANSFORM_OP := TRANSFORM (with VALUE_DER)?

TRANSFROM := COALESCE TRACK | DISCRETIZE TRACK

COALESCE := coalesce

DISCRETIZE := discretize

OVERLAPJOIN_RES := OVERLAPJOIN_OP | *LBracket* OVERLAPJOIN_OP *RBracket* TRACKALIAS

OVERLAPJOIN_OP := OVERLAPJOIN (with (VALUE_DER_METADATA | VALUE_DER | META_DATA))?

OVERLAPJOIN := INTERSECTJOIN | EXCLUSIVEJOIN | PROJECT

INTERSECTJOIN := TRACK intersectjoin TRACK

EXCLUSIVEJOIN := TRACK exclusivejoin TRACK

PROJECT := project TRACK on (TRACK | CREATE_BINS)

CREATE_BINS := generate bins with length *Integer*

VALUE_DER_METADATA := VALUE_DER, META_DATA | META_DATA, VALUE_DER

VALUE_DER := VD_TYPE using VALUE_MODEL

VD_TYPE := vd_sum | vd_diff | vd_product | vd_quotient | vd_avg | vd_max | vd_min | vd_left | vd_right

META_DATA := metadata

VALUE_MODEL := VM_TYPE model

VM_TYPE := each | all

SELECT_STAT := select ((distinct)? FIELD (, FIELD)* | SELALLEXP)

SELEXP := FIELD (as ATTRNAME)?

SELALLEXP := *

FIELD := ARITH_FUNC | AGG

ARITH_FUNC := (MUL_DIV | *Number*) ((+ | −) (MUL_DIV | *Number*))?

MUL_DIV := (ELEM | *Number*) ((* | /) (ELEM | *Number*))?

ELEM := INTERVAL_ATTR | *LBracket* ARITH_FUNC *RBracket*

INTERVAL_ATTR := ATTRNAME | TRACKNAME.ATTRNAME

TRACKNAME := *Identifier* | TRACKALIAS

ATTRNAME := chr | chrstart | chrend | value | *Identifier*

AGG := AGG_FUNC *LBracket* INTERVAL_ATTR *RBracket* | COUNT_ALL

AGG_FUNC := count | max | min | avg | sum

COUNT_ALL := count *LBracket* SELALLEXP *RBracket*

WHERE_STAT := where (OR_PREDICATE | CLOSEST_PREDICATE)

OR_PREDICATE := AND_PREDICATE (or AND_PREDICATE)?

AND_PREDICATE := NOT_PREDICATE (and NOT_PREDICATE)?

NOT_PREDICATE := PREDICATE | not (PREDICATE | *LBracket* OR_PREDICATE *RBracket*)

PREDICATE := NUMERIC_COMP | LOCATION_COMP | PATTERN_MATCHING

NUMERIC_COMP := (INTERVAL_ATTR | INTERVAL_LENGTH | INTERVAL_DIS | *Number*) COMP_OP (INTERVAL_ATTR | INTERVAL_LENGTH | INTERVAL_DIS | *Number*)

INTERVAL_LENGTH := length *LBracket* (TRACKNAME | CONS_INTERVAL) *RBracket*

INTERVAL_DIS := distance *LBracket* (TRACKNAME | CONS_INTERVAL) , (TRACKNAME | CONS_INTERVAL) *RBracket*

COMP_OP := < | = | ! = | > | <= | >=

LOCATION_COMP := (TRACKNAME | CONS_INTERVAL) LOC_COMP_OP (TRACKNAME | CONS_INTERVAL)

LOC_COMP_OP := overlaps with | precedes | follows | coincides with | is prefix of | is suffix of | is adjacent to | is within | contains | is upstream of | is downstream of

CONS_INTERVAL := *LeftSquareBracket* CHR, CHRSTART, CHREND (, STRAND)? *RightSquareBracket*

CHR := *Identifier*

CHRSTART := *Integer*

CHREND := *Integer*

STRAND := + | −

PATTERN_MATCHING := INTERVAL_ATTR (not)? like *RegularExpression*

CLOSEST_PREDICATE := TRACKNAME is closest to each TRACKNAME

GROUPBY_STAT := group by INTERVAL_ATTR (, INTERVAL_ATTR)*

ORDERBY_STAT := order by INTERVAL_ATTR (, INTERVAL_ATTR)*

SUBQUERY := *LBracket* QUERY *RBracket* TRACKALIAS

Conclusions

In this paper, we have described the Signal Track Query Language (STQL), an SQL-like declarative language that allows users to perform a variety of analysis by specifying only the analysis goals rather than all the computational details. We have demonstrated some typical use of STQL through 14 example queries, which cover both simple and composite analysis tasks. We have used these example queries to show that STQL usually provides a simpler solution than several other popular analysis approaches.

To make it easy to write and execute STQL queries, we have developed the Signal Track Analytical Research Tool (START). The Web-based user interface of START allows simple integrated analysis of private and commonly-used public signal tracks. It also provides the management of stored data and queries. The back-end system of START automatically translates STQL queries into executable programs that are run in parallel on multiple machines, without requiring the analysts to diverge their attention to finding a suitable parallelization strategy.

Together, STQL and START provide a simple and generic way for analyzing a large number of genomic signal tracks.

Availability and Requirements

Project name: Signal Track Analytical Research Tool

Project home page: https://github.com/stql/start

Operating system: Linux (Ubuntu recommended)

Programming language: Java

Other requirements: JDK 6 or higher, Hadoop installation

License: Apache License v2.0

Any restrictions to use by non-academics: license needed

List of abbreviations

AST	Abstract syntax tree
ChIP-seq	chromatin immunoprecipitation followed by high-throughput sequencing
DAG	Directed acyclic graph
ENCODE	Encyclopedia of DNA Elements
eRNA	Enhancer RNA
HDFS	Hadoop File System
HOT	High Occupancy of Trancription-related factors
FANTOM5	Functional Annotation of The Mammalian Genome Phase 5
HCC	Hepatocellular carcinoma
QB	Query block
RNA-seq	RNA (cDNA) sequencing
START	Signal Track Analytical Research Tool
SQL	Structured Query Language
STQL	Signal Track Query Language

Declarations

Ethics approval and consent to participate

Patients who underwent hepatectomy for liver cancer at the Prince of Wales Hospital (Hong Kong, China) were included in this study. All liver cancer patients gave written informed consent on the use of clinical specimens for research purposes. This study was approved by the Joint CUHKVNTEC Clinical Research Ethics Committee (2014.076).

Consent for publication

Not applicable

Availability of data and materials

The raw sequencing data for the HCC analysis have been deposited in the NCBI Sequence Read Archive (accession number: SRP073877). The files involved in comparing the different analysis approaches can be found at `https://github.com/stql/start`.

Competing interests

The authors declare that they have no competing interests.

Funding

This project is partially supported by the HKSAR RGC Theme-based Research Schemes T12-401/13-R (KH-OY and KYY) and T12-402/13N (KYY), Collaborative Research Fund C4017-14G (AS-LC and KYY), and General Research Funds HKU 712712E (BK), PolyU 5210/12E (EL) and PolyU 5204/13E (EL and KYY), and Research Committee of CUHK grants 3132964 and 3132821 (EL and KYY). The funding bodies were not involved in the design of the study or collection, analysis, or interpretation of data or in writing the manuscript.

Authors' contributions

BK, EL and KYY conceived the study. XZ, BK and KYY designed STQL. XZ, QZ and EDH developed and tested START. KH-OY, THH, AS-LC and KYY produced and analyzed the HCC data. XZ, QZ, KH-OY and TL compared STQL with other languages and systems. XZ, BK and KYY wrote the manuscript. All authors read and approved the final manuscript.

Acknowledgements

The authors would like to thank Dr. Sau-Dan Lee for helpful discussions.

Author details

[1]Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong, Hong Kong. [2]School of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. [3]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong. [4]Department of Anatomical and Cellular Pathology, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong. [5]Department of Molecular Medicine, University of Texas Health Science Center at San Antonio, San Antonio, Texas, United States of America. [6]School of Biomedical Sciences, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong. [7]Hong Kong Bioinformatics Centre, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong. [8]CUHK-BGI Innovation Institute of Trans-omics, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong. [9]Hong Kong Institute of Diabetes and Obesity, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong.

References

1. The Cancer Genome Atlas Research Network, Weinstein, J.N., Collisson, E.A., Mills, G.B., Shaw, K.R.M., Ozenberger, B.A., Ellrott, K., Shmulevich, I., Sander, C., Stuart, J.M.: The cancer genome atlas pan-cancer analysis project. Nature Genetics **45**(10), 1113–1120 (2013)
2. The ENCODE Project Consortium: An integrated encyclopedia of DNA elements in the human genome. Nature **489**(7414), 57–74 (2012)
3. Roadmap Epigenomics Consortium, Kundaje, A., Meuleman, W., Ernst, J., Bilenky, M., Yen, A., Heravi-Moussavi, A., Kheradpour, P., Zhang, Z., Wang, J., Ziller, M.J., Amin, V., Whitaker, J.W., Schultz, M.D., Ward, L.D., Sarkar, A., Quon, G., Sandstrom, R.S., Eaton, M.L., Wu, Y.-C., Pfenning, A.R., Wang, X., Claussnitzer, M., Liu, Y., Coarfa, C., Harris, R.A., Shoresh, N., Epstein, C.B., Gjoneska, E., Leung, D., Xie, W., Hawkins, R.D., Lister, R., Hong, C., Gascard, P., Mungall, A.J., Moore, R., Chuah, E., Tam, A., Canfield, T.K., Hansen, R.S., Kaul, R., Sabo, P.J., Bansal, M.S., Carles, A., Dixon, J.R., Farh, K.-H., Feizi, S., Karlic, R., Kim, A.-R., Kulkarni, A., Li, D., Lowdon, R., Elliott, G., Mercer, T.R., Neph, S.J., Onuchic, V., Polak, P., Rajagopal, N., Ray, P., Sallari, R.C., Siebenthall, K.T., Sinnott-Armstrong, N.A., Stevens, M., Thurman, R.E., Wu, J., Zhang, B., Zhou, X., Beaudet, A.E., Boyer, L.A., Jager, P.L.D., Farnham, P.J., Fisher, S.J., Haussler, D., Jones, S.J.M., Li, W., Marra, M.A., McManus, M.T., Sunyaev, S., Thomson, J.A., Tlsty, T.D., Tsai, L.-H., Wang, W., Waterland, R.A., Zhang, M.Q., Chadwick, L.H., Bernstein, B.E., Costello, J.F., Ecker, J.R., Hirst, M., Meissner, A., Milosavljevic, A., Ren, B., Stamatoyannopoulos, J.A., Wang, T., Kellis, M.: Integrative analysis of 111 reference human epigenomes. Nature **518**(7539), 317–330 (2015)
4. Masseroli, M., Pinoli, P., Venco, F., Kaitoua, A., Jalili, V., Palluzzi, F., Muller, H., Ceri, S.: GenoMetric query language: A novel approach to large-scale genomic data management. Bioinformatics **31**, 1881–1888 (2015)
5. Quinlan, A.R., Hall, I.M.: BEDTools: A flexible suite of utilities for comparing genomic features. Bioinformatics **26**, 841–842 (2010)
6. Goecks, J., Nekrutenko, A., Taylor, J., The Galaxy Team: Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biology **11**, 86 (2010)
7. Ernst, J., Kellis, M.: ChromHMM: Automating chromatin-state discovery and characterization. Nature Methods **9**(3), 215–216 (2012)
8. Sherry, S.T., Ward, M.-H., Kholodov, M., Baker, J., Phan, L., Smigielski, E.M., Sirotkin, K.: dbSNP: The NCBI database of genetic variation. Nucleic Acids Research **29**(1), 308–311 (2001)
9. Andersson, R., Gebhard, C., Miguel-Escalada, I., Hoof, I., Bornholdt, J., Boyd, M., Chen, Y., Zhao, X., Schmidl, C., Suzuki, T., Ntini, E., Arner, E., Valen, E., Li, K., Schwarzfischer, L., Glatz, D., Raithel, J., Lilje, B., Rapin, N., Bagger, F.O., Jorgensen, M., Andersen, P.R., Bertin, N., Rackham, O., Burroughs, A.M., Baillie, J.K., Ishizu, Y., Shimizu, Y., Furuhata, E., Maeda, S., Negishi, Y., Mungall, C.J., Meehan, T.F., Lassmann, T., Itoh, M., Kawaji, H., Kondo, N., Kawai, J., Lennartsson, A., Daub, C.O., Heutink, P., Hume, D.A., Jensen, T.H., Suzuki, H., Hayashizaki, Y., Muller, F., Consortium, T.F., Forrest, A.R.R., Carninci, P., Rehli, M., Sandelin, A.: An atlas of active enhancers across human cell types and tissues. Nature **507**(7493), 455–461 (2014)
10. Tyner, C., Barber, G.P., Casper, J., Clawson, H., Diekhans, M., Eisenhart, C., Fischer, C.M., Gibson, D., Gonzalez, J.N., Guruvadoo, L., Haeussler, M., Heitner, S., Hinrichs, A.S., Karolchik, D., Lee, B.T., Lee, C.M., Nejad, P., Raney, B.J., Rosenbloom, K.R., Speir, M.L., Villarreal, C., Vivian, J., Zweig, A.S., Haussler, D., Kuhn, R.M., Kent, W.J.: The UCSC genome browser database: 2017 update. Nucleic Acids Research **45**, 626–634 (2017)
11. Yip, K.Y., Cheng, C., Bhardwaj, N., Brown, J.B., Leng, J., Kundaje, A., Rozowsky, J., Birney, E., Bickel, P., Snyder, M., Gerstein, M.: Classification of human genomic regions based on experimentally-determined binding sites of more than 100 transcription-related factors. Genome Biology **13**, 48 (2012)

12. Kozanitis, C., Heiberg, A., Varghese, G., Bafna, V.: Using genome query language to uncover genetic variation. Bioinformatics **30**, 1–8 (2014)
13. Nordberg, H., Bhatia, K., Wang, K., Wang, Z.: BioPig: A hadoop-based analytic toolkit for large-scale sequence data. Bioinformatics **29**(23), 3014–3019 (2013)
14. Ovaska, K., Lyly, L., Sahu, B., Janne, O.A., Hautaniemi, S.: Genomic region operation kit for flexible processing of deep sequencing data. IEEE Transactions on Computational Biology and Bioinformatics **10**, 200–206 (2013)
15. Schumacher, A., Pireddu, L., Niemenmaa, M., Kallio, A., Korpelainen, E., Zanetti, G., Heljanko, K.: SeqPig: Simple and scalable scripting for large sequencing data sets in hadoop. Bioinformatics **30**, 119–120 (2014)
16. Wiewiórka, M.S., Messina, A., Pacholewska, A., Maffioletti, S., Gawrysiak, P., Okoniewski, M.J.: SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. Bioinformatics **30**, 2652–2653 (2014)
17. Whyte, W.A., Orlando, D.A., Hnisz, D., Abraham, B.J., Lin, C.Y., Kagey, M.H., Rahl, P.B., Lee, T.I., Young, R.A.: Master transcription factors and mediator establish super-enhancers at key cell identity genes. Cell **153**, 307–319 (2013)
18. Hsu, Y.-T., Gu, F., Huang, Y.-W., Liu, J., Ruan, J., Huang, R.-L., Wang, C.-M., Chen, C.-L., Jadhav, R.R., Lai, H.-C., Mutch, D.G., Goodfellow, P.J., Thompson, I.M., Kirma, N.B., Huang, T.H.-M.: Promoter hypomethylation of EpCAM-regulated bone morphogenetic protein gene family in recurrent endometrial cancer. Clinical Cancer Research **19**, 6272–6285 (2013)
19. White, T.: Hadoop: The Definitive Guide, 4th edn. O'Reilly Media, Inc., Sebastopol (2015)
20. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wycko, P., Murthy, R.: Hive: A warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment **2**, 1626–1629 (2009)
21. Harrow, J., Frankish, A., Gonzalez, J.M., Tapanari, E., Diekhans, M., Kokocinski, F., Aken, B.L., Barrell, D., Zadissa, A., Searle, S., Barnes, I., Bignell, A., Boychenko, V., Hunt, T., Kay, M., Mukherjee, G., Rajan, J., Despacio-Reyes, G., Saunders, G., Steward, C., Harte, R., Lin, M., Howald, C., Tanzer, A., Derrien, T., Chrast, J., Walters, N., Balasubramanian, S., Pei, B., Tress, M., Rodriguez, J.M., Ezkurdia, I., van Baren, J., Brent, M., Haussler, D., Kellis, M., Valencia, A., Reymond, A., Gerstein, M., Guigo, R., Hubbard, T.J.: GENCODE: The reference human genome annotation for the ENCODE project. Genome Research **22**, 1760–1774 (2012)
22. Sawey, E.T., Chanrion, M., Cai, C., Wu, G., Zhang, J., Zender, L., Zhao, A., Busuttil, R.W., Yee, H., Stein, L., French, D.M., Finn, R.S., Lowe, S.W., Powers, S.: Identification of a therapeutic strategy targeting amplified FGF19 in liver cancer by oncogenomic screening. Cancer Cell **19**, 347–358 (2011)
23. Arao, T., Ueshima, K., Matsumoto, K., Nagai, T., Kimura, H., Hagiwara, S., Sakurai, T., Haji, S., Kanazawa, A., Hidaka, H., Iso, Y., Kubota, K., Shimada, M., Utsunomiya, T., Hirooka, M., Hiasa, Y., Toyoki, Y., Hakamada, K., Yasui, K., Kumada, T., Toyoda, H., Sato, S., Hisai, H., Kuzuya, T., Tsuchiya, K., Izumi, N., Arii, S., Nishio, K., Kudo, M.: FGF3/FGF4 amplification and multiple lung metastases in responders to sorafenib in hepatocellular carcinoma. Hepatology **57**, 1407–1415 (2013)
24. Hentsch, B., Lyons, I., Li, R., Hartley, L., Lints, T.J., Adams, J.M., Harvey, R.P.: Hlx homeo box gene is essential for an inductive tissue interaction that drives expansion of embryonic liver and gut. Genes and Development **10**, 70–79 (1996)
25. Janssen, J.W.G., Vaandrager, J.-W., Heuser, T., Jauch, A., Kluin, P.M., Geelen, E., Bergsagel, P.L., Kuehl, W.M., Drexler, H.G., Otsuki, T., Bartram, C.R., Schuuring, E.: Concurrent activation of a novel putative transforming gene, myeov, and cyclin D1 in a subset of multiple myeloma cell lines with t(11;14)(q13;q32). Blood **95**, 2691–2698 (2000)
26. Wang, W., Huang, J., Wang, X., Yuan, J., Li, X., Feng, L., Park, J.-I., Chen, J.: PTPN14 is required for the density-dependent control of YAP1. Genes and Development **26**, 1959–1971 (2012)
27. Zhang, Q., He, A., Liu, C., Lo, E.: Closest interval join using MapReduce. In: Proceedings of the 2016 IEEE International Conference on Data Science and Advanced Analytics, pp. 302–311 (2016)
28. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Cnference on Networked Systems Design and Implementation (2012)

Figures

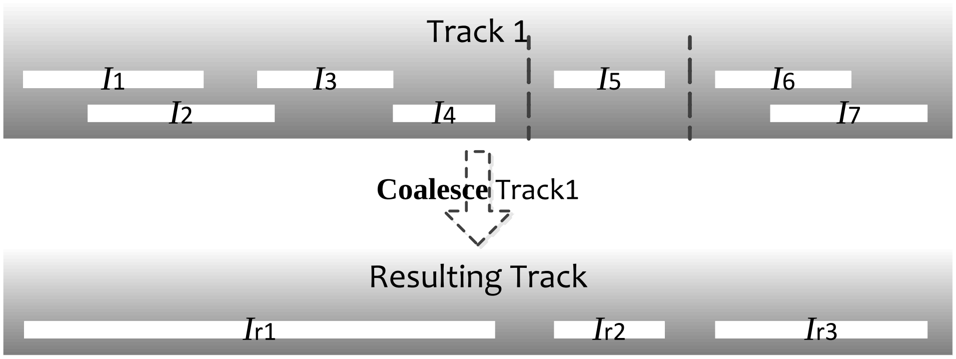


Figure 1 An example that illustrates the **coalesce** operator. $I_1..I_7$ are intervals in the input track that are allowed to overlap with or be adjacent to each other, while $I_{r1}..I_{r3}$ are the non-overlapping, non-adjacent intervals in the resulting track after the coalesce operation. I_{r1} is formed by merging I_1, I_2, I_3 and I_4, which occupy a contiguous block of genomic locations. I_{r2} is formed by I_5 alone, which does not overlap with or is adjacent to any other input intervals. I_{r3} is formed by merging I_6 and I_7.

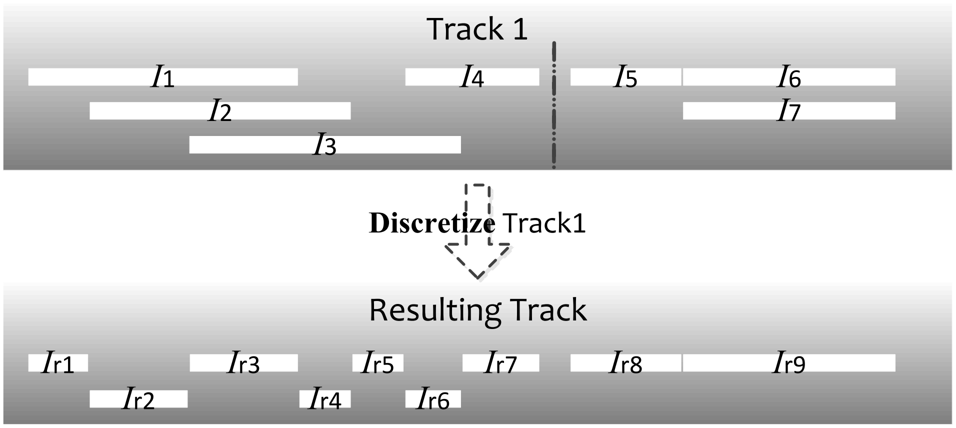


Figure 2 An example that illustrates the **discretize** operator. $I_1..I_7$ are intervals in the input track that are allowed to overlap with each other, while $I_{r1}..I_{r9}$ are non-overlapping intervals in the resulting track after the discretization operation. Each resulting interval is defined by the boundary positions of some input intervals. For example, I_{r1}'s starting position is the same as I_1's starting position, and its ending position is equal to I_2's starting position minus one. It should be noted that the output intervals I_{r8} and I_{r9} are adjacent to each other since they were produced from input intervals that were also adjacent (I_5 and I_6/I_7).

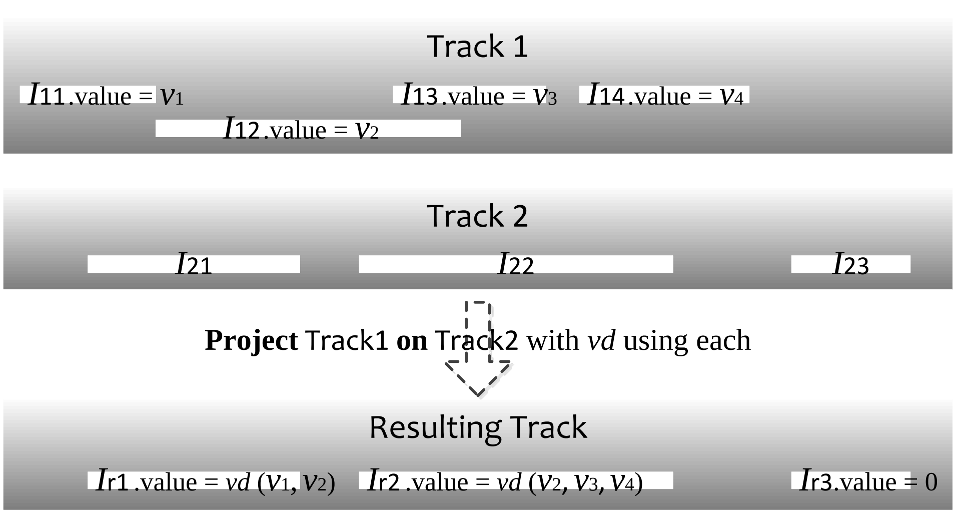


Figure 3 An example that illustrates the **project on** operator. $I_{11}..I_{14}$ are intervals in input track 1, $I_{21}..I_{23}$ are intervals in input track 2, while $I_{r1}..I_{r3}$ are intervals in the resulting track after the projection. The locations of the intervals in the resulting track are directly from the intervals in input track 2. The value of I_{r1} is determined by the values of I_{11} and I_{12} since they are the ones that overlap with I_{21}. The exact way of computing the value depends on the mathematical operator and the value model (which we use vd(v_1, v_2) here to mean the computation based on the values from intervals I_{11} and I_{12}). Similarly, the value of I_{r2} is determined by the values of I_{12}, I_{13} and I_{14} since they are the intervals that overlap with I_{22}. Since I_{23} does not overlap with any intervals in track 1, it does not receive any value from track 1 but is instead given the default value of 0.

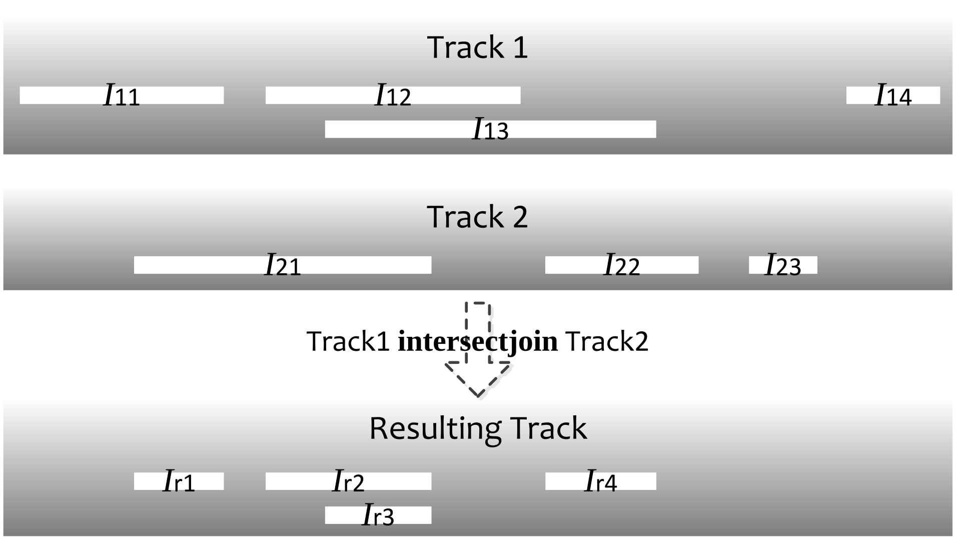


Figure 4 An example that illustrates the **intersectjoin** operator. $I_{11}..I_{14}$ are intervals in input track 1, $I_{21}..I_{23}$ are intervals in input track 2, while Ir1..I_{r4} are intervals in the resulting track after the intersect-join. I_{11} and I_{21} each produces only one resulting interval (I_{r1} and I_{r2} respectively) because they only overlap with I_{21}. I_{13} produces two resulting intervals (I_{r3} and I_{r4}) because it overlaps with both I_{21} and I_{22}. I_{14} does not produce any resulting interval because it does not overlap with any intervals in track 2.

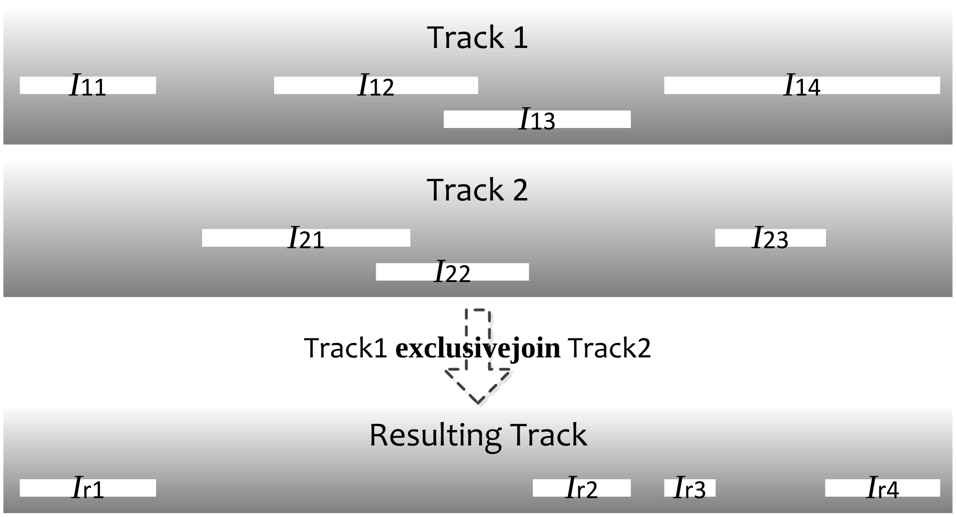


Figure 5 An example that illustrates the **exclusivejoin** operator. $I_{11}..I_{14}$ are intervals in input track 1, $I_{21}..I_{23}$ are intervals in input track 2, while $I_{r1}..I_{r4}$ are intervals in the resulting track after the exclusive-join. The whole interval of I_{11} remains to become I_{r1} in the resulting track, because it does not overlap with any interval in track 2. In contrast, the whole interval of I_{12} is not included in the resulting track, because it is completely covered by I_{21} and I_{22}. For I_{13}, the part of it not covered by I_{22} becomes interval I_{r2} in the resulting track. Finally, I_{14} is being cut by I_{23} into two intervals I_{r3} and I_{r4} in the resulting track.

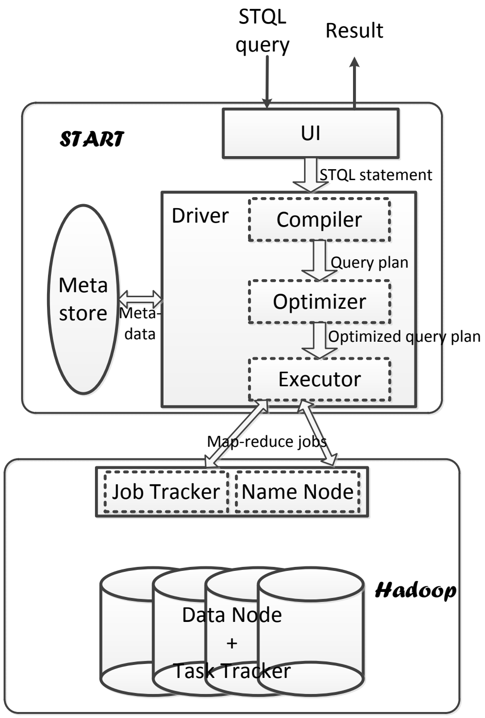
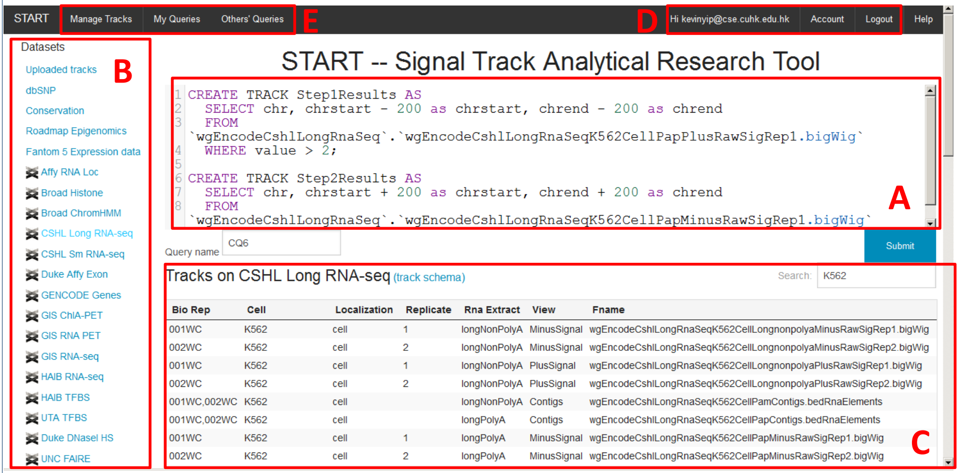


Figure 6 The overall architecture of START. The Web-based user interface helps users select signal tracks, construct STQL queries, submit queries, and retrieve execution results. It also provides various additional functionality, such as user management, storage for queries, data files and result files, and sharing of queries with other users. The metastore provides information about the stored signal tracks in the backend database. When a query is sent to the backend system, it is handled by a driver that consists of three main components. First, a compiler checks for potential syntactic and permission errors, and produces a parse tree of the query if no errors are found. Second, an optimizer analyzes the parse tree and determines an execution plan optimized for efficiency. Third, an executor calls the underlying system to execute the query. The underlying system is based on the Hadoop framework, which distributes the data files needed and performs the actual computations on multiple machines in parallel. When a job is finished, the results are stored and the user is notified to preview or download them using the user inferface.



START -- Signal Track Analytical Research Tool

```
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`
```

Query name: CQ6

Tracks on CSHL Long RNA-seq (track schema)

Search: K562

Bio Rep	Cell	Localization	Replicate	Rna Extract	View	Fname
001WC	K562	cell	1	longNonPolyA	MinusSignal	wgEncodeCshlLongRnaSeqK562CellLongnonpolyaMinusRawSigRep1.bigWig
002WC	K562	cell	2	longNonPolyA	MinusSignal	wgEncodeCshlLongRnaSeqK562CellLongnonpolyaMinusRawSigRep2.bigWig
001WC	K562	cell	1	longNonPolyA	PlusSignal	wgEncodeCshlLongRnaSeqK562CellLongnonpolyaPlusRawSigRep1.bigWig
002WC	K562	cell	2	longNonPolyA	PlusSignal	wgEncodeCshlLongRnaSeqK562CellLongnonpolyaPlusRawSigRep2.bigWig
001WC,002WC	K562	cell		longNonPolyA	Contigs	wgEncodeCshlLongRnaSeqK562CellPamContigs.bedRnaElements
001WC,002WC	K562	cell		longPolyA	Contigs	wgEncodeCshlLongRnaSeqK562CellPapContigs.bedRnaElements
001WC	K562	cell	1	longPolyA	MinusSignal	wgEncodeCshlLongRnaSeqK562CellPapMinusRawSigRep1.bigWig
002WC	K562	cell	2	longPolyA	MinusSignal	wgEncodeCshlLongRnaSeqK562CellPapMinusRawSigRep2.bigWig

Figure 7 The user interface of START. (A) The main text box for entering STQL queries. (B) A list of signal track categories of the tracks stored in the backend database. (C) The list of signal tracks in the selected category. (D) Menu items related to user accounts. (E) Menu items for managing and sharing stored queries and files.

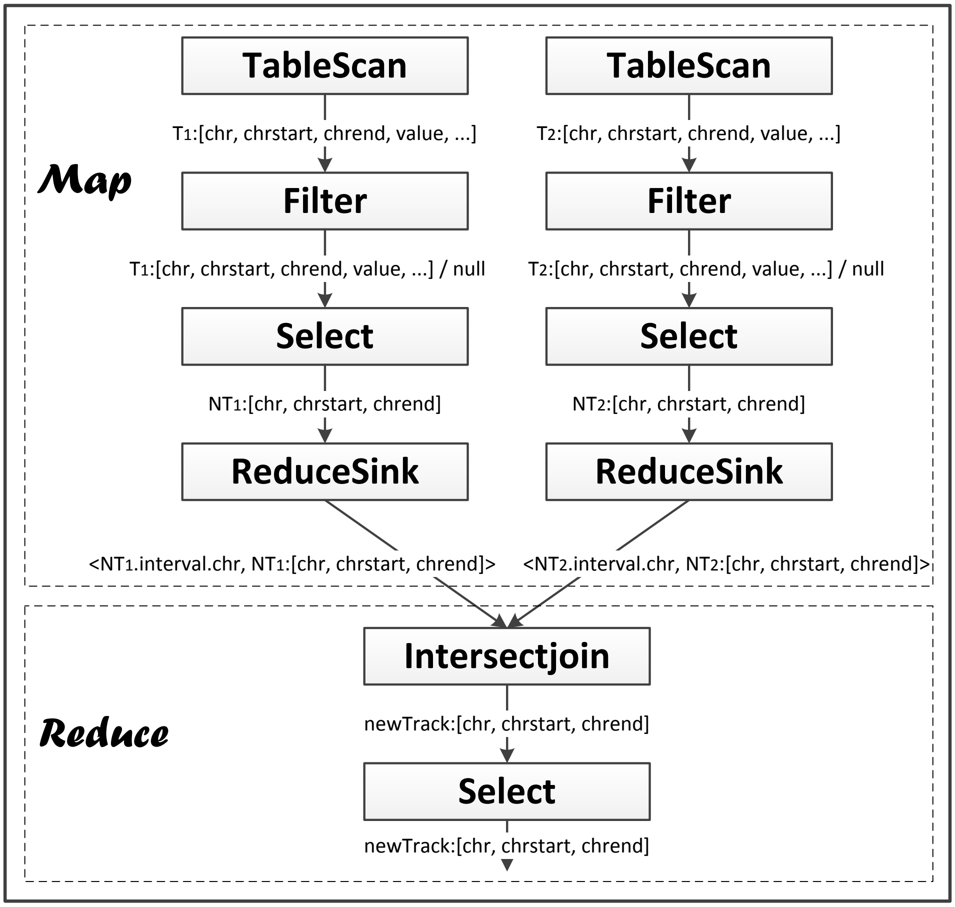

TableScan
TableScan
T1:[chr, chrstart, chrend, value, ...]
T2:[chr, chrstart, chrend, value, ...]
Map
Filter
Filter
T1:[chr, chrstart, chrend, value, ...] / null
T2:[chr, chrstart, chrend, value, ...] / null
Select
Select
NT1:[chr, chrstart, chrend]
NT2:[chr, chrstart, chrend]
ReduceSink
ReduceSink
<NT1.interval.chr, NT1:[chr, chrstart, chrend]>
<NT2.interval.chr, NT2:[chr, chrstart, chrend]>
Intersectjoin
Reduce
newTrack:[chr, chrstart, chrend]
Select
newTrack:[chr, chrstart, chrend]

Figure 8 A typical MapReduce job created by the executor from an STQL query.

Relation	Definition	Example use
I_1 **coincides with** I_2	I_1.chr = I_2.chr and I_1.chrstart = I_2.chrstart and I_1.chrend = I_2.chrend	From the tracks of two replicated experiments where each interval stores the average signal of a genomic bin, find out the bins with values in both experiments
I_1 **overlaps with** I_2	I_1.chr = I_2.chr and I_1.chrstart $\leq$ I_2.chrend and I_1.chrend $\geq$ I_2.chrstart	From a track of intervals that represent a type of signal, find out those that overlap the promoters defined as intervals in another track
I_1 **contains** I_2	I_1.chr = I_2.chr and I_1.chrstart $\leq$ I_2.chrstart and I_1.chrend $\geq$ I_2.chrend	From a track of intervals that represent transcription factor binding sites, find out those that contain single nucleotide variants defined as intervals in another track
I_1 **is within** I_2	I_1.chr = I_2.chr and I_1.chrstart $\geq$ I_2.chrstart and I_1.chrend $\leq$ I_2.chrend	From a track of intervals that represent genes, find out those that are contained by haplotype blocks defined as intervals in another track
I_1 **is adjacent to** I_2	I_1.chr = I_2.chr and (I_1.chrend + 1 = I_2.chrstart or I_1.chrstart − 1 = I_2.chrend)	From a track of intervals that represent different sequence elements, find out the flanking exons of an intron
I_1 **is prefix of** I_2	I_1.chr = I_2.chr and I_1.chrstart = I_2.chrstart and I_1.chrend $\leq$ I_2.chrend	From a track of intervals that represent genes and their sub-elements, find out the first exon of each gene on the positive strand
I_1 **is suffix of** I_2	I_1.chr = I_2.chr and I_1.chrstart $\geq$ I_2.chrstart and I_1.chrend = I_2.chrend	From a track of intervals that represent genes and their sub-elements, find out the first exon of each gene on the negative strand
I_1 **precedes** I_2	I_1.chr = I_2.chr and I_1.chrend $<$ I_2.chrstart	Ordering any type of intervals on the same chromosome
I_1 **follows** I_2	I_1.chr = I_2.chr and I_1.chrstart $>$ I_2.chrend	Ordering any type of intervals on the same chromosome
I_1 **is upstream of** I_2	I_1.chr = I_2.chr and ((I_2.strand = ‘+’ and I_1.strand = ‘+’ and I_1 **precedes** I_2) or (I_2.strand = ‘+’ and I_1.strand = ‘.’ and I_1 **precedes** I_2) or (I_2.strand = ‘−’ and I_1.strand = ‘−’ and I_1 **follows** I_2) or (I_2.strand = ‘−’ and I_1.strand = ‘.’ and I_1 **follows** I_2))	From a track of intervals that represent transcripts, define their promoter regions
I_1 **is downstream of** I_2	I_1.chr = I_2.chr and ((I_2.strand = ‘+’ and I_1.strand = ‘+’ and I_1 **follows** I_2) or (I_2.strand = ‘+’ and I_1.strand = ‘.’ and I_1 **follows** I_2) or (I_2.strand = ‘−’ and I_1.strand = ‘−’ and I_1 **precedes** I_2) or (I_2.strand = ‘−’ and I_1.strand = ‘.’ and I_1 **precedes** I_2))	From a track of intervals that represent sequence motifs, find out their downstream sequence elements defined as intervals in another track

Table 1 Relations defined in STQL for comparing different intervals.

Tables

STQL operation	**coalesce** , **discretize** or **project on**	**intersectjoin**	**exclusivejoin**
Values involved	$v_1 ... v_n$	v_1, v_2	v_1
vd_sum	$\sum_{i=1}^{n} v_i$	$v_1 + v_2$	N/A
vd_avg	$\frac{\sum_{i=1}^{n} v_i}{n}$	$(v_1 + v_2)/2$	N/A
vd_diff	N/A	$v_1 - v_2$	N/A
vd_product	$\prod_{i=1}^{n} v_i$	$v_1 \times v_2$	N/A
vd_quotient	N/A	$v_1 \div v_2$	N/A
vd_max	$\max_{i=1}^{n} v_i$	$\max(v_1, v_2)$	N/A
vd_min	$\min_{i=1}^{n} v_i$	$\min(v_1, v_2)$	N/A
vd_left	N/A	v_1	v_1
vd_right	N/A	v_2	N/A

Table 2 The full list of mathematical operations defined for STQL operations that create intervals. Starting from the third row, the first column shows the names of these mathematical operations that can be used in the <vd> placeholders in statements involving **coalesce** , **discretize** , **project on** , **intersectjoin** and **exclusivejoin** . These mathematical operations are used in Step 2 of value derivation. The second row defines the values involved in the operations. In the case of **intersectjoin** , exactly two values are involved, namely v_1 from the first track and v_2 from the second track. In the case of **exclusivejoin** , exactly one value is involved, namely v_1 from the first track. In the case of **coalesce** , **discretize** and **project on** , all values come from the same track and there can be one or more values involved. N/A indicates mathematical operators that cannot be used with the STQL operations.

Query	START	Bedtools	Python
SQ1	21	63	158
SQ2	30	71	220
SQ3	6	26	202
SQ4	34	61	336
SQ5	23	28	162
SQ6	12	24	146
SQ7	13	25	117
SQ8	14	25	91
CQ1	38	N/A	288
CQ2	53	N/A	460
CQ3	102	N/A	471
CQ4	105	164	500
CQ5	266	N/A	462
CQ6	50	83	202

Table 3 Number of tokens involved in the code of the different approaches on the 14 example queries. N/A indicates cases in which we were unable to find a trivial way to perform the analysis using the approach.

Query	Number of input tracks	Number of input intervals	Number of output intervals	START	Bedtools	Python	Galaxy
SQ1	1	57,059,743	23,857,046	207	407	1171	N/A
SQ2	2	11,517,945	33,312	50	135	184	N/A
SQ3	2	51,417	14,026	39	0.04	0.3	23
SQ4	2	11,517,945	1,054,854	47	21	42	408
SQ5	1	8,898,501	450,380	52	7	125	270
SQ6	2	18,839	5,702	46	0.04	21	N/A
SQ7	1	2,619,444	36,366	31	6	5	44
SQ8	1	2,619,444	1	33	2	3	30
CQ1	52	1,514,863	2,590,502	86	N/A	36	N/A
CQ2	3	2,938,174	29,225	300	N/A	7	N/A
CQ3	53	4,134,307	76,041	1340	N/A	84	N/A
CQ4	100	32,297,907	68,031	1680	262	420	N/A
CQ5	5	257,369,824	264	360	N/A	5289	N/A
CQ6	2	65,412,859	4,006,220	119	207	483	N/A

Table 4 Execution time of the different approaches on the 14 sample queries in seconds. N/A indicates cases in which we were unable to find a trivial way to perform the analysis using the approach.

Supplementary Materials

Sample queries

To illustrate the use of STQL in performing practical analyses, here we describe a number of complete sample queries, including both simple ones involving single statements and composite ones involving multiple statements. Each of these queries can be tested by pasting the whole statement(s) into the query box on the main page of START and submitting the query from there.

Simple queries

SQ1 Analysis task: To compute the average H3K4me1 signal at each 100bp bin across the whole genome, for identifying potential transcriptional enhancers.

Query template:

```
SELECT    *
FROM      (project T on generate bins with length 100
with vd_sum using EACH MODEL) NtInt
WHERE     NtInt.value > 0;
```

Example of real data:

- T: ‘wgEncodeBroadHistone‘. ‘wgEncodeBroadHistoneGm12878H3k04me1StdSigV2.bigWig‘ (An ENCODE ChIP-seq data file of H3K4me1 signals in the GM12878 cell line produced by the Broad Institute)

Explanations: This is a simple demonstration of the second form of the **project on** statement. In the bigWig file we use, the intervals are all non-overlapping. In this case, using **vd_sum**, **vd_avg**, **vd_product**, **vd_max** and **vd_min** would all give the same results.

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

Query template:

```
SELECT    *
FROM      (project T1 on (
                SELECT    DISTINCT chr, chrstart, chrend
                FROM      T2
                WHERE     feature = ’gene’) NtInt1
                with vd_avg using EACH MODEL) NtInt2
WHERE     NtInt2.value > 0;
```

Example of real data:

- T_1: ‘wgEncodeCshlLongRnaSeq‘. ‘wgEncodeCshlLongRnaSeqGm12878CellTotalPlusRawSigRep1.bigWig‘ (An ENCODE RNA-seq data file of total long RNA in the GM12878 cell line produced by the Cold Spring Harbor Laboratory)
- T_2: ‘wgEncodeGencode‘.‘gencode.v19.annotation.gtf‘ (Gencode [21] version 19 annotation file)

Explanations: In this query, a nested query is first used to select the sequence elements in the gene annotation file that correspond to genes. “feature” is a non-default attribute defined for the gene annotation track. A projection is then performed to compute the average RNA-seq signal of each gene, and the genes with non-zero expression are returned.

SQ3 Analysis task: To find the genomic regions covered by signal peaks of both H3K4me1 and H3K27ac, which are potential active enhancers in a particular context (the HCT116 human cell line in this case).

Query template:

```
SELECT    *
FROM      T1 intersectjoin T2;
```

Example of real data:

- T_1: ‘wgEncodeSydhHistone‘. ‘wgEncodeSydhHistoneHct116H3k04me1UcdPk.narrowPeak‘ (An ENCODE ChIP-seq data file of H3K4me1 signal peaks in the HCT116 cell line produced by the Stanford/Yale/Davis/Harvard sub-group)
- T_2: ‘wgEncodeSydhHistone‘. ‘wgEncodeSydhHistoneHct116H3k27acUcdPk.narrowPeak‘ (An ENCODE ChIP-seq data file of H3K27ac signal peaks in the HCT116 cell line produced by the Stanford/Yale/Davis/Harvard sub-group)

Explanations: This query demonstrates the use of the **intersectjoin** construct in finding common regions in different signal tracks.

SQ4 Analysis task: To identify 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.

Query template:

```
SELECT    *
FROM      T1 exclusivejoin (
          SELECT    chr, chrstart, chrend
          FROM      T2
          WHERE     feature = ’gene’ AND
                    attributes LIKE ’%gene_type “protein_coding”%’
                    AND
                    (attributes LIKE ’%level 1%’ OR
                    attributes LIKE ’%level 2%’)
          ) NtInt;
```

Example of real data:

- T_1: ‘wgEncodeCshlLongRnaSeq‘. ‘wgEncodeCshlLongRnaSeqGm12878CellTotalPlusRawSigRep1.bigWig‘ (An ENCODE RNA-seq data file of total long RNA in the GM12878 cell line produced by the Cold Spring Harbor Laboratory)
- T_2: ‘wgEncodeGencode‘.‘gencode.v19.annotation.gtf‘ (Gencode version 19 annotation file)

Explanations: This query demonstrates the use of the **exclusivejoin** construct in excluding regions. A nested query is used to select out only level-1 and level-2 protein coding genes from an annotation file, based on the non-default attribute “attributes” defined for the gene annotation track. These regions are then excluded from the expressed regions with RNA-seq signals. One could also easily modify the query to exclude also small flanking regions from each gene, by selecting for example “T_2.chrstart-1000” and “T_2.chrend

+1000” in the nested query, or by considering only regions with RNA-seq signals higher than a certain threshold as expressed, by pre-filtering T_1 using the WHERE clause.

SQ5 Analysis task: To identify contiguous genomic regions with significant expression, which could correspond to transcribed exons.

Query template:

```
SELECT   *
FROM     coalesce (
         SELECT    chr, chrstart, chrend, value
         FROM      T
         WHERE     value > 2) NtInt
         with vd_avg using EACH MODEL;
```

Example of real data:

- T: ‘wgEncodeCshlLongRnaSeq‘. ‘wgEncodeCshlLongRnaSeqGm12878CellTotalPlusRawSigRep1.bigWig‘ (An ENCODE RNA-seq data file of total long RNA in the GM12878 cell line produced by the Cold Spring Harbor Laboratory)

Explanations: This query demonstrates the use of the **coalesce** construct in joining overlapping and adjacent regions. A nested query is used to select genomic locations with an expression level larger than 2 (say in RPKM or other units). These regions are then joined together into larger contiguous regions by using **coalesce** .

SQ6 Analysis task: To identify 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.

Query template:

```
SELECT   *
FROM     T1 TInt1, T2 TInt2
WHERE    TInt1 overlaps with TInt2;
```

Example of real data:

- T_1: ‘wgEncodeSydhTfbs‘. ‘wgEncodeSydhTfbsHelas3CfosStdPk.narrowPeak‘ (An ENCODE ChIP-seq data file of Cfos binding signal peaks in the HeLa-S3 cell line produced by the Stanford/Yale/Davis/Harvard sub-group)
- T_2: ‘wgEncodeSydhTfbs‘. ‘wgEncodeSydhTfbsHelas3CjunStdPk.narrowPeak‘ (An ENCODE ChIP-seq data file of Cjun binding signal peaks in the HeLa-S3 cell line produced by the Stanford/Yale/Davis/Harvard sub-group)

Explanations: This query demonstrates the use of the **overlaps with** relation in the WHERE clause. The query returns Cfos binding peaks that overlap Cjun binding peaks. These two factors are both members of the AP-1 complex and are expected to have overlapping binding peaks. This query is different from taking an **intersectjoin** between the two tracks (which is another possible way to study co-binding events), because **intersectjoin** only returns the overlapping parts of the intervals but not whole Cfos binding peaks.

SQ7 Analysis task: To identify all annotated genes longer than a given length.

Query template:

```
SELECT    *
FROM      T TInt
WHERE     feature = 'gene' AND length(TInt) > 1000;
```

Example of real data:

- T: ‘wgEncodeGencode‘.‘gencode.v19.annotation.gtf‘ (Gencode version 19 annotation file)

Explanations: This query demonstrates the use of the **length()** function in the WHERE clause in filtering intervals. By changing the conditions in the WHERE clause, this query could also be used for identifying other types of sequence element.

SQ8 Analysis task: To count 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.

Query template:

```
SELECT    COUNT(*)
FROM      T
WHERE     feature = 'gene' AND
          attributes NOT LIKE '%gene_type “protein_coding”%';
```

Example of real data:

- T: ‘wgEncodeGencode‘.‘gencode.v19.annotation.gtf‘ (Gencode version 19 annotation file)

Explanations: This query demonstrates the use of the **COUNT()** function in the SELECT clause in computing an aggregated value of the resulting intervals. The selection condition in the WHERE clause also demonstrates how the NOT LIKE construct can be used to filter out protein coding genes from the results.

Composite queries

CQ1 Analysis task: To count 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) [11].

Query template:

```
FOR TRACK T IN (category=<track-category>, <track-selection-conditions>)
SELECT    chr, chrstart, chrend, value
FROM      T
COMBINED WITH UNION ALL AS Step1Results;

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

Example of real data:

- <track-category>: ‘SYDH TFBS‘ (ENCODE transcription factor binding signals from ChIP-seq experiments produced by the Stanford/ Yale/ Davis/ Harvard sub-group)

- <track-selection-conditions>: cell='GM12878' and fname LIKE '%Pk%' (considering only peak files from the cell line GM12878)

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 with a binding peak overlapping each resulting 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.

CQ2 Analysis task: To identify regions that 1) have active transcription factor binding, 2) are not within pre-defined promoter-proximal regulatory modules and 3) are at least 10kb away from high-confidence annotated genes. These regions are potentially gene-distal regulatory regions.

Query template:

CREATE TRACK Step1Results AS
SELECT NtInt_A.chr, NtInt_A.chrstart, NtInt_A.chrend
FROM (T_1 **exclusivejoin** T_2) NtInt_A;

CREATE TRACK Step2Results AS
SELECT NtInt_B.chr, NtInt_B.chrstart, NtInt_B.chrend
FROM Step1Results NtInt_B, T_3 TInt_3
WHERE TInt_3.feature = 'gene' AND
(TInt_3.attributes LIKE '%level 1%' OR
TInt_3.attributes LIKE '%level 2%') AND
distance(NtInt_B, TInt_3**)** < 10000;

SELECT *
FROM Step1Results **exclusivejoin** Step2Results;

Example of real data:

- T_1: 'HumanMetaTracks'.'BAR_Gm12878_merged.bed' (Regions with active transcription factor binding in GM12878 as defined in Yip et al. (2012) [11])
- T_2: 'HumanMetaTracks'.'PRM_Gm12878_merged.bed' (Promoter-proximal regulatory regions in GM12878 as defined in Yip et al. (2012) [11])
- T_3: 'wgEncodeGencode'.'gencode.v19.annotation.gtf' (Gencode version 19 annotation file)

Explanations: The first sub-query uses **exclusivejoin** to select regions with active transcription factor binding but are not within the pre-defined promoter-proximal regulatory regions. The second sub-query takes these regions and identifies those that are within 10,000bp 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. We designed three sub-queries for this task, rather than one single complex query (which is possible), to keep each sub-query short and easily understandable.

CQ3 Analysis task: To identify transcription factor binding regions, in the form of 100bp bins, that are at least 10kb 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 100bp bins as outputs for further analyses.

Query template:

```
FOR TRACK T IN (category=<track-category>, <track-selection-conditions>)
SELECT    chr, chrstart, chrend, value
FROM      T
COMBINED WITH UNION ALL AS Step1Results;

CREATE TRACK Step2Results AS
SELECT    NtInt$_A$.chr, NtInt$_A$.chrstart, NtInt$_A$.chrend
FROM      (project Step1Results on
          generate bins with length 100
with vd_sum using EACH MODEL) NtInt$_A$
WHERE     NtInt$_A$.value > 0;

CREATE TRACK Step3Results AS
SELECT    NtInt$_B$.chr, NtInt$_B$.chrstart, NtInt$_B$.chrend
FROM      T$_1$ TInt$_1$, Step2Results NtInt$_B$
WHERE     TInt$_1$.feature = 'gene' AND
          (TInt$_1$.attributes LIKE '%level 1%' OR
          TInt$_1$.attributes LIKE '%level 2%') AND
          distance(NtInt$_B$, TInt$_1$) < 10000;

SELECT    *
FROM      coalesce (
          SELECT    NtInt$_C$.chr, NtInt$_C$.chrstart, NtInt$_C$.chrend
          FROM      (Step2Results exclusivejoin Step3Results) NtInt$_C$
          ) NtInt$_D$;
```

Example of real data:

- <track-category>: 'SYDH TFBS' (ENCODE transcription factor binding signals from ChIP-seq experiments produced by the Stanford/ Yale/ Davis/ Harvard sub-group)
- <track-selection-condition>: cell='GM12878' and fname LIKE '%Pk%' (considering only peak files from the cell line GM12878)
- T$_1$: 'wgEncodeGencode'.'gencode.v19.annotation.gtf' (Gencode version 19 annotation file)

Explanations: The first sub-query stores all transcription factor binding peaks in a temporary track. The second sub-query maps these regions to 100bp 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 **exclusivejoin** to find bins far away from these genes, and join those that are adjacent into larger regions.

CQ4 Analysis task: To identify genomic regions, in the form of 2000bp bins, that overlap the binding peaks of at least 2 transcription factors. The average H3K27ac 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.

Query template:

```
FOR TRACK T IN (category=<track-category>, <track-selection-conditions>)
SELECT     NtInt_A.chr, NtInt_A.chrstart, NtInt_A.chrend, NtInt_A.value
FROM       (project T on
           generate bins with length 2000
with vd_sum using EACH MODEL) NtInt_A
WHERE      NtInt_A.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     NtInt_B.chr, NtInt_B.chrstart, NtInt_B.chrend, NtInt_B.value
FROM       (project T on Step3Results
with vd_sum using EACH MODEL) NtInt_B;

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

Example of real data:

- <track-category>: ‘SYDH TFBS‘ (ENCODE transcription factor binding signals from ChIP-seq experiments produced by the Stanford /Yale /Davis /Harvard sub-group)
- <track-selection-conditions>: cell=’K562’ and fname LIKE ’%Pk%’ (considering only peak files from the cell line K562)
- T: ‘wgEncodeBroadHistone‘. ‘wgEncodeBroadHistoneK562H3k27acStdSig.bigWig‘ (An ENCODE ChIP-

seq data file of H3K27ac signals in the K562 cell line produced by the Broad Institute)

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 2000bp 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 of (e.g., 2) 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 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.

CQ5 Analysis task: To identify 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.

Query template:

```
CREATE TRACK Step1Results AS
SELECT    chr, chrstart, chrend, strand
FROM      T_1
WHERE     feature = 'gene' AND
          attributes LIKE '%gene_type "protein_coding"%';

CREATE TRACK Step2Results AS
SELECT    DISTINCT NtInt_A.chr, NtInt_A.chrstart, NtInt_A.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 = '-') NtInt_A;

CREATE TRACK Step3Results AS
SELECT    NtInt_B.chr, NtInt_B.chrstart, NtInt_B.chrend,
          NtInt_B.value - NtInt_C.value as value
FROM      (project T_2 on Step2Results
with vd_sum using EACH MODEL) NtInt_B,
          (project T_3 on Step2Results
with vd_sum using EACH MODEL) NtInt_C
```

```
WHERE     NtInt_B coincides with NtInt_C;

CREATE TRACK Step4Results AS
SELECT    NtInt_D.chr, NtInt_D.chrstart, NtInt_D.chrend,
          NtInt_D.value - NtInt_E.value as value
FROM      (project T_4 on Step2Results
with vd_sum using EACH MODEL) NtInt_D,
          (project T_5 on Step2Results
with vd_sum using EACH MODEL) NtInt_E
WHERE     NtInt_D coincides with NtInt_E;

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

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

SELECT    *
FROM      (SELECT   NtInt_H.chr, NtInt_H.chrstart, NtInt_H.chrend,
                    NtInt_H.strand
          FROM      Step1Results NtInt_H,
                    (SELECT   chr, chrstart +1500 AS chrstart,
                              chrstart +1500 AS chrend
                    FROM      Step6Results) NtInt_I
          WHERE     NtInt_H.strand = '+' AND
                    NtInt_I is prefix of NtInt_H
          UNION ALL
          (SELECT   NtInt_J.chr, NtInt_J.chrstart, NtInt_J.chrend,
                    NtInt_J.strand
          FROM      Step1Results NtInt_J,
                    (SELECT   chr, chrend-1500 AS chrstart,
                    chrend-1500 AS chrend
                    FROM      Step6Results) NtInt_K
          WHERE     NtInt_J.strand = '-' AND
                    NtInt_K is suffix of NtInt_J) NtInt_L;
```

Example of real data:

- T_1: ‘wgEncodeGencode‘.‘gencode.v19.annotation.gtf‘ (Gencode version 19 annotation file)

- T_2: ‘wgEncodeSydhTfbs‘. ‘wgEncodeSydhTfbsGm12878JundIggrabSig.bigWig‘ (An ENCODE ChIP-seq data file of Cjun binding signals in the GM12878 cell line produced by the Stanford/Yale/Davis/Harvard sub-group)
- T_3: ‘wgEncodeSydhTfbs‘. ‘wgEncodeSydhTfbsGm12878InputStdSig.bigWig‘ (An ENCODE control experiment file using input DNA in the GM12878 cell line produced by the Stanford/Yale/Davis/Harvard sub-group)
- T_4: ‘wgEncodeSydhTfbs‘. ‘wgEncodeSydhTfbsK562JundIggrabSig.bigWig‘ (An ENCODE ChIP-seq data file of Cjun binding signals in the K562 cell line produced by the Stanford/Yale/Davis/Harvard sub-group)
- T_5: ‘wgEncodeSydhTfbs‘. ‘wgEncodeSydhTfbsK562InputStdSig.bigWig‘ (An ENCODE control experiment file using input DNA in the K562 cell line produced by the Stanford/Yale/Davis/Harvard sub-group)

Explanations: The first sub-query identifies all protein-coding genes. The second sub-query defines the promoter of each gene as the region from 1500bp upstream of the transcription start site to 500bp 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.

Since the results of the first two sub-queries are frequently used, they can be pre-constructed for reuse by various queries, which would simplify the whole analysis procedure. START allows users to store their custom tracks, which will be explained in the next section.

CQ6 Analysis task: To identify genomic regions with bi-directional transcription at their flanking regions (Figure S1), which could be potential enhancers producing enhancer RNAs (eRNAs) [**? ?**].

Query template:

```
CREATE TRACK Step1Results AS
SELECT    chr, chrstart - 200 AS chrstart, chrend - 200 AS chrend
FROM      $T_1$
WHERE     value > 2;

CREATE TRACK Step2Results AS
SELECT    chr, chrstart + 200 AS chrstart, chrend + 200 AS chrend
FROM      $T_2$
WHERE     value > 2;

SELECT    *
FROM      Step1Results intersectjoin Step2Results;
```

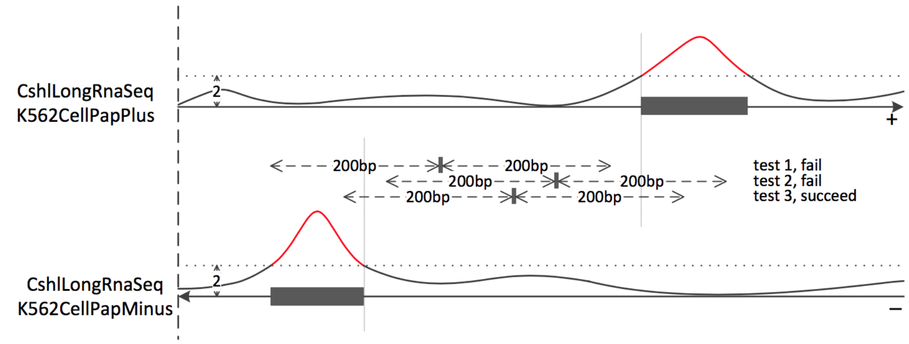


Figure S1 Illustration of the regions to be identified in this query. RNA-seq signals on the two strands are shown in the two tracks. In the middle we show three genomic regions in the form of vertical bars. The first and second regions do not satisfy the query requirements since they miss significant 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.

Example of real data:

- T_1: ‘wgEncodeCshlLongRnaSeq‘. ‘wgEncodeCshlLongRnaSeqK562CellPapPlusRawSigRep1.bigWig‘ (An ENCODE RNA-seq data file of total long RNA of the positive strand in the K562 cell line produced by the Cold Spring Harbor Laboratory)
- T_2: ‘wgEncodeCshlLongRnaSeq‘. ‘wgEncodeCshlLongRnaSeqK562CellPapMinusRawSigRep1.bigWig‘ (An ENCODE RNA-seq data file of total long RNA of the negative strand in the K562 cell line produced by the Cold Spring Harbor Laboratory)

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 200bp 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 200bp. 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 200bp downstream on the positive strand and 200bp upstream on the negative strand, which forms a bi-directional pattern indicative of eRNA [**?**].

Comparisons between STQL and SQL

Since STQL has an SQL-like syntax and a data model that is essentially a relation, one may wonder whether STQL queries can be easily expressed in SQL. In this section, we use four examples to show that some operations are much more difficult to perform using SQL than STQL.

Ex.1 The first example involves the **is closest to each** construct. In STQL, it is easy to find out the interval(s) in track T_2 closest to each interval in track T_1 using the following query:

```
SELECT     T1.chr, T1.chrstart AS start1, T1.chrend AS end1, T2.chrstart AS start2, T2.chrend AS end2
FROM       T1 TInt1, T2 TInt2
WHERE      TInt1 is closest to each TInt2;
```

To perform the same operation in SQL, three steps are needed, namely 1) computing the distance of all interval pairs from the two tracks, 2) finding the minimum distance for each interval in T_1, and 3) retrieving the corresponding pairs with these minimum distances:

```
WITH
/* 1. Calculate the distance between all pairs of intervals from T1 and T2 */
T1sDistance AS (
SELECT     T1.chr, T1.chrstart AS start1, T1.chrend AS end1, T2.chrstart AS start2, T2.chrend AS end2,
           CASE /* Different cases for calculating distance between two intervals */
           WHEN T1.chrstart <= T2.chrend AND T1.chrend >= T2.chrstart
                      THEN 0
           WHEN T1.chrend < T2.chrstart
                      THEN T2.chrstart - T1.chrend
           WHEN T1.chrstart > T2.chrend
                      THEN T1.chrstart - T2.chrend
           END AS distance
FROM       T1, T2
WHERE      T1.chr = T2.chr
),

/* 2. Calculate the minimum interval distance of each interval in T1 */
T1sMinDistance AS (
           SELECT     chr, start1, end1, min(distance) AS minDistance
           FROM       T1sDistance
           GROUP BY chr, start1, end1
)

/* 3. Find out the closest pairs based on the minimum distances */
SELECT     *
FROM       T1sDistance a
           LEFT JOIN T1sMinDistance b
                      ON a.chr = b.chr
                      AND a.start1 = b.start1
                      AND a.end1 = b.end1
WHERE      distance = minDistance
```

Ex.2 The second example involves the **coalesce** construct. Sample query SQ5 demonstrates how all the genomic positions with certain level of transcription signals are merged into disjoint regions using **coalesce** . Since each interval needs to be combined with an indefinite number of other intervals to form an output region, the operation cannot be performed using standard SQL. We wrote the following SQL query involving recursion to handle this task:

```
WITH RECURSIVE /* Recursively coalesce the intervals */
CoalesceGroup AS (
SELECT      chr, chrstart, chrend
FROM T
WHERE T.value > 2
UNION
SELECT      a.chr, a.chrstart, T.chrend
FROM        CoalesceGroup AS a
/* Join overlapping intervals */
JOIN c on    a.chr = c.chr
            AND a.chrstart <= c.chrend + 1 AND a.chrend >= c.chrstart - 1
WHERE       c.value > 2
),
CoalesceGroup2 AS (
SELECT      *, row_number() OVER (PARTITION BY chr, chrend ORDER BY chrstart) AS rn
FROM        CoalesceGroup
)

/* Compute the values of the output intervals */
SELECT      a.chr, a.chrstart, a.chrend, AVG(c.value) as value
FROM        (
            SELECT      chr, chrstart, MAX(chrend) AS chrend
            FROM        CoalesceGroup2
            WHERE       rn = 1
            GROUP BY chr, chrstart) a
            LEFT JOIN c ON a.chrstart <= c.chrend AND a.chrend >= c.chrstart
WHERE       c.value > 2
GROUP BY a.chr, a.chrstart, a.chrend
```

Ex.3 The third example involves the **discretize** construct. STQL can be used to discretize the intervals in a track into non-overlapping intervals:

```
    SELECT      *
    FROM        discretize T with vd_sum using EACH MODEL
```

The same operation can be performed by the following SQL query:

```
/* Determine the non-overlapping intervals */
WITH allPos AS (
SELECT      row_number() OVER (PARTITION BY chr ORDER BY pos) AS rn, chr, pos
FROM        ((
                        SELECT chr, d.chrstart AS pos FROM d GROUP BY chr, pos
                        UNION
                        SELECT chr, d.chrend+1 AS pos FROM d
                        WHERE d.chrend != (SELECT MAX(d.chrend) FROM d) GROUP BY chr, pos
            )
            UNION ALL
            (
                        SELECT chr, d.chrstart-1 AS pos FROM d
                        WHERE d.chrstart != (SELECT MIN(d.chrstart) FROM d) GROUP BY chr, po
                        UNION
                        SELECT chr, d.chrend AS pos FROM d GROUP BY chr, pos
            )) tmpUnion
),

grouping AS (
SELECT      a.chr, a.pos AS chrstart, b.pos AS chrend
FROM        allPos a LEFT JOIN allPos b ON a.rn+1 = b.rn AND a.chr = b.chr
WHERE       a.rn % 2 = 1)

/* Compute the values of the output intervals */
SELECT      a.chr, a.chrstart, a.chrend, SUM(d.value) AS value
FROM        grouping a JOIN d ON a.chr = d.chr AND a.chrstart <= d.chrend AND a.chrend >= d.chrsta
GROUP BY a.chr, a.chrstart, a.chrend
```

Ex.4 The last example invovles the **project on generate bins with length** constructs. Sample query SQ1 demonstrates how the average signal within each 100bp genomic bin can be easily computed using these constructs. To perform the same operation in SQL, it has to first define a new table consisting of the bin definitions, and then compute the average signal value in each bin:

```
/* 1. Create the bin definitions using the GENERATE_SERIES function in PostgreSQL */
WITH
Bin AS (
SELECT’chr1’ AS chr, c.b+1 AS chrstart, c.b+100 AS chrend FROM GENERATE_SERIES(0, 249250621,
UNION ALL
SELECT’chr2’ AS chr, c.b+1 AS chrstart, c.b+100 AS chrend FROM GENERATE_SERIES(0, 243199373,
UNION ALL
SELECT’chr3’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 198022430
UNION ALL
SELECT’chr4’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 191154276
UNION ALL
SELECT’chr5’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 180915260
UNION ALL
SELECT’chr6’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 171115067
UNION ALL
SELECT’chr7’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 159138663
UNION ALL
SELECT’chr8’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 146364022
UNION ALL
SELECT’chr9’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 141213431
UNION ALL
SELECT’chr10’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 13553474
UNION ALL
SELECT’chr11’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 13500651
UNION ALL
SELECT’chr12’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 13385189
UNION ALL
SELECT’chr13’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 11516987
UNION ALL
SELECT’chr14’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 10734954
UNION ALL
SELECT’chr15’ AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 10253139
UNION ALL
```

```
SELECT'chr16' AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 90354753
UNION ALL
SELECT'chr17' AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 81195210
UNION ALL
SELECT'chr18' AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 78077248
UNION ALL
SELECT'chr19' AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 59128983
UNION ALL
SELECT'chr20' AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 63025520
UNION ALL
SELECT'chr21' AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 48129895
UNION ALL
SELECT'chr22' AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 51304566
UNION ALL
SELECT'chrX' AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 15527056
UNION ALL
SELECT'chrY' AS chr, c.b+1 AS chrstart, c.b + 100 AS chrend FROM GENERATE_SERIES(0, 59373566
)

/* 2. Compute average signal value of each bin */
SELECT    *
FROM      (
          SELECT    Bin.chr, Bin.chrstart, Bin.chrend,
                    SUM(CASE /* Different cases based on the intersection between an interval an a
                    WHEN Bin.chrstart > T.chrstart AND Bin.chrend > T.chrend
                              THEN T.chrend - Bin.chrstart + 1
                    WHEN Bin.chrstart <= T.chrstart AND Bin.chrend > T.chrend
                              THEN T.chrend - T.chrstart + 1
                    WHEN Bin.chrstart > T.chrstart AND Bin.chrend <= T.chrend
                              THEN Bin.chrend - Bin.chrstart + 1
                    WHEN Bin.chrstart <= T.chrstart AND Bin.chrend <= T.chrend
                              THEN Bin.chrend - T.chrstart + 1
                    END * T.value)/(Bin.chrend - Bin.chrstart + 1) AS value
          FROM      Bin LEFT JOIN T on /* Join all overlapping intervals and bins */
                    (Bin.chrstart <= T.chrend and Bin.chrend >= T.chrstart AND Bin.chr = T.chr)
          GROUP BY Bin.chr, Bin.chrstart, Bin.chrend ) AS Ntint
WHERE     Ntint.value > 0
```