Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

Pao Yue-kong Library, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

http://www.lib.polyu.edu.hk

START: A SYSTEM FOR FLEXIBLE ANALYSIS OF HUNDREDS OF GENOMIC SIGNAL TRACKS IN FEW LINES OF SQL-LIKE QUERIES

ZHANG QIANG

Ph.D

The Hong Kong Polytechnic University

2016

The Hong Kong Polytechnic University

Department of Computing

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

ZHANG Qiang

A thesis submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

April 2016

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

. . . .

ZHANG Qiang

Abstract

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.

This thesis presents Signal Track Analytical Research Tool (START) and Signal Track Query Language (STQL) for easy analysis of signal tracks. STQL is an 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), a MapReduce-based system that includes a Web-based user interface and a back-end execution system.

By running some typical analyses tasks, we show that the START+STQL solution is usually the simplest, and the parallel execution achieves significant speed-up with large data files.

Acknowledgments

My research work presented in this thesis will be surpassed in future, however, my deepest feelings to all these people mentioned here is endless, which surpass space and distance.

First and foremost, for the development and production of this thesis itself, I feel a deep sense of gratitude to my supervisor: Dr. Eric Lo. It is my honor to be his Ph.D. student. I was often motivated by his passion for research, his inspiration especially his smart way of problem-solving. I appreciate all his contributions of time, ideas, and funding to help me through the Ph.D. journey. I am also thankful for the excellent example he has provided as a successful individual marching to one's goal.

I also thank Dr. Kevin Yip for his useful suggestions and guidance when we cooperate in the research project. Dr. Ben Kao is very insightful when discussing the research, thanks to him. Dr. Ken Yiu also provided me excellent advice when I encountered difficulties during the research process. Thanks to Xinjie Zhu and Eric Ho's help during the system development.

Thanks to my parents who raised me. They are always the tower of strength.

Every weekend's video chat with them is the most relaxing time during my postgraduate years. I would not have gone so far without their understanding and unconditional support.

Last but not least, I am indebted to all the buddies in our Database Group. We have spent a lot of happy and valuable time in the past years. They are Duncan Yung, Jianguo Wang, Cliz Sun, Tanya Aldyn-ool, Andy He, Jeppe Thomsen, Yu Li, Petrie Wong, Ziqiang Feng, Bob Tang, Captial Li, Wenjian Xu, Ran Bai, Chuanfei Xu, Chris Liu, Julia Lp, Lili Zhang, Edison Chan.

There are still many friends who accompany me during the whole Ph.D. journey, they always occupy a space in my heart.

Contents

Declaration **i**

Abstract **iii**

Acknowledgments **v**

Contents **vii**

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**

2 Background **7**

2.1 Genomic Data Analysis System 7

2.2 Large Scale Data Processing System 9

3 Signal Track Analytical Research Tool **11**

3.1 Data Model 11

3.2 Basic constructs in STQL 13

3.2.1 The SELECT clause 13

3.2.2 The FROM clause 14

3.2.3 The WHERE clause 17

3.2.4 Other optional clauses 19

3.3 Advanced constructs in STQL 20

3.3.1 Creating a new track from an existing track 20

3.3.2 Creating a new track from two existing tracks 22

3.3.3 Value derivation and inheritance of metadata 26

3.3.4 Using dynamically created tracks 30

3.3.5 Data definition and manipulation statements 30

3.3.6 Selection and looping over signal tracks 32

3.4 Signal Track Analytical Research Tool (START) 33

3.4.1 Front-end: Web-based user interface 35

3.4.2 Back-end: parallel execution system 37

3.4.3 Interface between front-end and back-end: Metastore . . . 41

3.5 Results 41

3.5.1 Example queries 41
3.5.2 Comparison with other approaches 44
3.5.3 Case study 46

4 Closest Interval Join Using MapReduce **49**

4.1 Related Work 54
4.1.1 Overlap Interval Join on MapReduce 55
4.1.2 kNN join on MapReduce 58
4.1.3 Similarity Join on MapReduce 63
4.1.4 Centralized Overlap Interval Join Algorithms 64
4.1.5 Summary 66
4.2 The Solutions 66
4.2.1 Broadcast Algorithm(BA) 67
4.2.2 Neighbor Replicating Algorithm(NRA) 68
4.2.3 Distance-Aware Algorithm(DAA) 70
4.3 Experiments 81
4.3.1 Real Data 82
4.3.2 Synthetic Data 84
4.3.3 Summary 89

5 Conclusions **91**

Appendix A Full set of example queries **95**

A.1 Simple queries 95

A.2 Composite queries 102

Appendix B STQL grammar rules **115**

Bibliography **121**

List of Figures

1.1 UCSC genome browser example 3

1.2 Growth of DNA Sequencing 4

3.1 An example that illustrates the **coalesce** operator 21

3.2 An example that illustrates the **discretize** operator 22

3.3 An example that illustrates the **project on** operator 24

3.4 An example that illustrates the **intersectjoin** operator 24

3.5 An example that illustrates the **exclusivejoin** operator 25

3.6 The overall architecture of START 34

3.7 The user interface of START 36

3.8 A typical MapReduce job created by the executor from an STQL query 39

4.1 Closest interval join example 50

4.2 Basic DNA structure 50
4.3 Find closest genes for SNPs 51
4.4 Overlap interval join example 53
4.5 Overlap interval join in detail 56
4.6 Emitted intermediate results by mappers 60
4.7 The partitioning of R and S 61
(a) Partitioning of R 61
(b) Partitioning of S 61
4.8 The bounding and pruning techniques 63
4.9 Interval data cannot apply triangle inequality 63
4.10 An example about NRA 69
4.11 Motivating example 72
4.12 The new boundary for each partition 72
4.13 An example shows some intervals intersect more than one partition 73
4.14 An example shows duplicate results 74
4.15 Three different regions in terms of a query interval 77
4.16 An example used for our algorithm 78
4.17 Time breakdown for real data 82
4.18 Shuffling cost for real data 83

4.19 Execution Time vs. Data Size 85

4.20 Shuffling Cost vs. Data Size 86

4.21 Time breakdown for G2 86

4.22 Execution Time vs. Number of Partitions 87

4.23 Execution Time vs. Number of Slave Nodes 88

4.24 Varying the skewness on interval length 88

4.25 Varying the skewness of starting position 89

List of Tables

3.1 Relations defined in STQL for comparing intervals 18

3.2 The full list of mathematical operations in STQL 28

3.3 Number of tokens involved between different systems 44

3.4 Execution time between different systems 45

4.1 Intermediate results output by each mapper 57

4.2 Intervals received in each reducer 57

4.3 The blocks each reducer receives 59

4.4 Default parameters for synthetic data generator 84

4.5 Varying data size 85

Chapter 1

Introduction

The rapid development of new applications of high-throughput sequencing and the sharp reduction in 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 [55]. Major collaborative projects such as ENCODE [56] and Roadmap Epigenomics [48] 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 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.

Figure 1.1 shows a screenshot of UCSC genome browser [13]. It displays the detailed information for a region of DNA of chromosome 21. Each track is represented by a horizontal line. Some tracks such as UCSC genes, RefSeq genes only show the gene locations while some such as Layered H3K27AC shows the signal values. Single nucleotide polymorphism(SNP) genes are also showed in this figure. Each track is an interval set which particularly describes the information about different regions of the DNA. From the figure, we can see that every displayed item in a track spans a series of continuous positions of the DNA sequence. These items are the genomic intervals we will process.

Signal track data are obtained from DNA sequencing. More and more DNA sequencing data means we can extract more signal track data. Figure 1.2 shows the increase of DNA sequencing data every year. The left axis shows the total number of human genomes sequenced and the right axis shows the annual sequencing capacity in the world. The 1000 Genomes Project [23], TCGA [20],

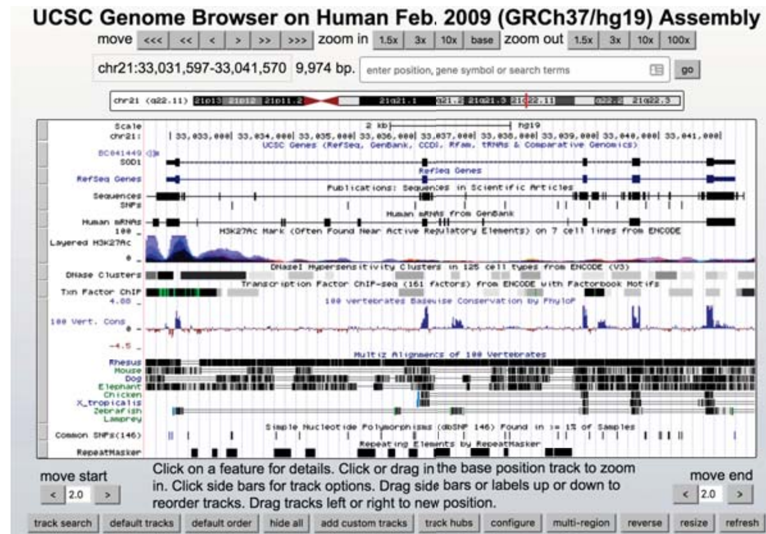


Figure 1.1: UCSC genome browser example

ExAC [7] are representative projects that aim for different sequencing goals.

Analysis of signal tracks usually involves multiple steps. Typical operations at each step include a 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.

As the volume and complexity of signal track data have both increased dramatically in recent years [62], 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

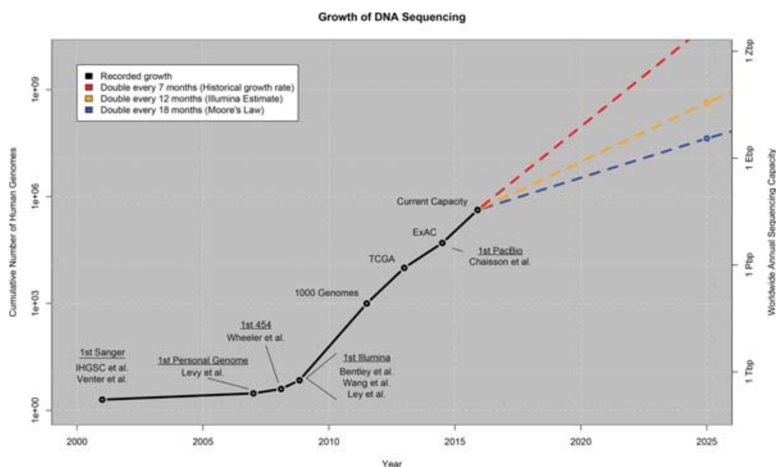


Figure 1.2: Growth of DNA Sequencing [62]

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, in this thesis 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.

We have also implemented a system based on MapReduce [2] 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 con-

struct 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 ENCODE, RoadMap Epigenomics, FANTOM5 [16], and other sources. 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, which comes with a detailed installation guide.

During the system implementation, other related research problems arose. Interval join is an important operation on interval data in START in order to associate related intervals. It is like Theta-join [44], in which a pair of intervals must satisfy the join condition to output possible results. *Closest interval join* is a join operation between two interval sets, which can be described as follows.

Given two sets of intervals R and S, the *closest interval join* of R against S means, for each $r \in R$, find all the intervals in S that are closest to r. r's closest intervals from S means among all the intervals of S, there are some intervals which have the minimum distance between it and r. The distance between two intervals is defined by the minimum length expansion such that these two intervals can intersect with each other. If two intervals have already intersected, their distance is 0.

To improve the efficiency of START, we studied the closest interval join

operation in detail and proposed a new MapReduce algorithm to handle it.

To summarize, our main contributions include:

- Considering the engineering implementation efficiency and common users' habits, we proposed STQL to help users express the complex tasks involved in signal track analysis, which make it much easier for them to finish the tasks.
- To support STQL and massive signal track data, we designed and implemented a MapReduce-based system, START, to store, retrieve and compute these signal track data.
- To optimize an important join operator for signal track data analysis, we proposed a novel MapReduce algorithm. Experiments with both of the real and synthetic data show its performance efficiency comparing with other approaches.

The remainder of this thesis is organized as follows: Chapter 2 introduces some existing systems which handle genomic data and large scale data processing systems. Chapter 3 presents the detail design and implementation of STQL and START. Chapter 4 presents the novel MapReduce algorithm to handle the closest interval join operation in START. This work has been published in [64]. Chapter 5 concludes the thesis.

Chapter 2

Background

It has been a long history for bioinformaticians to build the corresponding system to do genomic data analysis, and it is more popular to do the comprehensive analysis of genomic data in academic and clinical research contexts in recent years [22]. With the rapid growth of genomic data [8], it is natural to consider storing and processing genomic data with modern large scale data processing system. So we first introduce some traditional genomic data processing systems and then some new emerging large scale data processing systems.

2.1 Genomic Data Analysis System

Data visualization can facilitate the ability of humans to absorb and understand information. In order to satisfy this requirement, some web-based programs were built which allow users to view genomic data, especially human genome graphically. Three major browsers are developed in this area: UCSC

genome browser [13], Ensembl [6] and MapViewer [12].

UCSC genome browser is an interactive website for users to access genome sequence data. It uses MySQL as the back-end database system to store all the genomic data and is optimized to support fast interactive performance. To speed up the interaction when performing the queries, some indexes are created. The goal of Ensembl is to automatically annotate the genome, integrate this annotation with other available data and make all of them publicly available via the web. It has integrated source of genome annotations for many different species. It also provides APIs for users to access. Map Viewer allows users to view and search organism's complete genome, display chromosome maps and zoom into progressively greater levels of details, down to the sequence data for a region of interest. Another similar tool is Integrative Genomics Viewer [10].

InterMine [11] is an open source data warehouse built specifically for the integration and analysis of complex biological data. It uses PostgreSQL as the back-end database system. It provides a Parser that can integrate data from many common biological data sources and formats. In addition to that, it provides a framework to add user's data. Moreover, it has a built-in attractive, user-friendly web interface that works 'out of the box' and can be easily customized.

Galaxy [32] is another popular web-based genome analysis system. Similar to other systems, it can allow users to upload their own data and apply various computational tools to analyze the data. Most of the computational tools are standalone tools written with python or C++, and the Galaxy developer can easily upgrade these tools without changing the UI. The most useful feature of

Galaxy is that it can record each procedure applied to the data, so it is very convenient for users to share the data and the whole workflow with other users.

BEDTools [5] is a powerful toolset for genomic arithmetic. These tools can allow a user to intersect, merge, count, complement and shuffle genomic intervals from multiple files in widely-used genomic file formats such as BAM, BED, GFF/GTF, VCF. They are UNIX-style command line tools, so users have to specify all parameters from the command line to manipulate them. Each tool is designed to finish a relatively simple task, so users need to think out their own workflow carefully to combine multiple tools.

2.2 Large Scale Data Processing System

MapReduce [26] is a popular programming model for large scale data storing and processing. It breaks a program into a map and a reduce function. All of the data are represented by key-value pairs during the whole program. Multiple machines can apply the same map function to different parts of the input data and generate key-value pairs as the intermediate results. The intermediate results are first grouped and then shuffled to different machines according to the keys through the network. Multiple machines can apply reduce function concurrently on the intermediate results to compute the final results.

Hadoop [2] is a typical implementation of MapReduce. It uses HDFS [52] to store large data. JobTracker and TaskTracker are designed to control the map and reduce task. As the intermediate result are serialized to local hard disk before being shuffled, Hadoop provides excellent fault tolerance.

To improve the speed of Hadoop, Spark [4] is designed to support in-memory processing. All of the intermediate results are stored and processed in memory. The core of Spark is a data structure called the resilient distributed dataset (RDD). It is a read-only multi-set of data items distributed over a cluster of machines, and it can also provide fault tolerant function.

In order to make it much easier for a common user to express the analytical task, Hive [57] is designed as a data warehousing infrastructure built on top of MapReduce. It supports an SQL-like language called HiveQL and translates it into MapReduce jobs. Pig [3] is also a large scale data analysis platform built on top of MapReduce. Unlike Hive, Pig offers a textual language called Pig Latin [45], which has advantages such as ease of programming, opportunities for optimization and extensibility. Pig also translates the job expressed with Pig Latin into MapReduce jobs at the end.

Chapter 3

Signal Track Analytical Research Tool(START) *

In this chapter, we first describe the constructs of the SQL-like language STQL, then introduce its back-end system START, finally we present example queries and the comparison with other approaches.

3.1 Data Model

Our data model is very similar to relational data model [21]. The relational data model is based on a relation or a table. A tuple or a row contains all the data of a single instance of the table. A column identifies an attribute. An attribute value has a type of that attribute. In addition to support basic attribute types

*This is a joint work with Xinjie Zhu. I am responsible for the system design, implementation, verification and all the experiments.

such as integer, string, float point number, we proposed an **interval** data type to represent DNA segment and **track** which is table-like that contains a set of intervals.

An **interval** is a compound data type, which contains several fields:

- *chr:* It indicates the chromosome to which a DNA segment belongs. The type is string, the value is like "chr1", "chr2".
- *chrstart:* It indicates the start position (inclusive) of the DNA segment. The type is integer.
- *chrend:* It indicates the end position (inclusive) of the DNA segment. The type is integer.
- *value:* It is a value associated with the DNA segment. The type is float point number. The value can be obtained by different interpreting of the DNA segment.
- *strand:* DNA has double helix structure, so this field indicates on which strand the DNA segment is. The type is string. The value can be "+", "-" or ".". "+" or "-" indicates the two strands respectively, "." means don't know or don't care which strand it is on.
- *other metadata:* There are also other optional information to describe features of the DNA segment. For example, a source field indicates which program generated the information.

In most cases, a DNA segment can be expressed with an interval type like this [*chr*, *chrstart*, *chrend*] or [*chr*, *chrstart*, *chrend*, *strand*] if we care which strand

it is from.

Track is like a table, and it is a set of meaningful DNA segments. For example, SNPs can be considered a track which contains SNP intervals.

3.2 Basic constructs in STQL

The formal grammar of STQL is given in Appendix B . 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';
```

3.2.1 The SELECT clause

The SELECT clause includes a comma-separated list of attributes of the queried intervals to be returned. 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. 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, 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;
```

3.2.2 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    T1.chr, T1.chrstart, T1.chrend, T1.value,
          T2.chr AS chr2, T2.chrstart AS chrstart2,
          T2.chrend AS chrend2, T2.value AS value2
FROM      T1, T2;
```

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   T1.chr, T1.chrstart, T1.chrend, T1.value,
         T2.chr AS chr2, T2.chrstart AS chrstart2,
         T2.chrend AS chrend2, T2.value AS value2
FROM     T1 RIGHT JOIN T2 ON T1.chr=T2.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, more efficient algorithms are used to avoid performing the costly Cartesian product.

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.

3.2.3 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., **length(**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 3.1 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:

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	Finding genomic bins with positive signals in two replicated experiments
I_1 **overlaps with** I_2	I_1.chr = I_2.chr and I_1.chrstart ≤ I_2.chrend and I_1.chrend ≥ I_2.chrstart	Counting the number of sequencing reads that overlap with each promoter
I_1 **contains** I_2	I_1.chr = I_2.chr and I_1.chrstart ≤ I_2.chrstart and I_1.chrend ≥ I_2.chrend	Finding transcription factor binding sites that contain single nucleotide variants
I_1 **is within** I_2	I_1.chr = I_2.chr and I_1.chrstart ≥ I_2.chrstart and I_1.chrend ≤ I_2.chrend	Checking if a gene is within a certain haplotype block
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)	Finding 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 ≤ I_2.chrend	Finding 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 ≥ I_2.chrstart and I_1.chrend = I_2.chrend	Finding 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 intervals on the same chromosome
I_1 **follows** I_2	I_1.chr = I_2.chr and I_1.chrstart > I_2.chrend	Ordering 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))	Defining 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))	Finding sequence elements downstream of a sequence motif

Table 3.1: Relations defined in STQL for comparing intervals

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

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).

3.2.4 Other optional clauses

Similar to SQL, STQL provides a GROUP BY clause for grouping intervals and performing aggregations (**COUNT()**, **SUM()**, **AVG()**, **MIN()**, **MAX()**, etc.) 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.

3.3 Advanced constructs in STQL

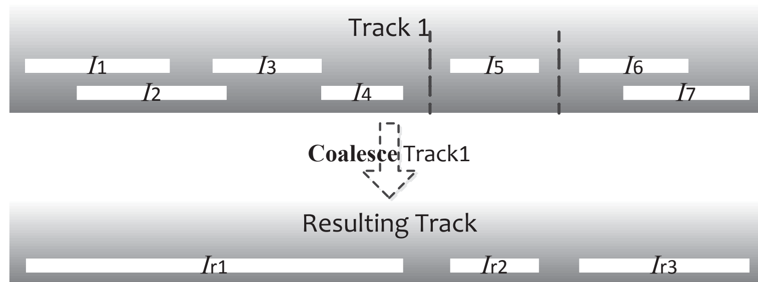
3.3.1 Creating a new track from an existing track

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 [60]. This type of operations can be performed by using the **coalesce** operator, which 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 3.1. Complete query examples using **coalesce** and other advanced constructs will be given later.

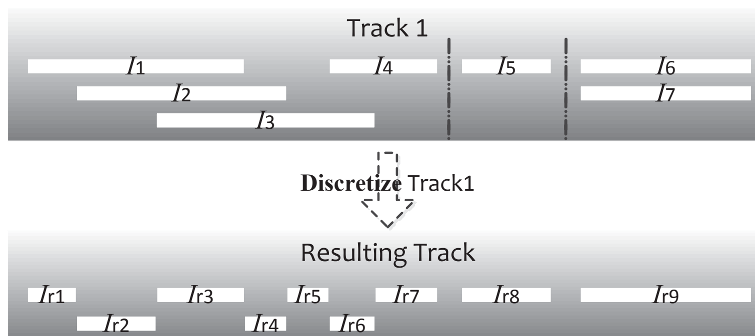


Note: $I_1..I_7$ are intervals in the input track that are allowed to overlap with or be adjacent to each other, while $I_{r1}..I_{r3}$ are the non-overlapping, non-adjacent intervals in the resulting track after the coalesce operation. I_{r1} is formed by merging I_1, I_2, I_3 and I_4, which occupy 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 3.1: An example that illustrates the **coalesce** operator

Another common operation for processing overlapping regions is to use their boundary locations to define discrete intervals (Figure 3.2). 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>]



Note: $I_1..I_7$ 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.

Figure 3.2: An example that illustrates the **discretize** operator

3.3.2 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 3.3). 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**]]

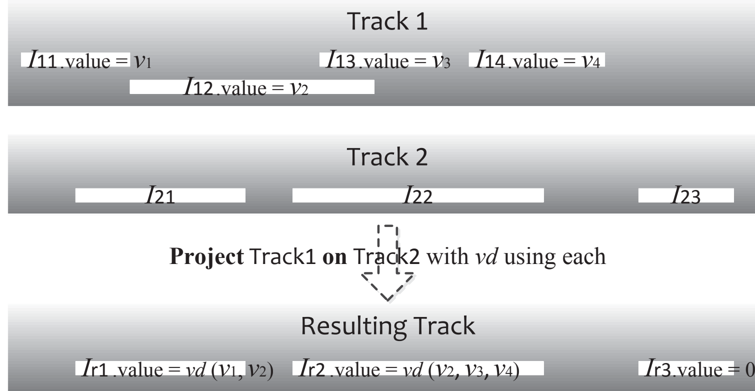
where <bin-size> is the size of each bin in base pairs.

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 3.4). 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 3.5):



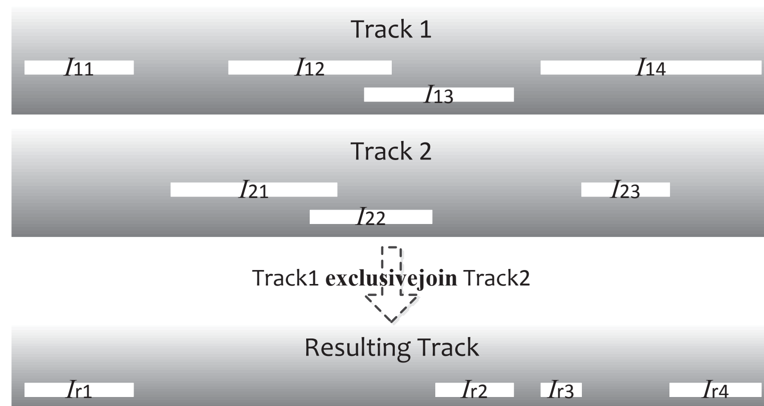
Note: $I_{11}..I_{14}$ are intervals in input track 1, $I_{21}..I_{23}$ are intervals in input track 2, while $I_{r1}..I_{r3}$ are intervals in the resulting track after the projection. The locations of the intervals in the resulting track are directly from the intervals in input track 2. The value of 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 3.3: An example that illustrates the **project on** operator

Note: $I_{11}..I_{14}$ 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 3.4: An example that illustrates the **intersectjoin** operator

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



Note: $I_{11}..I_{14}$ 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 3.5: An example that illustrates the **exclusivejoin** operator

Finally, UNION ALL forms a new track that keeps all intervals from the two input tracks without removing duplicates. 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;
```

3.3.3 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 exact raw read counts, an interval having a certain value means that every genomic location in the interval is covered by that number of reads. 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 a gene is represented by an interval, and its value indicates its expression level, all genomic locations of the gene collectively own the expression value.

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 3.4 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}.**length**(I_{11}) + I_{21}.**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**.

STQL operation	**coalesce discretize 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

Note: 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.

Table 3.2: The full list of mathematical operations in STQL

Table 3.2 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.

3.3.4 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.

3.3.5 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 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.

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.

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.

3.3.6 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:

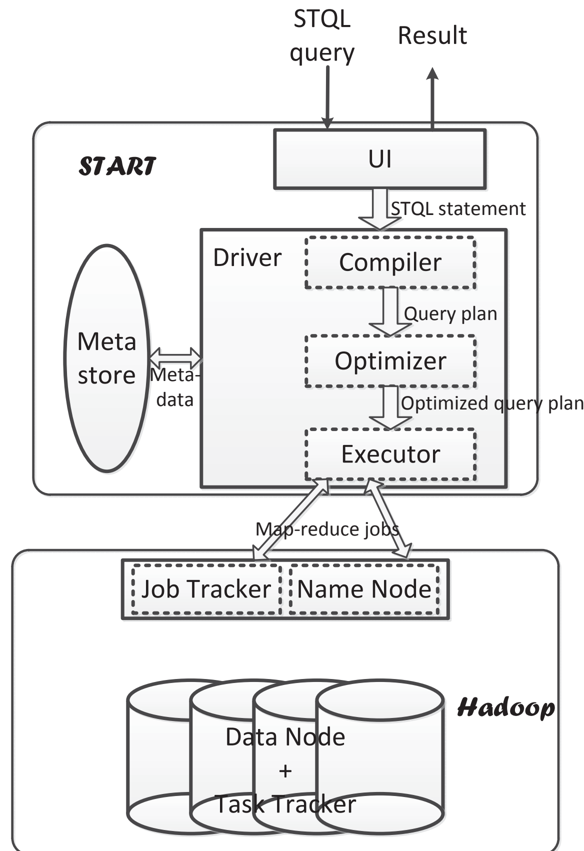
```
FOR TRACK TInt IN (category=‘SYDH TFBS’,
cell=‘GM12878’ and fname LIKE ‘%Pk%’)
SELECT     TInt.chr, TInt.chrstart, TInt.chrend
FROM       TInt
COMBINED WITH UNION ALL AS AllPeaks;
```

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.

3.4 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 3.6).



Note: 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.

Figure 3.6: The overall architecture of START

The purpose of the Web-based user interface is to provide a simple way for users to execute STQL. We have pre-loaded around 10,000 signal tracks from ENCODE, Roadmap Epigenomics, FANTOM5 [16] and other sources into our database for users to integrate these data into their analyses.

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.

3.4.1 Front-end: Web-based user interface

START provides a Web-based user interface at http://yiplab.cse.cuhk.edu.hk/start/ (Figure 3.7). 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

Note: (A) The main text box for entering STQL queries. (B) 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.

Figure 3.7: The user interface of START

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. Data files can be uploaded in a number of standard file formats, and multiple files can be uploaded at the same time in a zip package. 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.

3.4.2 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. It can be described in detail with three components.

Translation: In the back-end, we use Hadoop [2] 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 originally facilitated by Hive [57], 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. It does not require 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), ANTLR [1] is used in this step.

2) traversing the AST to create a query block (QB) and record necessary parsing information in the QB. AST is a tree data structure that can be traversed by a program. The program will fill in the QB when it is traversing the AST, finally the query statement was transformed into the filled data structure which can be understood by the computer.

3) interacting with the metastore to retrieve metadata of the involved signal tracks. In this step, the program will access the QB iteratively to do some basic checking such as whether the queried track exist, the queried interval atrribute exist.

4) generating a query plan in the form of a directed acyclic graph (DAG) of logical operations based on the QB. The program will access the QB iteratively again and generate corresponding operators according to the information collected in QB. For example, if the boolean variable hasFilter in QB which indicates there is a condition in the query statement, a Filter operator will be generated and added to the DAG. If hasGroupBy varialbe is true, a GroupBy operator will also be generated and added to the corresponding position of DAG.

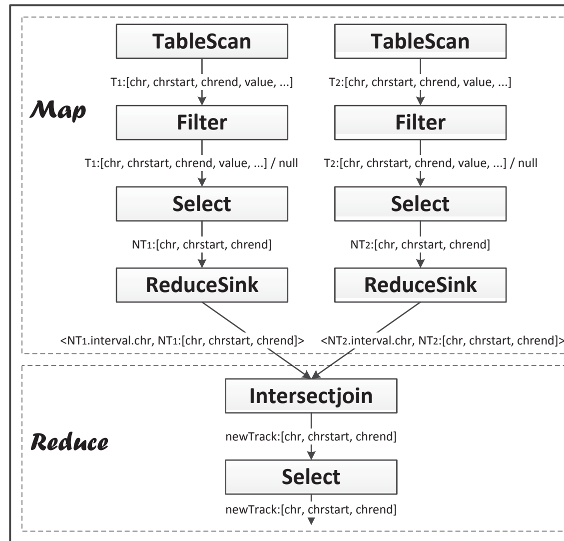


Figure 3.8: A typical MapReduce job created by the executor from an STQL query

Execution: The DAG of logical operations are then converted into executable jobs in Hadoop. Figure 3.8 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, such as having a value larger than 2 in this example. 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, which in this example include the chr, chrstart and chrend attributes, and calculates chrstart $+/-$ 200 and chrend $+/-$ 200. 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.

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.

3.4.3 Interface between front-end and back-end: Metastore

Here we describe the detail about metastore that bridges the front-end and the back-end. 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.

3.5 Results

3.5.1 Example queries

We used STQL to perform 14 representative analysis tasks. The full list of example queries and their detailed application scenarios are given in Appendix A. Here we show two queries as illustrative examples.

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

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

In the first example, the task is 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 high occupancy (HOT) regions [61].

The first sub-query demonstrates the use of FOR TRACK IN () 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 group neighboring genomic locations with the same number of overlapping transcription factor binding peak count into a region disjoint from other regions. These counts are computed by using the **vd_sum** operation with the **EACH MODEL** of interval values.

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

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

SELECT    *
FROM      Step1Results intersectjoin Step2Results;
```

In the second example, the task is to identify genomic regions with bi-directional transcription at their flanking regions, which could be potential enhancers producing enhancer RNAs (eRNAs) [24, 39]:

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. These regions are shifted 200bp to the left to 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. Each region in the final signal track 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 [24].

3.5.2 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 [47], Galaxy [32] 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.

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

Note: 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.

Table 3.3: Number of tokens involved between different systems

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 3.3).

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.

Query	START	Bedtools	Galaxy	Python
SQ1	207	407	N/A	1171
SQ2	50	135	N/A	184
SQ3	39	0.04	23	0.3
SQ4	47	21	408	42
SQ5	52	7	270	125
SQ6	46	0.04	N/A	21
SQ7	31	6	44	5
SQ8	33	2	30	3
CQ1	86	N/A	N/A	36
CQ2	300	N/A	N/A	7
CQ3	1340	N/A	N/A	84
CQ4	1680	262	N/A	420
CQ5	360	N/A	N/A	5289
CQ6	119	207	N/A	483

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

Table 3.4: Execution time between different systems

Table 3.4 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 3.4), 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.

3.5.3 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, to analyze some sequencing data using two different approaches. The data involved were DNA methylation data we produced by MBDCap-seq [37] 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 [34] 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 [51], FGF4 is related to HCC drug response [17], and HLX is involved in normal liver development [35]. 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 [38], while LRR1 is involved in cell growth control [59]. 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.

Chapter 4

Closest Interval Join Using MapReduce

An interval can be represented as `[s,e]`, where `s` and `e` respectively expresses the start and the end point of the interval, which also contains all the points between `s` and `e`. A number of real-world data can be modeled by interval such as the duration of a weather event and a segment of a DNA strand.

Figure 4.1 is an example about the closest interval join, suppose $R = \{r_1, r_2, r_3\}$ and $S = \{s_1, s_2, s_3, s_4, s_5\}$. For r_1, obviously it has a distance 0 with s_1 and s_2, so s_1 and s_2 are its closest intervals. For r_2, the distance between s_2 and r_2 is $9 - 6 = 3$, the distance between s_3 and r_2 is $13 - 11 = 2$, so s_3 is its closest interval. For r_3, its closest interval is s_4, because they intersect each other. So the final result of the *closest interval join* operation of R against S is $(r_1, s_1), (r_1, s_2), (r_2, s_3), (r_3, s_4)$.

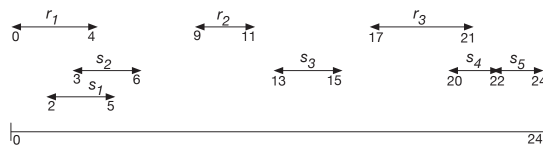


Figure 4.1: Closest interval join example

Closest interval join has many important applications. Take the bioinformatics field as the example, DNA has two paired strands which form the double helix structure. The smallest unit of the strand is one base, which can be represented by A, T, C and G. The base can be considered as an integer position. Therefore, a region of a DNA strand can be represented with an interval `[s,e]`, `s` and `e` represents the start and end base respectively. In order to understand the biological systems, biologists conduct a variety of experiments to find biological functions of different regions of DNA [65], these regions are called *genomic intervals* [9]. Take Figure 4.2 as the example, it illustrates a genomic interval, which can be represented as `[3,12]`.

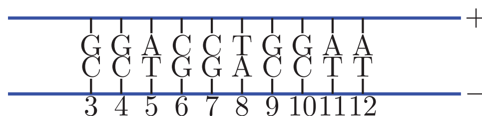


Figure 4.2: Basic DNA structure

In bioinformatics, SNP(single nucleotide polymorphism) [15] is a variation in a single base which may occur at some specific position, for instance, C is changed to T. SNP is actually a genomic interval with length 1. Related studies are usually performed to determine whether these SNPs affect other genes. Genes are also genomic intervals. In most cases, it is unclear which gene a SNP affects, and the first to check is often the gene closest to the SNP, so this task can be

converted to find closest intervals from gene intervals for a set of SNP intervals [58, 46, 36]. Figure 4.3 shows that bioinformatician want to find the closest genes for each SNP.

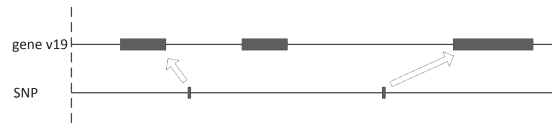


Figure 4.3: Find closest genes for SNPs

Interval data can be very large and continue to increase in size due to the advancement of data acquisition technology. For example, the 1000 Genome Project aims to establish an extensive catelogue of human genetic variation. This project needs to sequence the genomes of at least thousands of individuals from different populations, which will result in multiple massive raw big data files. The Cancer Genome Atlas(TCGA) is a project aiming at exploring the entire spectrum of genomic changes responsible for human cancer. It has collected genomic characterization of the tumor genomes from more than 11,000 cases regarding about 30 different cancer types. These experiments also produced a large number of files.

Due to the high availability of different types of genomic data from numerous individual genomes, it is now possible to run multiple analysis of many diverse genomic features simultaneously. This requires a new generation of convenient platform to process heterogeneous datasets as well as the state-of-the-art parallel computing strategies to achieve scalability and performance.

With about ten years development, MapReduce has already been the de facto standard for big data processing framework. A popular open source imple-

mentation is Hadoop [2], which has been adopted widely both in academic and industry. Consequently, considering how to store interval data and perform computation on MapReduce platform is a reasonable and interesting problem. To our best knowledge, we are the first to study *closest interval join* on MapReduce.

A typical MapReduce program mainly consists of a *Mapper* and a *Reducer* class. All the input and output are represented with $(key, value)$ pair for *Mapper* and *Reducer* to process. Users have to implement customized *map* function in *Mapper*, and *reduce* function in *Reducer*. The *map* function is applied in parallel to every input record from HDFS and emit a set of new $(key, value)$ pairs and write them into local disks as the intermediate results. After that, pairs with the same key from all *Mapper* output will be shuffled to a *Reducer* for the *reduce* function to process through the network. Finally, the newly emitted $(key, value)$ pairs are written to HDFS back as the final result.

In this chapter, we study *closest interval join* on MapReduce. The work most similar to our problem is *overlap interval join* between two interval sets using MapReduce, which has been discussed in literature [19]. Overlap interval join operation is often used to correlate intervals of different events. It means for each $r \in R$ and $s \in S$, the goal is to output all (r, s) pairs that r *overlaps* s. *overlaps* means for interval $[s_r, e_r]$ and $[s_s, e_s]$, if $s_r < s_s < e_r < e_s$, we say interval $[s_r, e_r]$ *overlaps* $[s_s, e_s]$. Take Figure 4.4 as the example, the output of overlapped interval pairs are $(r_1, s_1), (r_1, s_2)$ and (r_3, s_4) . In this example, R and S are stored in HDFS as input files, each interval is represented by a tuple, such as `[2,5]` for s_1.

The authors of [19] proposed a MapReduce algorithm to solve *overlap inter-*

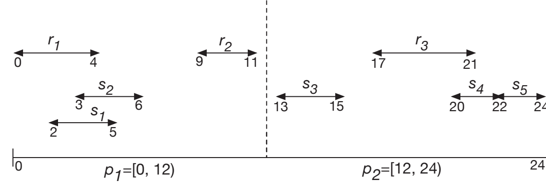


Figure 4.4: Overlap interval join example

val join. The main idea of [19] is to first specify a key set $\mathcal{K} = \{k_i, \ldots, k_j\}$, and assign each interval to one or multiple keys from the key set in the *map* function, then all the data with the same key are shuffled and processed in the same *reduce* function. There are three challenges here. **First**, how to choose the key set for a specific problem? Some choices may result in heavy skewness if many records are assigned to one key. In this chapter, we call the key set *partition scheme* for convenience. **Second**, how to map an input record to one or multiple keys, especially in the context of processing interval join on MapReduce platform? An inappropriate mapping mechanism for R and S may affect the correctness of the result. We call the mechanism *mapping rules*, which are implemented in *map* function. Take Figure 4.4 as the example, there are $R = \{r_1, r_2, r_3\}, S = \{s_1, s_2, s_3, s_4, s_5\}$, and an *overlap interval join* wants to output all (r, s) pairs that r *overlaps* s. [19] uses a set of non-overlapping ranges as the *partition scheme* $\mathcal{P}$. In the example, $\mathcal{P} = \{p_1, p_2\}$ with $p_1 = [0, 12), p_2 = [12, 24)$. In this simple example, one *mapping rule* in [19] is to assign each interval to the partition it is within, so finally r_1, r_2, s_1, s_2 are assigned to p_1, while r_3, s_3, s_4, s_5 are assigned to the key p_2. Then two *Reducer* instances are started to compute the results in parallel. However, the simple *partition scheme* together with the simple *mapping rules* cannot solve our *closest interval join* problem correctly. For example, in Figure 4.4, there is no chance for r_2 and s_3 to be in the same partition for comparison

according to the *mapping rules* in [19], but they form one result for *closest interval join*. **Third**, how to choose an efficient local algorithm for *reduce* function to process join problem.

In this chapter, we present a novel MapReduce algorithm for handling the *closest interval join*. Our algorithm is equipped with a better *partition scheme* and correct *mapping rules* to output all the closest pairs. Besides, we also designed an efficient local algorithm for the *reduce* function to find the closest intervals quickly. We conducted the experiments on both real and synthetic data. The results show that our method can outperform two baseline methods by several orders of magnitude.

The rest of the chapter is organized as follows. Section 4.1 reviews some related work. Section 4.2 describes three different solutions to handle this problem. A performance study based on real and synthetic data is conducted in Section 4.3.

4.1 Related Work

Interval data is ubiquitous in many fields. Driven by the demand for support of interval related queries and joins, many research work have been done to embed interval related operations into current data-processing systems. Related work includes efforts to process overlap interval join on MapReduce(Section 4.1.1), to compute kNN join(Section 4.1.2) or similarity join(Section 4.1.3) for N-dimension objects on MapReduce, and centralized algorithms to process interval joins(Section 4.1.4).

4.1.1 Overlap Interval Join on MapReduce

There are several works about processing overlap interval joins using MapReduce directly. [19] has developed novel algorithms to optimize 2-way and multi-way overlap interval joins. These presented methods are expected to be integrated with already implemented spatio-temporal data processing system such as *Spatial-Hadoop* [29], *Sci-Hadoop* [18] and *CloST* [54]. Consider a kind of spatial-temporal environment modeling data, interval `[s,e]` can model the duration of an event such as a *rainfall*, *high wind speed*, *high temperature* and *high pollutant concentration* observed, `s` and `e` represents the start and end time respectively. For example, `[7 AM, 7:15 AM]` can indicate that a *high wind speed* was observed during the period of 7 AM to 7:15 AM.

Suppose there has already been two sets of intervals R and S. R is the set of intervals which indicates all the *high wind speed* events, S indicates all the *high temperature* events. If all the event pairs that *high temperature* starts during the period of a *high wind speed* and ends after it are required to output, R *overlaps* S can be used to find all the pairs. This is an overlap interval join application.

Look at Figure 4.5, there are interval sets $R = \{r_1, r_2, r_3, r_4\}$ and $S = \{s_1, s_2, s_3, s_4\}$, for each $r \in R$ and $s \in S$, the goal is to output all pairs (r, s) if r *overlaps* s. [19] proposed a MapReduce algorithm to solve this problem. The idea can be described in the following three parts.

partition scheme: All the intervals of the two sets lie within a range, for example, in Figure 4.5, all the intervals are in the range $[t_0, t_3)$. Suppose there are three continuous ranges here, which are $[t_0, t_1), [t_1, t_2), [t_2, t_3)$, and $\mathcal{P} = \{p_1, p_2, p_3\}$ is a *partition scheme* with $p_1 = [t_0, t_1), p_2 = [t_1, t_2), p_3 = [t_2, t_3)$.

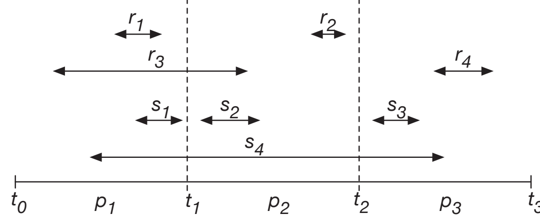


Figure 4.5: Overlap interval join in detail

Mapper: The *partition scheme* is loaded into main memory in each *Mapper*, at least two $Mapper$ instances are started to read R and S in parallel. Each $Mapper$ will read an interval I one by one, and check which partition it belongs to according to some *mapping rules*, this logic is finished in the *map* function. Then the *Mapper* outputs intermediate results in the format of $(pid, (label, I))$. *pid* means the partition id, for example, it may be p_1, p_2, or p_3. p_1, p_2 and p_3 can also be considered as the reducer id, because each partition corresponds to a reducer. *label* means which set the interval is from, it may be R or S, which can be represented with one byte in practical implementation. I is the interval object.

Now let us discuss more about the *mapping rules* used in [19]. **The first rule:** if an interval is totally within a partition, it will be sent to the that partition. Look at Figure 4.5, it is easy to decide *pid* for intervals such as $r_1, r_2, r_4, s_1, s_2, s_3$, because they are only within one partition, so they are sent to the corresponding partition seperately. However, careful considerations are required for interval r_3 and s_4, because both of them intersect multiple partitions. If they are sent to all of these partitions, there will be duplicate results in reducers, the same result pair will be output by different reducers. To solve this, [19] proposed **the second rule:** this kind of interval from R is sent to all

partitions it intersects, while this kind of interval from S will be sent to only the first partition it intersects. So r_3 is sent to partition p_1 and p_2 while s_4 is sent to only p_1. Table 4.1 shows an example of the intermediate results output by each mapper.

mapper1:	$(p_1, (R, r_1))$	$(p_2, (R, r_2))$	$(p_1, (R, r_3))$
	$(p_2, (R, r_3))$	$(p_3, (R, r_4))$	
mapper2:	$(p_1, (S, s_1))$	$(p_2, (S, s_2))$	$(p_3, (S, s_3))$
	$(p_1, (S, s_4))$		

Table 4.1: Intermediate results output by each mapper

The reason is if r_3 is not sent to all the partitions it intersects, there will be some results missed, for example, (r_3, s_2) will be missed if r_3 is not sent to p_2. On the other hand, this kind of interval from S does not need to be sent to all partitions it intersects. Take s_4 as the example, if there exists r *overlaps* s_4, then the start point of r must be smaller than the start point of s_4 according to the definition of *overlaps*, so s_4 just needs to be sent to the first partition it intersects.

Reducer: After map phase, all key-value pairs are shuffled to corresponding reducers according to their keys. Table 4.2 shows what intervals each reducer finally received. Then the computation will be performed in each reducer in parallel, the final result is the union of all the reducer output, which are (r_1, s_1), (r_3, s_2) and (r_3, s_4).

reducer	intervals	output
p_1	r_1, r_3, s_1, s_4	(r_1, s_1), (r_3, s_4)
p_2	r_2, r_3, s_2	(r_3, s_2)
p_3	r_4, s_3	

Table 4.2: Intervals received in each reducer

This example shows that though a *partition scheme* is defined, it still requires to define *mapping rules* to decide which partitions an interval should go, and it depends on the specific properties of a problem in order to output the correct result.

The *partition scheme* and *mapping rules* defined for the *overlap interval join* cannot handle the *closest interval join* directly. In the example above, (r_2, s_3) have no chance to meet each other in the same reducer for computation, but it is a result for *closest interval join*.

4.1.2 kNN join on MapReduce

kNN join on MapReduce is formulated as: for each object r in R, find its k nearest neighbors in set S. *Closest interval join* can be considered as a case of $1NN$ join problem. We introduce two different MapReduce algorithms here.

The first method is from [63]. It is similar to Cartesian product, which means there is a chance for $\forall r \in R$ and $\forall s \in S$ to compare with each other. This method divides the whole Cartesian product job into several independent tasks so they can be executed on MapReduce in parallel.

The method consists of two MapReduce jobs. The map phase of the first job divides R and S into n equally-sized blocks randomly, such as $R_1, R_2 ... R_n$ and $S_1, S_2 ... S_n$. Then each $R_i S_j$ combination will be sent to a reducer, so the number of reducers needed is n^2. To achieve this, each interval is emitted n times in order for it to have the chance to compute with those n blocks from another set. The reduce phase is then started to compute local kNN result for each r. Suppose there are $R = \{r_1, r_2, r_3, r_4, r_5, r_6\}, S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$.

Let $n = 2$, which means there will be $n^2 = 4$ reducers, say they are p_1, p_2, p_3, p_4. Let $R_1 = \{r_1, r_2, r_3\}, R_2 = \{r_4, r_5, r_6\}, S_1 = \{s_1, s_2, s_3\}, S_2 = \{s_4, s_5, s_6\}$. Figure 4.6 shows that the *map* function emits two key-value pairs for each input interval in this example.

After the map phase, intermediate results with the same key will be shuffled to the same reducer. Table 4.3 shows the blocks each reducer receives. It demon-

Reducer	p_1	p_2	p_3	p_4
Blocks	R_1, S_1	R_1, S_2	R_2, S_1	R_2, S_2

Table 4.3: The blocks each reducer receives

strates that each block from R has a chance to compare against each block from S. The output of each reducer is in the format of $(r, (s, |r, s|))$, $|r, s|$ means the distance between r and s. Then the kNN computation is performed locally in each reducer. After local kNN computation per reducer, another MapReduce job is used to find the global kNN result for each object $r \in R$. The map phase of the second job is to read the output of the first job and emit them to corresponding reducers using r as the key, each reducer will find global kNN result for every r and output them.

The problem of this method is the huge shuffling traffic incurred in the two jobs. In the first job, as each block has to be shuffled to n reducers to compare with n blocks from another, the total data shuffled is $O((|R| + |S|) \cdot n)$. After the first MapReduce job, there will be roughly k results for an object r among n reducers, because each *Reducer* will compute a kNN result for the object r. So in the second MapReduce job, each r and its k results will be shuffled to the same *Reducer* from n output files, therefore the total data shuffled is $O(|R|nk)$.

$$r_1 \rightarrow \begin{cases} (p_1, (R, r_1)) \\ (p_2, (R, r_1)) \end{cases} \quad r_2 \rightarrow \begin{cases} (p_1, (R, r_2)) \\ (p_2, (R, r_2)) \end{cases} \quad r_3 \rightarrow \begin{cases} (p_1, (R, r_3)) \\ (p_2, (R, r_3)) \end{cases}$$

$$r_4 \rightarrow \begin{cases} (p_3, (R, r_4)) \\ (p_4, (R, r_4)) \end{cases} \quad r_5 \rightarrow \begin{cases} (p_3, (R, r_5)) \\ (p_4, (R, r_5)) \end{cases} \quad r_6 \rightarrow \begin{cases} (p_3, (R, r_6)) \\ (p_4, (R, r_6)) \end{cases}$$

$$s_1 \rightarrow \begin{cases} (p_1, (S, s_1)) \\ (p_3, (S, s_1)) \end{cases} \quad s_2 \rightarrow \begin{cases} (p_1, (S, s_2)) \\ (p_3, (S, s_2)) \end{cases} \quad s_3 \rightarrow \begin{cases} (p_1, (S, s_3)) \\ (p_3, (S, s_3)) \end{cases}$$

$$s_4 \rightarrow \begin{cases} (p_2, (S, s_4)) \\ (p_4, (S, s_4)) \end{cases} \quad s_5 \rightarrow \begin{cases} (p_2, (S, s_5)) \\ (p_4, (S, s_5)) \end{cases} \quad s_6 \rightarrow \begin{cases} (p_2, (S, s_6)) \\ (p_4, (S, s_6)) \end{cases}$$

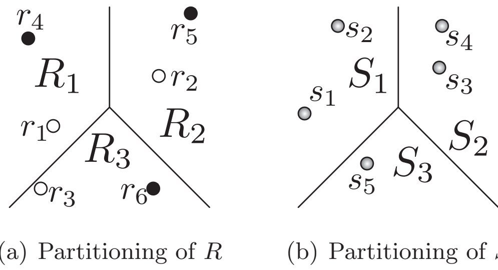
Figure 4.6: Emitted intermediate results by mappers

The analysis demonstrated the total data shuffled depends on $|R|$, $|S|$, n and k, and it is almost proportional to each of them. So a large amount of data will be shuffled during the two jobs, and the experiments from our experiment part have also shown that.

Authors in [43] proposed a more complicated MapReduce algorithm to efficiently compute kNN result, which emits much smaller size of intermediate results as well as much lower shuffling cost. The main idea is to first partition R into disjoint subsets $R_1, \ldots, R_n$ based on *a set of pivot objects* $\mathcal{P} = \{p_1, \ldots, p_n\}$ from R. Then for each R_i the algorithm will find a subset $S_i^{'}$ from S that guarantees all kNN results for $\forall r \in R_i$ exist in $S_i^{'}$, then R_i and $S_i^{'}$ will be shuffled to the same reducer for computing and the correct kNN results are output. To find $S_i^{'}$ correctly, some advanced bounding and pruning techniques are used. However, these techniques are based on triangle inequality, which cannot apply to interval data directly, we will explain it later.

Job1

Suppose a set of pivot objects $\{p_1, \ldots, p_n\}$ from R are selected. The map phase of this job will partition R and S by assigning each object to its closest pivot. Figure 4.7 is an example to illustrate the partitioning process. Suppose there are $R = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ and $S = \{s_1, s_2, s_3, s_4, s_5\}$, $k = 2$. Suppose r_4, r_5, r_6 are selected from R as the pivot objects. Now R and S are emitted into disjoint partitions $R_1, R_2, R_3, S_1, S_2, S_3$ by $\{r_4, r_5, r_6\}$. At the same time, the *Mapper* also collects statistics for each partition. The statistics includes: the number of objects N in the partition, the minimum distance $L(R_i)$(resp. $L(S_i)$) and the maximum distance $U(R_i)$(resp. $U(S_i)$) from an object in partition R_i(resp. S_i) to the pivot p_i. In addition to that, for each S_i, the statistics also includes the first k smallest distances between the objects in S_i and the pivot p_i. There is no reduce phase in this job.



(a) Partitioning of R (b) Partitioning of S

Figure 4.7: The partitioning of R and S

Job2

The first MapReduce job has assigned each object a *pid* to indicate which reducer it should go. For example, r_1, r_4, s_1, s_2 should go to reducer P_1. Now there is another problem arose, the set $S_1 = \{s_1, s_2\}$ may not contain all the

kNN results for $R_1 = \{r_1, r_4\}$. There may exist objects from other S_j that also become the kNN result for objects in R_1. For example, from Figures 4.7(a) and 4.7(b), s_5 is the closest object to r_1, but it does not belong to S_1. This is what the map phase of this job will resolve. It will find a correct set S_i' from $S_1, \dots, S_n$ to make sure all kNN results for R_i are in S_i'. In this example, $S_1' = \{s_1, s_2, s_5\}$, it includes objects from S_1 and S_3. Actually, the map phase will inspect every s from all partitions $S_1, \dots, S_n$ to decide whether it should be a possible kNN result for partition R_i.

Bounding and Pruning Techniques: To decide whether an object s is a possible result for some r of R_i, [43] first developed the upper and lower bound of the distance between s and r. Suppose $s \in S_j$ and $r \in R_i$, $|r, s|$ denotes the distance between r and s, $ub(s, R_i)$ denotes the upper bound of $|r, s|$, and $lb(s, R_i)$ denotes the lower bound of $|r, s|$, which means

$$lb(s, R_i) \leqslant |r, s| \leqslant ub(s, R_i) \tag{4.1}$$

According to the collected statistics, the algorithm can reason a threshold θ_i for partition R_i, which means that for all kNN result of R_i, the distance $|r, s|$ should not be greater than θ_i. Therefore the *pruning rule* is: if $lb(s, R_i) \leqslant \theta_i$, it may be a possible result of R_i, so it will be sent to reducer P_i; else, it will be pruned. Obtaining $lb(s, R_i)$, $ub(s, R_i)$ and θ_i will use the triangle inequality frequently. Look at Figure 4.8, according to triangle inequality, it is easy to obtain that $|r_1, s_5| \geqslant |r_4, s_5| - |r_4, r_1|$, $|r_4, r_1| \leqslant U(R_1)$, $|r_4, s_5| \geqslant |r_4, r_6| - |r_6, s_5|$, so finally $|r_1, s_5| \geqslant |r_4, r_6| - U(R_i) - |r_6, s_5| = lb(s_5, R_1)$. Then this $lb(s_5, R_1)$ is compared with θ_1 according to the previous *pruning rule* to decide whether s_5

should be contained in $S_1^{'}$.

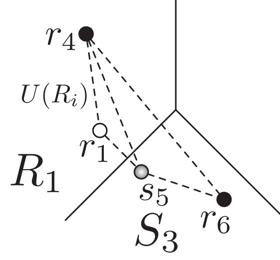


Figure 4.8: The bounding and pruning techniques

However, the triangle inequality cannot hold for interval data. Take Figure 4.9 as the example. Obviously that $|r_1, s_5| \leqslant |r_4, r_6| - U(R_i) - |r_6, s_5|$, which violates the triangle inequality. As there is no way to compute the $lb(s_5, R_1)$ for interval data, that *pruning rule* cannot apply. The essential reason is that the interval distance defined in our problem actually is affected by the interval's internal length, but distance between N-dimentional objects are usually defined independent of the objects' extent. That's why the bounding and pruning techniques cannot be adopted to solve our problem directly.

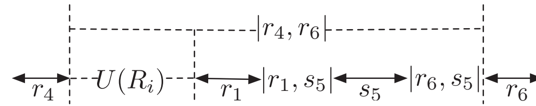


Figure 4.9: Interval data cannot apply triangle inequality

4.1.3 Similarity Join on MapReduce

Similarity join [50] can be defined as: given one object set R, distance threshold θ and the distance function, for all $r \in R$, the goal is to output all the pairs (r_1, r_2) which satisfies $|r_1, r_2| \leqslant \theta$. The difference between the problem and

the *closest interval join* is that similarity join has a fixed threshold while *closest interval join* does not have. More specifically, in *closest interval join*, even an interval is very far away(way beyond a threshold), it can still be a result if it is the closest interval of another one. The bounding and pruning techniques developed in similarity join also depend on the triangle inequality property, which cannot apply to interval data directly as well.

4.1.4 Centralized Overlap Interval Join Algorithms

Traditional centralized algorithms for *overlap interval join* [33, 53, 42, 28, 31, 49, 40, 25] can be categoried into three classes: *Sort-Merge Algorithms*, *Partition-Based Algorithms* and *Index-Based Algorithms*.

Sort-Merge Algorithms consist of two phases. The first phase is to sort the two sets R and S according to the start or end points of the interval. The second phase is to scan R and S simultaneously and merge the intervals which overlap with each other. The Sort-Merge Algorithms can be considered as an improvement of Nested-Loop Algorithms by using the order property so as to prune those interval pairs that obviously cannot output the result. [33] proposed a sort-merge version called TJ-1, which sorts R and S based on an ascending primary order on the start point and an ascending secondary order on the end point. Then it scans sorted R in order, and maintains a special pointer for S. This pointer is explicitly incremented under certain conditions as the algorithm proceeds. Scanning S is always started from this special pointer instead of the beginning of S, so less records have to be read during the scan of S in every iteration.

Partition-Based Algorithms consist of two phases. The first phase is to partition interval sets R and S, for instance, R and S are partitioned into $R_1, \ldots, R_n$ and $S_1, \ldots, S_n$. The second phase is to load the combination R_iS_i into memory and perform the join. How to decide the partition interval boundary to make sure each partition obtains rough amounts of intervals is an important problem. [53] uses histograms to count the number of intervals falling in each partition and adjust the partition boundaries so as to guarantee each partition has nearly the same amount of intervals. [42] presents another partition based algorithm for centralized overlap interval join. In this algorithm, an interval is mapped to a point in a two-dimentional space where x-coordinate represents the interval start point and y-coordiate represents the interval lengh. Then the space is partitioned into regions. During the join phase, a region of R has to be compared with multiple regions of another interval set. The two-dimensional space makes finding the corresponding regions that may overlap a region intuitive. [28] proposed a new partition strategy in order to make sure the overlap interval join operation is independent of short- and long-lived intervals. So this method is still efficient with long-lived intervals.

Indexed-Based Algorithms are to build index for S, and perform the query for each interval of R. The index could be preexisting or built on the fly. It could be: quadtree [31], loose quadtree [49], relational interval tree [40], or segment tree [25].

The works described above are centralized algorithms for overlap join but not for closest join and the primary optimization goal is to reduce the number of disk I/Os. However, this is different from a MapReduce algorithm, because the goal of a MapReduce algorithm is to reduce the size of intermediate results to

decrease the network shuffling cost. In addition to that, these algorithms except Nested-Loop cannot be used to compute the closest interval join result unless with substantial modification.

4.1.5 Summary

Compared with above-related work, our closest interval join problem is indeed different. The strategies for overlap interval join cannot be applied to our problem directly because the overlap is just one case in our problem setting. More cases need to be considered to solve our problem correctly. kNN and similarity join work exploit triangle inequality to prune as much as possible computation, however, it is only correct for multi-dimensional objects. Other centralized algorithms focus on reducing the number of disk I/Os, which are different from a MapReduce algorithm. In a MapReduce environment, the bottleneck is usually the huge intermediate data that need to be shuffled, so the goal is to reduce this. In addition to that, centralized algorithms also only consider overlap interval join instead of closest interval join problem.

We have derived two algorithms from existing literature which focus on other problems originally. But we found that there is still a significant improvement if tailored for closest interval join problem. The experiments also demonstrate it.

4.2 The Solutions

In this section, we propose three algorithms called *Broadcast Algorithm(BA)* in Section 4.2.1, *Neighbor Replicating Algorithm(NRA)* in Section 4.2.2, and

Distance-Aware Algorithm(DAA) in Section 4.2.3.

BA is going to divide R and S randomly into partitions $R_1, \ldots, R_n$ and $S_1, \ldots, S_n$, and broadcast every R_i to all the partitions of S to compute the result. So each reducer is responsible for computing a combination of R_i and S_j. No pruning techniques are used. NRA also tries to partition R and S, but only sends R_i to possible partitions of S to compute the result. However, R_i will be sent to some additional partitions of S that cannot produce results for it. Compared with NRA, DAA is more efficient by developing better partition scheme and mapping rules to send R_i to even fewer partitions for computation without missing any result. In addition to that, we also propose an efficient local algorithm to compute the closest interval join used by each reducer.

We will use $|r, s|$ to represent the distance between interval $r = [s_r, e_r]$ and $s = [s_s, e_s]$, formal definition of $|r, s|$ is:

$$|r, s| = \begin{cases} 0, & \text{if } r \text{ intersects } s \\ \min(|s_s - e_r|, |s_r - e_s|), & \text{otherwise} \end{cases}$$
$$\text{where } r \text{ intersects } s \text{ iff } s_r \leqslant e_s \wedge s_s \leqslant e_r$$

4.2.1 Broadcast Algorithm(BA)

The broadcast algorithm is based on the kNN join MR algorithm [63] that we described in Section 4.1.2. The main idea is to first divide R and S into $R_1, \ldots, R_n$ and $S_1, \ldots, S_n$ randomly. Then all R_iS_j can be computed in parallel, so there are n^2 reducers. As there are n output files for each $r \in R_i$, these files are sent to the same machine to compute the final result for each r. When

implemented in MR algorithm, it contains two MR jobs. The *Mapper* of the first job is to read each interval one by one and emit n key-value pairs for it. Then all the intermediate results are broadcasted to different *Reducer*s for computing local result for $R_i S_j$. The *Mapper* of the second job is to read the result output by the first job, and emit them with $r \in R$ as the key, so result with the same r are shuffled to the same reducer for computing the final result for r.

4.2.2 Neighbor Replicating Algorithm(NRA)

This algorithm is based on the overlap interval join MR algorithm [19] that we discussed in Section 4.1.1, where each interval will be sent to multiple partitions according to a partition scheme. A simple range partition scheme may output wrong answer if we simply send r to the partition that it intersects. To fix that, an intuitive idea is to send r not only to the partition p_i that it intersects, but also its *neighbor partitions*, namely p_{i-1} and p_{i+1}. For example, in Figure 4.10, instead of sending r_2 to p_3 only, we also send r_2 to p_2. However, this works only when the partition scheme is properly chosen. Take Figure 4.10 as the example, the domain is equally divided into three partitions. Because of the skew distribution, S has no answer intervals in p_2 and p_3. So r_2 will only be sent to p_2 and p_3, while its closest interval s_3 is in p_1. Therefore, to guarantee that NRA is correct, we need to carefully choose the partition scheme such that *each partition contains at least one interval.* To do that, we borrow idea from [27] to perform sampling (e.g., 1%) to build a partition scheme such that each partition contains some answer intervals. For example, we can use $\mathcal{P} = \{p_1 = [0, 4), p_2 = [4, 8), p_3 = [8, 36)\}$ as the partition scheme.

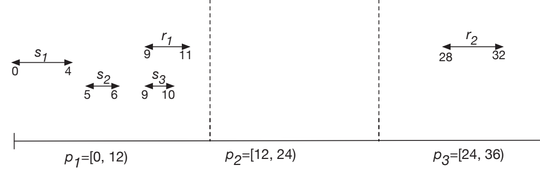


Figure 4.10: An example about NRA

We can now use three MapReduce jobs to find out all closest pairs.

The first job is to decide the partition scheme by sampling. Suppose the total range for R and S is $[t_0, t_n)$, now we want to get a partition scheme $\mathcal{P} = \{p_1 = [t_0, t_1), p_2 = [t_1, t_2), \ldots, p_n = [t_{n-1}, t_n)\}$ to make sure every partition has at least one $s \in S$ fall in. Algorithm 1 explains this job. The map phase is to sample some intervals from S. Then all the sampled intervals are shuffled to one reducer. The reducer first sorts the intervals by the start point on ascending order. Then it divides the data into a partition for every k elements, so we use the start point of the first element of each partition as the partition boundary (Line 11 in REDUCER of Algorithm 1).

The second job is to partition R and S based on the partition scheme obtained by the sampling job, and then compute the closest pairs within each reducer. Specifically, each $r \in R$ will be sent to all partitions it intersects as well as the neighbor partitions, while each $s \in S$ will be sent to only those partitions it intersects in the map phase. Algorithm 2 shows the pseudocode for this *Mapper*.

The reduce phase is the same as the first job of BA so that each reducer employs a local algorithm to compute the result for a combination of R_iS_i. Since the NRA replicates r more than one time, the closest pair obtained in each partition could be a local optimum. So, we need **the third job** to find out the

global optimum for each r. This is the same as the last job of BA that the map phase sends all the local optimum with the same r to a reducer and the reducer finds the global optimum for that r.

Algorithm 1 The sampling job

```
 1: class MAPPER
 2:     method SETUP
 3:         sampleRatio ← specified by user
 4:     end method
 5:     method MAP(interval I)
 6:         generate random float ratio ∈ [0, 1]
 7:         if ratio < sampleRatio then
 8:             EMIT(1, I)
 9:     end method
10: end class
 1: class REDUCER
 2:     method SETUP
 3:         n ← user specified                              ▷ Number of partitions
 4:     end method
 5:     method REDUCE(1, list of interval I)
 6:         S ← list of I                                   ▷ suppose S starts from S[0]
 7:         total ← the size of S
 8:         k = total/n
 9:         sort S by start point on ascending order
10:         for all j ∈ {1, . . . , n − 1} do
11:             t_j ← the start point of interval S[j ∗ k − 1]
12:             EMIT(null, t_j)
13:         end for
14:     end method
15: end class
```

4.2.3 Distance-Aware Algorithm(DAA)

Although NRA can reduce a large amount of replications of R and S compared with BA, it still has a lot of unnecessary replications. For instance, NRA needs to replicate r_3 to p_1 in Figure 4.11, but r_3's closest interval is in p_2 indeed. Recall that NRA needs three MR jobs. The first job is to obtain the partition scheme by sampling. The second job is to partition R and S according to the

partition scheme in map phase and compute local result in each reduce phase. The third job is to find global closest intervals for each $r \in R$. The distance-aware algorithm(DAA) is going to modify the second and the third job of NRA. Instead of partitioning R and S at the same time, DAA only partitions S and obtains the new partition boundary in the second job. Then the map phase of the third job partitions R according to the new partition scheme and computes the result in reduce phase.

Algorithm 2 The map phase of the second job of NRA

```
1: class Mapper
2:     method Setup
3:         label ← R or S
4:         load partition scheme file into memory
5:         obtain partition mapping {p_1 = [t_0, t_1), ..., p_n = [t_{n-1}, t_n)} from partition scheme
   file
6:     end method
7:     method Map(interval I)
8:         i, j ← index of continuous partitions {p_i, ..., p_j} that I intersects
9:         for all k ∈ {i, ..., j} do
10:            Emit(k, (label, I))
11:        end for
12:        if label is R then
13:            if i > 1 then
14:                Emit(i − 1, (label, I))
15:            if j < n then
16:                Emit(j + 1, (label, I))
17:    end method
18: end class
```

4.2.3.1 Algorithm Description

Still take Figure 4.11 as the example.

The first job is to do sampling on S, and the partition scheme obtained is $\mathcal{P} = \{p_1 = \texttt{[0,12)}, p_2 = \texttt{[12,24)}\}$. Then the second job partitions S according to the partition scheme, the results are: partition p_1 contains s_1 while p_2 contains

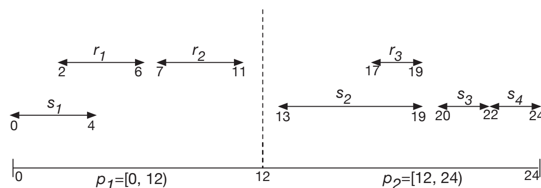


Figure 4.11: Motivating example

$\{s_2, s_3, s_4\}$. The key of the second job is its reduce phase, which *adjusts the partition scheme*. For example, p_1 contains s_1 = [0,4], so p_1 is adjusted into p'_1 = [0,4). p_2 contains $\{s_2, s_3, s_4\}$, so it finds the leftmost start point and the rightmost end point of these intervals and use them as the new partition boundary, so p_2 is adjusted into p'_2 = [13,24). Then the third job partitions R according to the adjusted partition scheme. The mapping rule is: if r intersects multiple partitions, it is sent to those partitions; if it does not intersect any partition, it is sent to the partition closest to it.

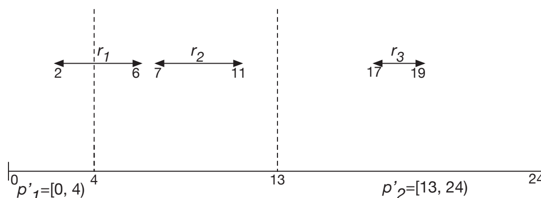


Figure 4.12: The new boundary for each partition

Look at Figure 4.12, the adjusted new partition scheme is $\mathcal{P} = \{p'_1$ = [0,4), p'_2 = [13,24)$\}$. Now consider r_1, as it intersects p'_1 but is far away from p'_2, it is only sent to p'_1. For r_2, it does not intersect any partition, so we compute the distance between it and its neighbor partitions, then we have $|r_2, p'_1| = 7 - 4 = 3$, and $|r_2, p'_2| = 13 - 11 = 2$. So, unlike NRA that sends r_2 to both p'_1 and p'_2, DAA sends r_2 only to p'_2. For r_3, it will be sent only to p'_2, be-

cause r_3 only intersects p'_2. That's why this algorithm is called distance-aware, because it considers the distance information between r and each partition p'_i to partition R smartly. The reduce phase of the third job performs the closest interval join within each partition and output the result.

However, there could be the situation that some intervals from S intersect more than one partitions, which requires special care to guarantee the correctness of DAA.

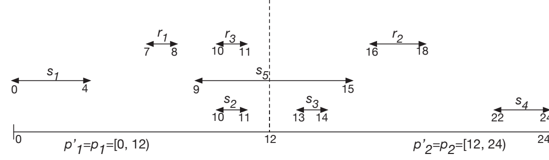


Figure 4.13: An example shows some intervals intersect more than one partition

Take Figure 4.13 as the example, suppose the initial partition scheme obtained by the first sampling job is $\mathcal{P} = \{p_1 = \texttt{[0,12)}, p_2 = \texttt{[12,24)}\}$. The second job partitions S according to the partition scheme. It is easy to handle s_1, s_2, s_3, s_4, because they are only within one partition and will be sent to that partition accordingly. But s_5 intersects two partitions. We have two options to handle s_5 here: (o_1) sending s_5 to **either** p_1 or p_2; (o_2) sending s_5 to **both** p_1 and p_2. The first option (o_1) will give us the wrong query answer. For example, if s_5 is sent to p_1, then p_1 contains $\{s_1, s_2, s_5\}$, while p_2 contains $\{s_3, s_4\}$. After partition boundary adjustment, $p'_1 = \texttt{[0,15)}$, $p'_2 = \texttt{[13,24)}$. Using the adjusted partition scheme to partition R, when considering $r_2 = \texttt{[16,18)}$, we will send r_2 to p'_2. However, this is wrong, since the actual closest interval to r_2 is s_5 which is not in p'_2. Similarly, the mapping rules will output wrong answer for r_1 if we send s_5 to p_2 only.

So, for this special case, our mapping rule uses option (o_2). For example, after sending s_5 to p_1 and p_2, we get adjusted partitions p'_1 = [0,15) and p'_2 = [9,24). Both r_1 and r_2 can find their closest answer intervals if we send them to their closest adjusted partition, namely, sending r_1 to p'_1, and sending r_2 to p'_2.

Note that the adjusted partitions p'_1 = [0,15) and p'_2 = [9,24) intersect each other, therefore it makes r_3 = [10,11] fall into both partitions. To eliminate this redundancy, we further amend our partition adjustment method: suppose the boundary of the old partition p is $[left, right)$, the leftmost start point and the rightmost end point of all intervals falling in that partition is lm and rm, if $lm > left$, we adjust $left$ to lm; if $rm < right$, we adjust $right$ to rm. If both of the two conditions cannot be satisfied, we will not adjust the partition. For example, in Figure 4.13, after replicating s_5 to p_1 and p_2, partition p_1 contains $\{s_1, s_2, s_5\}$, so $rm = 15$ which is greater than the old partition boundary 12, so we don't adjust it. Based on the same reason, we don't adjust p_2 either.

The resulting adjusted partitions are still p'_1 = [0,12) and p'_2 = [12,24). According to these adjusted partitions, r_3 is sent to only p'_1. The new partition scheme is still correct for all the other query intervals.

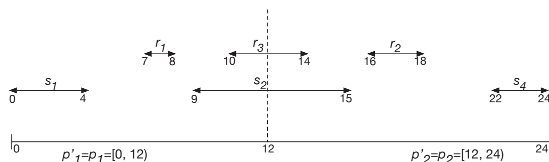


Figure 4.14: An example shows duplicate results

Duplicate Removing There is one remaining issue in DAA. That is, it may

create duplicate results. Figure 4.14 shows an example. According to our mapping rules, r_3 and s_2 are sent to partition p_1 as well as p_2. So the result pair (r_3, s_2) will appear in both p_1 and p_2. To remove the duplication, we first find the common part of the result pair. For example, the common part of the result pair (r_3, s_2) is `[10,14]`, then we check which partition the start point of the common part falls in, and we only keep the result pair in that partition and discard other duplicate ones. As the start point of the common part of (r_3, s_2) is 10, and it falls in partition p_1, so we output the result pair (r_3, s_2) in p_1 but discard duplicates in other partitions.

Job implementation The sampling job is the same as NRA's. The second job is to partition S and adjust the partition scheme as Algorithm 3 shows. Algorithm 4 shows the third job of DAA, which partitions R according to the adjusted partition scheme and compute the final result.

4.2.3.2 Finding the Closest Interval Pairs within a Reducer

In Algorithm 4, the reduce phase has to employ an algorithm to compute the closest interval join for R_iS_i. As there is no efficient centralized algorithm for closest interval join, we design an efficient algorithm to find the closest interval pairs within a single reducer.

Let $R_i = \{r_1, \cdots, r_m\}$ and $S_i = \{s_1, \cdots, s_n\}$ be the set of intervals that are sent to partition p'_i. One algorithm to find out the closest pairs is the simple nested loop one, where R_i is put in the outer loop and S_i is put in the inner loop. For each $r \in R_i$, it will scan the whole set of S_i to obtain the closest interval(s). Let M be the size of R_i and N be the size of S_i. The time complexity of such

Algorithm 3 The adjustment job

```
1: class Mapper
2:     method Setup
3:         load partition scheme file into memory
4:         obtain partition mapping {p_1 = [t_0, t_1), ..., p_n = [t_{n-1}, t_n)} from partition scheme file
5:     end method
6:     method Map(interval I)
7:         i, j ← index of continuous partitions {p_i, ..., p_j} that I intersects
8:         for all k ∈ {i, ..., j} do
9:             Emit(k, I)
10:        end for
11:    end method
12: end class
1: class Reducer
2:     method Setup
3:         load partition scheme file into memory
4:         obtain partition mapping {p_1 = [t_0, t_1), ..., p_n = [t_{n-1}, t_n)} from partition scheme file
5:     end method
6:     method Reduce(k, list of interval I)
7:         start, end ← look up k in partition mapping
8:         min ← find minimum start point from list of interval I
9:         max ← find maximum end point from list of interval I
10:        if min > start then start = min
11:        if max < end then end = max
12:        update p_k = [start, end) to partition scheme file
13:        emit all intervals to file
14:    end method
15: end class
```

algorithm is $O(M \times N)$.

The problem of the simple nested loop algorithm is that it needs to repeatedly scan S_i. We want to improve this nested loop algorithm by skipping some s when scanning S_i in the inner loop.

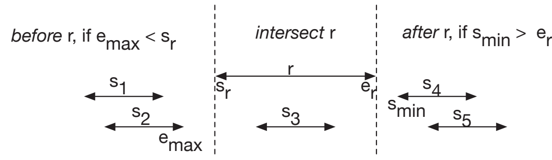


Figure 4.15: Three different regions in terms of a query interval

Algorithm 4 The last job of DAA

```
 1: class Mapper
 2:     method Setup
 3:         load adjusted partition scheme file into memory
 4:         obtain partition mapping {p_1, ..., p_n} from partition scheme file
 5:     end method
 6:     method Map(interval I)
 7:         i, j ← index of continuous partitions {p_i, ..., p_j} that closest to I
 8:         for all k ∈ {i, ..., j} do
 9:             Emit(k, I)
10:         end for
11:     end method
12: end class
 1: class Reducer
 2:     method Reduce(k, list of interval I)
 3:         setR ← list of interval I
 4:         construct setS by reading files for partition p_k
 5:         FindClosestAndEmit(setR, setS)              ▷ refer to Algorithm 5
 6:     end method
 7: end class
```

Given a query interval $r = [s_r, e_r]$, the whole domain of S_i can be divided into 3 regions, which are described as follows: (i) the first region is the one in which the largest end point is smaller than s_r. (ii) the second region is the one in which all the intervals intersect r. (iii) the third region is the one in which the smallest start point is greater than e_r. We say interval of S_i in region (i) is *before*

r, and the interval in region (iii) is *after* r. Figure 4.15 shows us an example about these regions. e_{max} is the largest end point in the first region, s_{min} is the smallest start point in the third region. In this figure, we can find that if there are answer intervals intersect r, they must be the closest answer intervals to r. Besides, the interval *before* r with the largest end point is closer than other intervals before r. This implies that if such an interval is not the closest interval to r, neither are other intervals before r. Similar observation can be obtained for the intervals after r. That is, if the interval after r with the smallest start point is not the closest interval to r, neither are other intervals *after* r.

These *before* and *after* relationships are useful for skipping some unnecessary s when scanning S_i. Specifically, each one can be used to derive a skipping technique as follows:

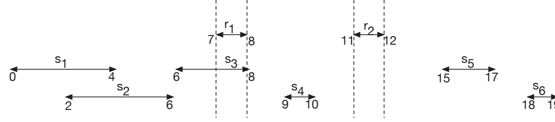


Figure 4.16: An example used for our algorithm

Skipping technique (i): We first describe how to use the *after* relationship to skip some answer intervals. Let us take Figure 4.16 as an example. Given the query interval r_1, we now scan S_i according to the increasing order of their start points. As soon as we scan to s_4, we can skip the remaining answer intervals s_5 and s_6, since we have $|r_1, s_3| == 0$ and $|r_1, s_4| > 0$. We are sure that all answer intervals after r_1 cannot be the closest intervals to it. The case is similar for r_2. As soon as we scan to s_5, we can skip the remaining answer interval s_6, since we have $|r_2, s_4| < |r_2, s_5|$, where s_4 is before r_2 with the largest end point and s_5 is after r_2 with the smallest start point.

Skipping technique (ii): The above showcases the usefulness of the *after* relationship. Actually, the *before* relationship is also useful to skip some scanning of S_i. For example, if we have known that the answer interval before r_1 with the largest end point is s_2, when turning to the next query interval r_2, we do not need to scan s_1 this time, we can just begin scanning from s_2.

The whole procedure is summarized in Algorithm 5. Line 10 will start scanning S from the position we scanned last time. In Line 11, we update the start position of S for the next r according to the skipping technique (ii). The reason we collect this set of answer intervals is that, if there are no answer intervals intersect r, this collected answer interval set could be its closest intervals. We continue to scan S till all answer intervals that intersect r (if any) are collected. If there exist answer intervals intersect r, we can skip the remaining answer intervals *after* r based on skipping technique (i). If not, we first check whether the answer intervals *before* r are closer to r than those *after* r in Line 19. If the answer is yes, we can skip the remaining answer intervals based on skipping technique (i). If no, Line 22 will scan S till all answer intervals *after* r with the smallest start point are collected. All remaining answer intervals will be skipped based on skipping technique (i). The algorithm will check which are the closest answer intervals and output the results in Line 23 to Line 26.

Let M be the size of R_i and N be the size of S_i. In Algorithm 5, we need to first sort R_i and S_i. These take $O(MlogM)$ and $O(NlogN)$ time. In the nested loop part, we only need to scan R once, so the time for this part is M. It would be a little bit complicated to analyze the time complexity of the inner loop. In fact, the scanning of S is output sensitive. Firstly, we have the $startIndex$ move from left to right gradually in the algorithm, which will take $O(N)$ time

Algorithm 5 Closest interval join(Local)

Input: Input interval sets R and S
Output: All closest interval pairs (r, s)
1: Sort R by increasing order of start point.
2: Sort S by increasing order of start point.
3: $startIndex \leftarrow 1$
4: **for all** $r \in R$ **do**
5: $beforeDist_{min} \leftarrow \infty$ ▷ The minimum distance between r and s, where s is *before* r.
6: $afterDist_{min} \leftarrow \infty$ ▷ The minimum distance between r and s, where s is *after* r
7: $BeforeClosestSet \leftarrow \varnothing$ ▷ The set of answer intervals which are before r with distance equals to $beforeDist_{min}$.
8: $AfterClosestSet \leftarrow \varnothing$ ▷ The set of answer intervals which are after r with distance equals to $afterDist_{min}$.
9: $OverlapResult \leftarrow \varnothing$
10: $BeforeClosestSet \leftarrow$ Start from $S[startIndex]$ till find the set of answer intervals that are *before* r with the largest end point.
11: $startIndex$ ←Find the index of first interval in S which are *before* r with the largest end point. ▷ Skipping the remaining answer intervals using technique (i).
12: $beforeDist_{min} \leftarrow Distance(r, S[b])$
13: $b \leftarrow$ startIndex + length(BeforeClosestSet) ▷ Index of S which may intersect or after r
14: $OverlapResult \leftarrow$ Start from $S[b]$ till find the set of answer intervals that intersect r
15: **if** $OverlapResult \neq \varnothing$ **then**
16: Output $(r, s), \forall s \in OverlapResult$
17: **Continue**
18: $afterDist_{min} \leftarrow Distance(r, S[b])$ ▷ There are no answer intervals overlapping r, so the interval after r could be the closest one.
19: **if** $beforeDist_{min} < afterDist_{min}$ **then**
20: Output $(r, s), \forall s \in BeforeClosestSet$
21: **Continue**
22: $AfterClosestSet \leftarrow$ Start from $S[b]$ till find the set of answer intervals that are *after* r with the smallest start point.
23: **if** $beforeDist_{min} > afterDist_{min}$ **then**
24: Output $(r, s), \forall s \in AfterClosestSet$
25: **else if** $beforeDist_{min} == afterDist_{min}$ **then**
26: Output $(r, s), \forall s \in BeforeClosestSet \cup AfterClosestSet$
27: **Continue**
28: **end for**

in total. Secondly, assuming the number of closest answer intervals for r_k is c_k, then for each r_k, we need to scan at most $(c_k + 1)$ of S_i, in order to collect all

the answers. This is because the movement of $startIndex$ has been counted by the first part. In the worst case, as soon as the $startIndex$ stops at the answer interval before r_k with the largest end point, we only need to further scan at most c_k to collect all the answers plus one more scan to determine that the rest cannot be the answers any more. Hence, the time complexity of our Algorithm 5 is $O(MlogM + NlogN + T)$, where T is the number of results output by the algorithm.

4.3 Experiments

In this section, we study the performance of our proposed methods. We conducted our experiments on both real and synthetic data. Our cluster has 10 machines connected with Gigabit network. Each machine has one Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz and 16GB RAM. We set one machine as the master node and the other 9 as slave nodes. Hadoop 2.7.1 is used in our experiment. As the map phase only reads each interval and process it one by one, it does not need large memory to store some intermediate results, so it is enough to allocate 1GB RAM for each *Mapper*. The reduce phase needs to read the two sets R and S into memory before performing the closest interval join, and some auxiliary data structures are created during the process, so we allocate 6GB RAM to each *Reducer* to guarantee it has enough memory to work with our real and synthetic data. So two concurrent *Reducer* instances can be launched in each node, and the maximum number of reducers can be run in parallel is 18 because we have 9 slaves. The HDFS block size is set to 128MB. To match the maximum number of reducers, the default number of partitions for BA and

NRA are 18. For BA, R and S are divided into 18 partitions respectively, which results in $18 * 18$ reducers. We choose this partition number because we want each reducer to consume roughly the same amount of data as NRA and DAA.

4.3.1 Real Data

As mentioned before, closest interval join operation is frequently conducted on signal track data in bioinformatics field. So we use these signal track data as our real data. Specifically, we use two real signal track files from UCSC Genome Bioinformatics Site [14]. They are the data files about the binding of a protein called CTCF to the DNA in two different cell lines, with or without estrogen treatment. The purpose is to find the binding site in a cell line that is closest to the binding site in another cell line, which can be part of an analysis of finding cell-type-specific binding. The two files are named R and S. File R contains 60 million intervals and file S contains 53 million intervals. The final result shows that there are about 80% pairs are closest intervals with distance greater than 0.

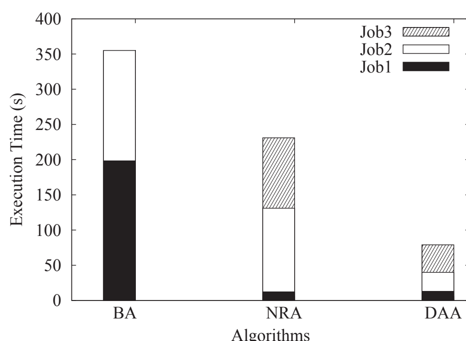


Figure 4.17: Time breakdown for real data

Figure 4.17 shows the running time of the three algorithms. We can see that DAA can outperform NRA and BA by 3 times and 5 times respectively.

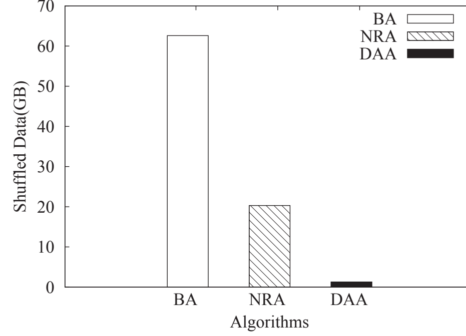


Figure 4.18: Shuffling cost for real data

The main reason is that a lot of unnecessary replications of R and S are avoided by our mapping rules, which saves the shuffling cost and computation time in the reducer. This can be observed in Figure 4.18, which summarizes the total size of shuffled data of each algorithm. For example, DAA's shuffled size is only about 10% of BA. Besides, since both BA and NRA can only find the local optimum in the first place, they need an extra job to find the global optimum. This introduces another overhead. Figure 4.17 also shows the time breakdown of the three algorithms. The last job of BA (i.e., Job2) and NRA (i.e., Job3) are the extra jobs to find the global optimum, while DAA only needs one job (i.e., Job3) to carry out the actual closest computation. Remember that Job1 for DAA is to build the initial partition scheme and Job2 is to adjust the initial partition scheme, and Job3 is to partition R and S according to our mapping rules based on partition scheme.

In addition, for the local algorithm used in each reducer, we did a comparison between the simple nested loop and our proposed Algorithm 5. We find that, even for DAA, it takes extremely long time to finish when the simple nested loop algorithm is adopted in each reducer. To verify that, we ran the two centralized

algorithms in one reducer with the real data R and S. The experiment shows that Algorithm 5 takes about 200 seconds to finish, but nested loop needs at least 12 hours. As it was too slow, we terminated it after 12 hours. Then we generated 1 million intervals for R and S respectively. It takes about 2 seconds for Algorithm 5 to finish, but 6700 seconds for nested loop algorithm to finish. As the time complexity of nested loop algorithm is quadratic, we can estimate the rough running time based on the current experiment result. It shows that the performance improvement gained by Algorithm 5 is significant.

4.3.2 Synthetic Data

In order to study the performance of the three algorithms in a controlled manner, we generated synthetic data. Similar to [19], the parameters we use to control the synthetic intervals are: 1) number of intervals N; 2) the start point distribution SD; 3) the interval length distribution LD. The domain range $D = [0, 1B)$, B = billion, it means each interval's start and end point lie in this range. The average interval length is set to $L = 10$. The default parameters we used are shown in Table 4.4.

Parameter	Default Value
N	700M, M = million
SD	Uniform
LD	Uniform

Table 4.4: Default parameters for synthetic data generator

Effect of data size: To study the performance of our method when varying the data size, we generate four groups of dataset. We fixed the size of the second interval set and varied the size of the first linearly. The details are shown in

Table 4.5. For example, in group G1, we generate 350 million intervals for R and 700 million intervals for S. The running time for G1-G4 is shown in Figure 4.19. The corresponding shuffling data size is shown in Figure 4.20. When the data size is large, our method can outperform BA by an order of magnitude and NRA by more than 5 times. The main reason is again that a lot of unnecessary replications of R and S are avoided by our mapping rules. Furthermore, we can observe that our algorithm scales linearly when the data size increases. Figure 4.21 is a time breakdown of the three algorithms. We can see that under G2, the running time of Job1 and Job2 in DAA become negligible when the data size is large.

Group	First Interval Set	Second Interval Set
G1	R=350M	S=700M
G2	R=700M	S=700M
G3	R=1.05B	S=700M
G4	R=1.4B	S=700M

Table 4.5: Varying data size

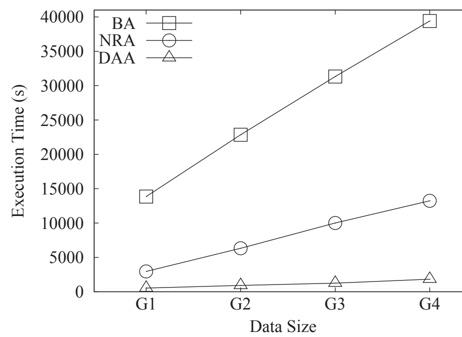


Figure 4.19: Execution Time vs. Data Size

Effect of number of partitions: In this experiment, we study the performance of our method when varying the number of partitions by specify it via program

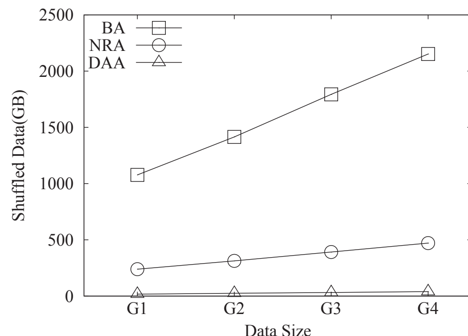


Figure 4.20: Shuffling Cost vs. Data Size

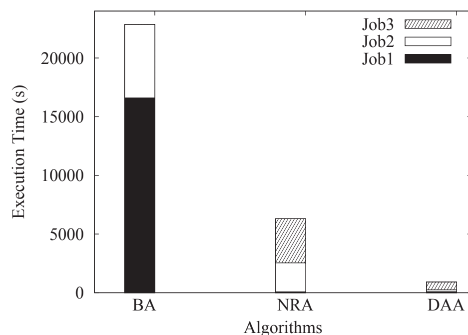


Figure 4.21: Time breakdown for G2

arguments. Figure 4.22 shows the result. We can see that the running time first decreases as the number of partitions increases, but then goes up when the number of partitions is larger than 54. In theory, the more partitions, the better the performance it is. This is because the running time of our algorithm is sublinear to the number of partitions. As the partition number increases, the number of intervals received by each partition would be smaller. This results in faster completion time of each reducer. Remember that the time complexity of our Algorithm 5 running in each reducer is $O(MlogM + NlogN + T)$. Assume we double the number of partitions, then the number of intervals received by each reducer will be half of before and the running time of each reducer becomes

$O(\frac{M}{2}log\frac{M}{2} + \frac{N}{2}log\frac{N}{2} + \frac{T}{2}) = O(\frac{M}{2}(logM - log2) + \frac{N}{2}(logN - log2) + \frac{T}{2})$, which is more than 2 times faster. However, Figure 4.22 shows a sweet spot at number 36. There are two reasons. First, since the maximum number of concurrent reducers is fixed, the more partitions, the more waves we need to finish the reducers. For example, when we double the partition number from 18 to 36, it requires 2 waves to finish all reducers. Second, as the number of reducers increases, the task start-up and scheduling overhead become dominant and cancel out the performance gained by Algorithm 5. These explain the "U" shape of Figure 4.22.

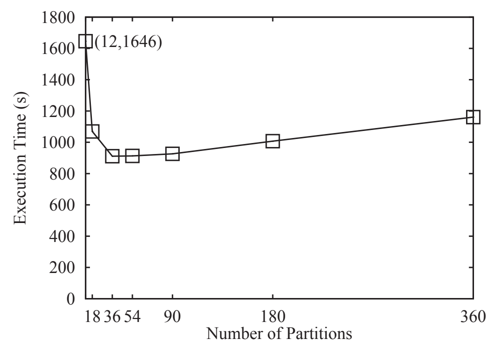


Figure 4.22: Execution Time vs. Number of Partitions

Effect of cluster size: In this experiment, we vary the number of nodes for DAA. In our cluster, one node is the master node for job scheduling and the other 9 are the slave nodes. So we changed the slave nodes from 1 to 9 using the G2 data. The number of partitions we configured is 18 in each experiment, each node can run two reducers concurrently. Figure 4.23 shows that the running time decreases when the number of working nodes increases. The sharpest decrease happened when the number is changed from 1 to 3. That's because when the number is 1, there is only one slave node that works. As there are only two reducers running concurrently in a node, it will take one slave node 9 waves to finish the whole

job. When there is almost no parallelism, the overhead of spawning a full fledged program (mapper or reducer) will become an unignorable factor which affects the total running time. If the number of slave nodes is 3, there can be 6 reducers running at the same time, only 3 waves are needed to finish this. As parallelism increases, the overhead of startup/shutdown of mappers and reducers becomes less and less influential in the total execution time, so the time almost linearly reduced with the number of nodes increases such as from 3 to 9.

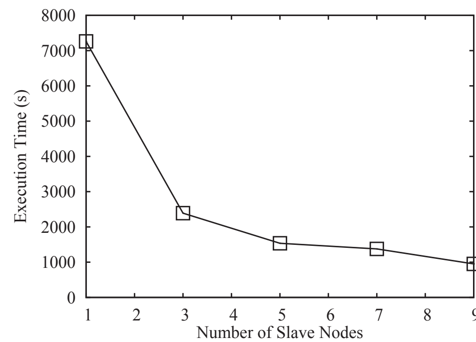


Figure 4.23: Execution Time vs. Number of Slave Nodes

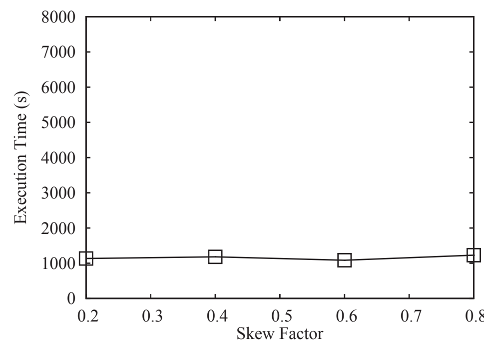


Figure 4.24: Varying the skewness on interval length

Effect of data skew: In this experiment, we vary the interval length distribution and start point distribution of G2 data. We use zipf distribution, which has a parameter θ. When θ is smaller, the data tends to uniform; when it is larger,

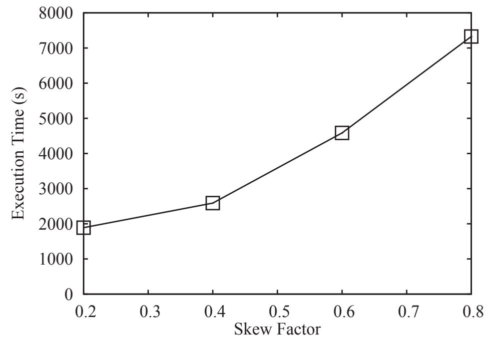


Figure 4.25: Varying the skewness of starting position

the data tends to skew. Figure 4.24 shows the result of DAA when we vary the interval length through θ but keep the start points uniformly distributed. It shows that the performance of DAA is not influenced much because the start points remain uniformly distributed and DAA's first job (sampling) would still evenly partition the data as before. Figure 4.25 shows the result of DAA when we vary the start point through θ but keep the interval lengths uniformly distributed. It follows the folklore and indicates an increase in running time because such data skewness increases the data volume for a small number of reducers. How to deal with data skewness in MR environment is a standalone topic [41] and we regard that as a future work in the context of interval data.

4.3.3 Summary

In summary, there are three reasons that DAA is always the best. **First**, DAA takes extra efforts to adjust the partition boundary before it partitions R. Compared with NRA, this strategy guarantees each r in R can be shuffled to only the exact partitions that will generate the output and thus save a lot of traffic. On the other hand, NRA will send r to the neighbor partitions as

well. **Second**, both of BA and NRA have to find the global optimum with one additional MapReduce job. This job has to start a reducer for each r and shuffle all the r related results to it, which will cause large data traffic. **Third**, the efficient local algorithm of DAA further helps reduce the computation time greatly.

Chapter 5

Conclusions

The system presented in this thesis consists of four major components:

1. A declarative language called Signal Track Query Language (STQL), which is an SQL-like language we specifically designed to suit the needs for analyzing genomic signal tracks.

2. A system built on top of a parallel architecture based on the MapReduce distributed storage and processing framework for big data and the Hive data warehouse infrastructure. It facilitates the execution of each user task on multiple machines in our computer cluster in parallel.

3. A simple and user-friendly website that helps users construct and execute queries, upload/download compressed data files in various formats, manage stored data, queries and analysis results, and share queries with other users. It also provides a complete help system, detailed specification of STQL, and a large number of sample queries for users to learn STQL and try it easily.

Private files and queries are not accessible by other users.

4. A repository of public data popularly used for large-scale genomic data analysis, including data from ENCODE and Roadmap Epigenomics, that users can use in their analyses.

The current version of START has been running for almost one year. It has already been used by our collaborators.

Finally, to optimize one key operation closest interval join in START, we designed three algorithms to handle it on MapReduce. The broadcast algorithm is easy to implement, but will cause a huge network traffic which makes it extremely slow in practice. The neighbor replicating algorithm will reduce the network traffic by sending intervals to just its neighbor partitions, but there are still unnecessary replications. The distance-aware algorithm considers adjusting the partition boundary and the distance between an interval and the partition in order to send the interval to exact partitions that will output the result. Besides, we also proposed an efficient centralized algorithm to compute the closest interval join in each reducer, which has a significant improvement to the total job execution time. The experiments showed the distance-aware algorithm combined with the efficient centralized algorithm is scalable and can gain the shortest execution time compared with other two algorithms.

In the future, there are some interesting directions to explore:

- With technology advancement, the implementation of MapReduce framework has also changed a lot. It would be interesting to integrate our system to Spark [4] or integrate GPU acceleration power into our system [30].

- As different users will issue similar queries at the same time or different queries will touch the same track data, how to connect queries into workflows and offer more effective resource allocation policies with the declarative system is another interesting topic.

- In addition to offering STQL, it will also be helpful to provide simple APIs for advanced users to manipulate the track data more flexibly.

Appendix A

Full set of example queries

A.1 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’.‘wgEncodeBroadHistoneGm12878H3k04-me1StdSigV2.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-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'.'wgEncodeCshlLongRnaSeqGm128-78CellTotalPlusRawSigRep1.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: 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 T_1 **intersectjoin** T_2;

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‘.‘wgEncodeCshlLongRnaSeqGm128-78CellTotalPlusRawSigRep1.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‘.‘wgEncodeCshlLongRnaSeqGm128-78CellTotalPlusRawSigRep1.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 T_1 $TInt_1$, T_2 $TInt_2$
WHERE $TInt_1$ **overlaps with** $TInt_2$;

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.

A.2 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 high occupancy (HOT) regions [61].

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 FOR TRACK IN () 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 group neighboring genomic locations with the same number of overlapping transcription factor binding peak count into a region disjoint from other regions. These counts are computed by using the **vd_sum** operation with the **EACH MODEL** of interval values.

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 [61])
- T_2: ‘HumanMetaTracks‘.‘PRM_Gm12878_merged.bed‘ (Promoter-proximal regulatory regions in GM12878 as defined in [61])
- 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 bind-

ing 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'.'wgEncodeBroadHistoneK562H3k27ac-StdSig.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      T1
WHERE     feature = 'gene' AND
          attributes LIKE '%gene_type "protein_coding"%';

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

CREATE TRACK Step3Results AS
SELECT    NtIntB.chr, NtIntB.chrstart, NtIntB.chrend,
NtIntB.value - NtIntC.value as value
FROM      (project T2 on Step2Results with vd_sum using
           EACH MODEL) NtIntB,
          (project T3 on Step2Results with vd_sum using
           EACH MODEL) NtIntC
```

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.

CQ6 Analysis task: To identify genomic regions with bi-directional transcription at their flanking regions, which could be potential enhancers producing enhancer RNAs (eRNAs) [24, 39].

Query template:

```
CREATE TRACK Step1Results AS
SELECT    chr, chrstart - 200 AS chrstart, chrend - 200 AS chrend
FROM      T1
WHERE     value > 2;

CREATE TRACK Step2Results AS
SELECT    chr, chrstart + 200 AS chrstart, chrend + 200 AS chrend
FROM      T2
WHERE     value > 2;

SELECT    *
FROM      Step1Results intersectjoin Step2Results;
```

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. Each region in the final signal track 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 [24].

Appendix B

STQL grammar rules

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

Bibliography

[1] ANTLR. http://www.antlr.org/.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Apache Pig. https://pig.apache.org/.

[4] Apache Spark. http://spark.apache.org/.

[5] Bedtools. http://bedtools.readthedocs.org/en/latest.

[6] Ensembl Genome Browser. http://www.ensembl.org/index.html.

[7] Exome Aggregation Consortium ExAC Browser. http://exac.broadinstitute.org/.

[8] Genome researchers raise alarm over big data. http://www.nature.com/news/genome-researchers-raise-alarm-over-big-data-1.17912/.

[9] Genomic Intervals. https://www.biostarhandbook.com/unit/intervals/genomic-intervals.html.

[10] Integrative Genomics Viewer. http://software.broadinstitute.org/software/igv/.

[11] InterMine. http://intermine.org/.

[12] Map Viewer. http://www.ncbi.nlm.nih.gov/mapview/.

[13] UCSC Genome Browser. https://genome.ucsc.edu/.

[14] WgEncodeOpenChromChip for download. http://hgdownload.cse.ucsc.edu/goldenPath/hg19/encodeDCC/wgEncodeOpenChromChip/.

[15] What is SNP. http://ghr.nlm.nih.gov/handbook/genomicresearch/snp.

[16] Robin Andersson, Claudia Gebhard, Irene Miguel-Escalada, Ilka Hoof, et al. An atlas of active enhancers across human cell types and tissues. *Nature*, 507(7493):455–461, 2014.

[17] Tokuzo Arao, Kazuomi Ueshima, Kazuko Matsumoto, Tomoyuki Nagai, et al. FGF3/FGF4 amplification and multiple lung metastases in responders to sorafenib in hepatocellular carcinoma. *Hepatology*, 57:1407–1415, 2013.

[18] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *SC*, 2011.

[19] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A. Faruquie, L. Venkata Subramaniam, and Mukesh K. Mohania. Processing Interval Joins On Map-Reduce. In *EDBT*, 2014.

[20] Lynda Chin, Jannik N Andersen, and P Andrew Futreal. Cancer genomics: from discovery science to personalized medicine. *Nature medicine*, 17(3):297–303, 2011.

[21] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[22] Ana Conesa and Ali Mortazavi. The common ground of genomics and systems biology. *BMC Systems Biology*, 8(2):1–10, 2014.

[23] 1000 Genomes Project Consortium et al. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 2012.

[24] Leighton J Core, Andre L Martins, Charles G Danko, Colin T Waters, et al. Analysis of nascent RNA identifies a unified architecture of initiation regions at mammalian promoters and enhancers. *Nature Genetics*, 46(12):1311–1320, 2014.

[25] Mark De Berg, Otfried Cheong Schwarzkopf, Marc Van Kreveld, and Mark Overmars. In *Computational Geometry: Algorithms and Applications*, pages 219–241. Springer, 2008.

[26] Dean, Jeffrey and Ghemawat, Sanjay. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[27] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *VLDB*, 1992.

[28] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Overlap interval partition join. In *SIGMOD*, 2014.

[29] Ahmed Eldawy and Mohamed F. Mokbel. A Demonstration of Spatial-Hadoop: An Efficient Mapreduce Framework for Spatial Data. In *VLDB*, 2013.

[30] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K. Govindaraju. Mars: Accelerating mapreduce with graphics processors. *IEEE Trans. Parallel Distrib. Syst.*, 22(4):608–620, 2011.

[31] Raphael A. Finkel and Jon Louis Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. In *Acta Informatica*, 1974.

[32] Jeremy Goecks, Anton Nekrutenko, James Taylor, and The Galaxy Team. Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11:R86, 2010.

[33] Himawan Gunadhi and Arie Segev. Query Processing Algorithms for Temporal Intersection Joins. In *ICDE*, 1991.

[34] Jennifer Harrow, Adam Frankish, Jose M. Gonzalez, Electra Tapanari, et al. GENCODE: The reference human genome annotation for the ENCODE project. *Genome Research*, 22:1760–1774, 2012.

[35] B Hentsch, I Lyons, R Li, L Hartley, et al. 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.

[36] H. Heyn, E. Vidal, H. J. Ferreira, M. Vizoso, et al. Epigenomic analysis detects aberrant super-enhancer DNA methylation in human cancer. *Genome Biol.*, 17(1):11, 2016.

[37] Ya-Ting Hsu, Fei Gu, Yi-Wen Huang, Joseph Liu, et al. Promoter hypomethylation of EpCAM-regulated bone morphogenetic protein gene fam-

ily in recurrent endometrial cancer. *Clinical Cancer Research*, 19:6272–6285, 2013.

[38] Johannes W. G. Janssen, Jan-Willem Vaandrager, Tanja Heuser, Anna Jauch, et al. 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.

[39] Tae-Kyung Kim, Martin Hemberg, Jesse M. Gray, Allen M. Costa, et al. Widespread transcription at neuronal activity-regulated enhancers. *Nature*, 465(7295):182–187, 2010.

[40] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing Intervals Efficiently in Object-Relational Databases. In *VLDB*, 2000.

[41] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. SkewTune: Mitigating Skew in Mapreduce Applications. In *SIGMOD*, 2012.

[42] Hongjun Lu, Beng Chin Ooi, and Kian-Lee Tan. On Spatially Partitioned Temporal Join. In *VLDB*, 1994.

[43] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient Processing of k Nearest Neighbor Joins using MapReduce. In *PVLDB*, 2012.

[44] Alper Okcan and Mirek Riedewald. Processing Theta-joins Using MapReduce. In *SIGMOD*, 2011.

[45] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, et al. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[46] A. Petersen, C. Alvarez, S. DeClaire, and N. L. Tintle. Assessing methods for assigning SNPs to genes in gene-based tests of association using common variants. *PLoS ONE*, 8(5):e62161, 2013.

[47] Aaron R. Quinlan and Ira M. Hall. BEDTools: A flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26:841–842, 2010.

[48] Roadmap Epigenomics Consortium, Anshul Kundaje, Wouter Meuleman, Jason Ernst, et al. Integrative analysis of 111 reference human epigenomes. *Nature*, 518(7539):317–330, 2015.

[49] Hanan Samet, Jagan Sankaranarayanan, and Michael Auerbach. Indexing Methods for Moving Object Databases: Games and Other Applications. In *SIGMOD*, 2013.

[50] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. ClusterJoin: A Similarity Joins Framework using Map-Reduce. In *PVLDB*, 2014.

[51] Eric T. Sawey, Maia Chanrion, Chunlin Cai, Guanming Wu, et al. Identification of a therapeutic strategy targeting amplified FGF19 in liver cancer by oncogenomic screening. *Cancer Cell*, 19:347–358, 2011.

[52] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[53] Inga Sitzmann and Peter J. Stuckey. Improving Temporal Joins Using Histograms. In *DEXA*, 2000.

[54] Haoyu Tan, Wuman Luo, and Lionel M. Ni. Clost: a hadoop-based storage system for big spatio-temporal data analytics. In *CIKM*, 2012.

[55] The Cancer Genome Atlas Research Network, John N Weinstein, Eric A Collisson, Gordon B Mills, et al. The cancer genome atlas pan-cancer analysis project. *Nature Genetics*, 45(10):1113–1120, 2013.

[56] The ENCODE Project Consortium. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57–74, 2012.

[57] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, et al. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

[58] K. Wang, M. Li, and M. Bucan. Pathway-based approaches for analysis of genomewide association studies. *Am. J. Hum. Genet.*, 81(6):1278–1283, Dec 2007.

[59] Wenqi Wang, Jun Huang, Xin Wang, Jingsong Yuan, et al. PTPN14 is required for the density-dependent control of YAP1. *Genes and Development*, 26:1959–1971, 2012.

[60] Warren A. Whyte, David A. Orlando, Denes Hnisz, Brian J. Abraham, et al. Master transcription factors and mediator establish super-enhancers at key cell identity genes. *Cell*, 153:307–319, 2013.

[61] Kevin Y. Yip, Chao Cheng, Nitin Bhardwaj, James B. Brown, et al. Classification of human genomic regions based on experimentally-determined binding sites of more than 100 transcription-related factors. *Genome Biology*, 13:R48, 2012.

[62] Stephens ZD, Lee SY, Faghri F, Campbell RH, et al. Big Data: Astronomical or Genomical? In *PLoS BIOLOGY*, 2015.

[63] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *EDBT*, 2012.

[64] Qiang Zhang, Andy He, Chris Liu, and Eric Lo. Closest Interval Join Using MapReduce. In *DSAA*, 2016.

[65] Yiwen Zhu, Miek C. Jong, Kelly A. Frazer, Elaine Gong, et al. Genomic interval engineering of mice identifies a novel modulator of triglyceride production. *Proceedings of the National Academy of Sciences*, 97(3):1137–1142, 2000.