Framework for Supporting Genomic Operations

Abdulrahman Kaitoua, Pietro Pinoli, Michele Bertoni and Stefano Ceri

Abstract—Next Generation Sequencing (NGS) is a family of technologies for reading the DNA or RNA, capable of producing whole genome sequences at an impressive speed, and causing a revolution of both biological research and medical practice. In this exciting scenario, while a huge number of specialized bio-informatics programs extract information from sequences, there is an increasing need for a new generation of systems and frameworks capable of integrating such information, providing holistic answers to the needs of biologists and clinicians. To respond to this need, we developed GMQL, a new query language for genomic data management that operates on heterogeneous genomic datasets. In this paper, we focus on three domain-specific operations of GMQL used for the efficient processing of operations on genomic regions, and we describe their efficient implementation; the paper develops a theory of binning strategies as a generic approach to parallel execution of genomic operations, and then describes how binning is embedded into two efficient implementations of the operations using Flink and Spark, two emerging frameworks for data management on the cloud.

Index Terms—Genomic data management, Data modeling, Query languages, Operations for genomics, Cloud-based systems.

1 Introduction

Next Generation Sequencing (NGS) is a technology for reading the DNA that is changing biological research and will change medical practice; thanks to the availability of millions of whole genome sequences, genomic data management may soon become the biggest and most important "big data" problem of mankind.

Data management in genomics applies to three different phases. *Primary analysis* is concerned with producing raw data in the form of short reads of DNA or RNA sequences; *secondary analysis* is concerned with extracting the DNA or RNA sequences from the reads (alignment) or evaluating (or *calling*) specific features from aligned files (e.g., mutations or peaks of expression); this processing is performed by a large number of bioinformatics tools, some developed by using Pig [44] or Spark [53]. Data management systems developed so far concentrate on secondary analysis (e.g., [31], [47]); Adam [2], an offset of Spark dedicated to genomics, is also focused on secondary analysis. But the most important, emerging problem is the so-called tertiary analysis (see Fig. 1), which is concerned with *sense making*, e.g., discovering how heterogeneous regions interact with each other, by integrating heterogeneous DNA features, such as variations (e.g., a mutation in a given DNA position), or peaks of expression (e.g., regions with higher DNA read density), or structural properties of the DNA, e.g., break points (where the DNA is damaged) or junctions (where DNA creates loops).

In this context, we are currently developing a new, holistic approach to genomic data modelling and querying that uses cloud-based computing to manage heterogeneous data produced by NGS technology [33]. Our approach is based on a new, high-level query language, called GenoMetric Query Language (GMQL), which enables building new datasets from a repository of existing datasets, using algebraic operations. GMQL can be used for querying thousands of samples of processed data, which are becoming available at large sequencing centers, and are being assembled by international consortia (such as ENCODE [21], TCGA [52], and 1000 Genomes Project [1]).

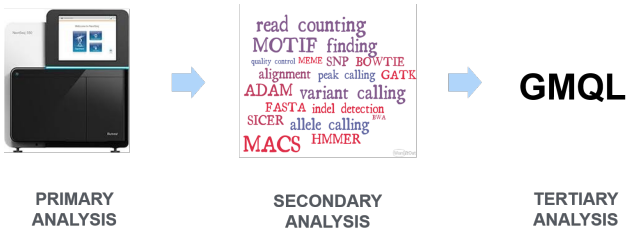

Fig. 1. Phases of genomic data analysis.

Our approach is truly multidisciplinary, as it combines data modeling, big data, cloud computing, systems architecture and parallel algorithms; in addition, it applies to genomics, whose relevance is huge and emerging. To the best of our knowledge, no academic group is specifically focusing on tertiary data analysis; the spin-off company *Paradigm4*[1], also focuses on tertiary analysis, by developing genomic adds-on to SciDB [5]; they advocate the use of SciDB, a specialized scientific database, rather than cloud computing. While our work is centered upon the outcomes of NGS technology, other research is concerned with the integration of *omics* datasets, as resulting, e.g. from genomics, proteomics, and interactomics; among them, [26] addressed data integration and parallel processing of microarray data, focusing on mutations and pharmacogenomics.

In our initial release, available for download[2], we translated GMQL to PIG [8]; we are currently working towards a new GMQL release, that will become available during 2016, and will support two parallel implementations, re-

• *A. Kaitoua, P. Pinoli, M. Bertoni and S. Ceri are with the Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano, Italy. E-mail: {firstname.lastname}@polimi.it*

1. Founded by Turing award Mike Stonebraker.
2. http://www.bioinformatics.deib.polimi.it/genomic_computing/

spectively using Flink [6] and Spark [9], two emerging data frameworks. Architecture design is driven by strong platform portability requirements: the two implementations differ only in the encoding of about twenty data management operations, while the compiler, logical optimizer, and APIs/UIs are independent from the adoption of either framework. In this paper, we focus on the parallel implementation of three domain-specific GMQL operations on the cloud, called *map, cover and join*, each with several variants (e.g., distal and mindistance joins, summit and flat variants for cover); they are the highly critical operations, as the performance of GMQL depends essentially on these operations.

In [12] we have presented a comparison of Flink and Spark at work on much simpler abstractions for genomics; in this paper, we extend that work by considering the broader spectrum of domain-specific GMQL operations (with a rich set of syntactic and semantic variants), we fully develop algorithms for join, cover and map (with an optimized, three-step approach to join evaluation) and we develop a theory of **binning strategies** as a generic approach to their parallel execution (which allows a simplification of the parallel processing).

The remaining of this paper is structured as follows. Section 2 describes the model describing DNA regions and the main operations for computing result regions as effect of domain-specific operations, named `JOIN`, `MAP`, and `COVER`. Section 3 defines our implementation of domain-specific operations using Flink and Spark, and in particular develops binning algorithms used in order to parallelize the operations over huge numbers of genomic regions. Section 4 presents the evaluation of our implementation, and discusses a number of optimizations and parameter tunings which are needed to increase performance. Section 5 presents the state of the art and discusses other related current approaches and implementations. Section 6 concludes.

2 Genomic Applications

Genomic applications discussed in this paper share a common, simple model based on the notion of *DNA region*, i.e. a portion of the genome placed within one chromosome and further characterized by a start and stop position. Regions are aligned with respect to a reference genome (e.g. reference `hg19` for the human species), therefore regions are related to each other by a single system of coordinates. We can think of a region as defined by the triple `<chromosome,start,stop>`, e.g. `<7,145677,678999>`, with `start<stop`, and it is possible to define operations among them, such as taking two regions and computing their union, difference, and intersection; these operations produce a result when two regions are both on the same chromosome, e.g. an intersection region is produced when one end of either regions is enclosed within the two ends of the other region. Every region may have a value describing the properties of the region, called the region's *features*; the value can be missing, or it can be a single real number (e.g., if the region represents a portion of the DNA reacting to a given treatment using an antibody, the value may be an indication of the significance of the reaction), or as complex as the sequence of amino acids which are present in the region (i.e. a string of the letters `A`, `C`, `G`, `T` with a length equal to `stop-start`; each position - or basis - can also be associated with an encoded indication of the confidence of that specific position). Thus, the value of a region can be complex, and has to be associated with a given signature, which depends on the type of experiment producing it.

We collect the regions which are produced by each experimental condition (e.g. a specific cell culture) within a file or *sample*; we then collect the samples produced by the same experiment type (hence with the same signature) within the same *dataset*; each sample within a dataset has a unique identifier.

In summary, each region is represented as a quintuple:

```
Sample = 1837
Chromosome = 1
Start = 15834
Stop = 16135
Values = []
```

A GMQL query (or program) is expressed as a sequence of GMQL operations with the following structure:

```
<var> = operation(<parameters>) <vars>
```

where each variable stands for a GMQL dataset. Operations are either unary (with one input variable), or binary (with two input variables), and construct one result variable; all operations produce a result dataset consisting of several samples, whose identifiers are either inherited by the operands or generated by the operation. Most GMQL operations are extension of classic relational algebra operations, twisted to the needs of genomics. Three domain-specific operations, called `COVER`, `JOIN` and `MAP`, significantly extend the expressive power of classic relational algebra.

In [33], we demonstrated the expressive power and flexibility of GMQL through multiple biological examples, including finding distal bindings in transcription regulatory regions, associating transcriptomics and epigenomics, and finding somatic mutations in exons. The full GMQL data model includes also metadata, which are not used in this paper. Compared with languages which are currently in use by the bioinformatics community, GMQL is *declarative* (it specifies the structure of the results, leaving its computation to each operation's implementation) and *high-level* (one GMQL query typically substitutes for a long program which embeds calls to region manipulation libraries); the progressive computation of variables recalls other algebraic languages (e.g. *Pig Latin*, [8]). We next focus on the relevant aspects of domain-specific GMQL operations for genomics.

2.1 Join

The `JOIN` operation applies to two datasets, respectively called **anchor** and **experiment**; the operation produces a result sample for every pair of samples of the operand datasets, whose identifier is obtained by applying a hash function to the identifiers of the operand samples; the regions within each result sample are generated from the regions of the operand samples that satisfy a genometric predicate; their coordinates are computed according to four region composition options and their values are obtained

by concatenating the values of the regions of the operands[3]. Thus, the join operation produces results that can grow quadratically both in the number of samples and of regions; hence, it is the most critical GMQL operation from a computational point of view.

Genometric predicates are based on the notion of **genomic distance**, defined as the number of bases (i.e. nucleotides) between the closest opposite ends of two regions, measured (using a numeric type, e.g. `Integer`) from the right end of the region with left end lower coordinate.[4] A genometric predicate is a sequence of distal conditions, defined as follows:

- `UP/DOWN` denotes the *upstream* and *downstream* directions of the genome. They are interpreted as predicates that must hold on the region of the experiment; `UP` is true when it is in the *upstream genome* of the anchor region[5]. When this clause is not present, distal conditions apply to both the directions of the genome.
- `MD(K)` denotes the *minimum distance* clause; it selects the K regions of the experiment at minimal distance from the anchor region. When there are ties (i.e. regions at the same distance from the anchor region), regions of the experiment are kept in the result even if they exceed the K limit.
- `DLE(N)` denotes the *less distance* clause; it selects all the regions of the experiment such that their distance from the anchor region is less than or equal to N bases[6].
- `DGE(N)` denotes the *greater distance* clause; it selects all the regions of the experiment such that their distance from the anchor region is greater than or equal to N bases.

Genometric clauses are composed by strings of distal conditions; a genometric clause is **well-formed** only if it includes the *less distance* clause; we expect all clauses to be well formed, possibly because the clause `DLE(Max)` is automatically added at the end of the string, where `Max` is a problem-specific maximum distance.

Examples. The following strings are legal genometric predicates:

```
DGE(500), UP, DLE(1000), MD(1)
DGE(50000), UP, DLE(100000)
DLE(2000), MD(1), DOWN
MD(100), DLE(3000)
```

Note that different orderings of the same distal clauses may produce different results; this aspect has been designed in order to provide all the required biological meanings, and is further discussed in Section 3.1.1, where we discuss distal clause evaluation.

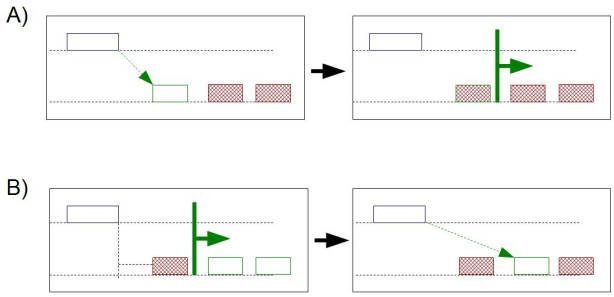

Fig. 2. Different semantics of genometric clauses due to the ordering of distal conditions. The vertical bar is set at distance 100 from the reference region. In case (A) the minimum distance region is first selected (on the left) and then excluded by the distance predicate (on the right), therefore no region is produced. In case (B) the distance predicate selects two regions (on the left), out of which the minimum distance region is selected (on the right).

Example. In Fig. 2 we show an evaluation of the following two clauses relative to an anchor region: `A: MD(1),DLE(100), B: DLE(100),MD(1).` In case A, the `MD(1)` clause is computed first, producing one region which is next excluded by computing the `DLE(100)` clause; therefore, no region is produced. In case B, the `DLE(100)` clause is computed first, producing two regions, and then the `MD(1)` clause is computed, producing as result one region[7].

Similarly, the clauses `A: MD(1),UP` and `B: UP,MD(1)` may produce different results, as in case A the minimum distance region is selected regardless of streams and then retained iff it belongs to the upstream of the anchor, while in case (B) only upstream regions are considered, and the one at minimum distance is selected.

2.2 Map

`MAP` is a binary operation over two datasets, respectively called **reference** and **experiment**. Let us consider one reference sample, with a set of reference regions; the operation computes, for each sample in the experiment, new values produced by aggregation functions over the values of the experiment regions that intersect with each reference region; we say that *experiment regions are mapped to reference regions*. The operation produces a regular structure, called **genomic space**, where each experiment sample is associated with a row, each reference region with a column, and each matrix entry is a single value[8]. Thus, a `MAP` operation allows a quantitative reading of experiments with respect to the reference regions; when the biological function of the reference regions is not known, `MAP` helps in extracting the most interesting regions out of many candidates.

Example. Fig. 3 shows the effect of this `MAP` operation on a small portion of the genome; the input consists of one

3. See [33], where a full account of the join operation is presented, including region composition options, join partitioning, and metadata management.

4. With our choice of interbase coordinates, intersecting regions have distance less than 0 and adjacent regions have distance equal to 0; if two regions belong to different chromosomes, their distance is undefined (and predicates based on distance fail).

5. *Upstream* and *downstream* are technical terms in genomics, and they are applied to regions on the basis of their *strand*. For regions of the *positive strand*, `UP` is true for those regions of the experiment whose right end is lower than the left end of the anchor, and `DOWN` is true for those regions of the experiment whose left end is higher than the right end of the anchor. For the *negative strand*, ends and disequations are exchanged.

6. DGE(-1) is true when the region of the experiment overlaps with the anchor region; DGE(0) is true when the region of the experiment is adjacent to or overlapping with the anchor region.

7. The two queries can be expressed as: *produce the minimum distance region iff its distance is less than 100 bases* and *produce the minimum distance region after 100 bases*.

8. Biologists typically consider the transposed matrix, because there are fewer experiments (on columns) than regions (on rows). Such matrix can be observed using heat maps, and its rows and/or columns can be clustered to show patterns.

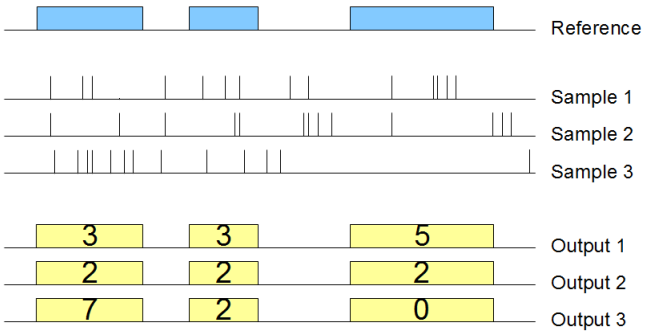

Fig. 3. Example of map using one sample as reference and three samples denoting mutations as experiment, using the Count function.

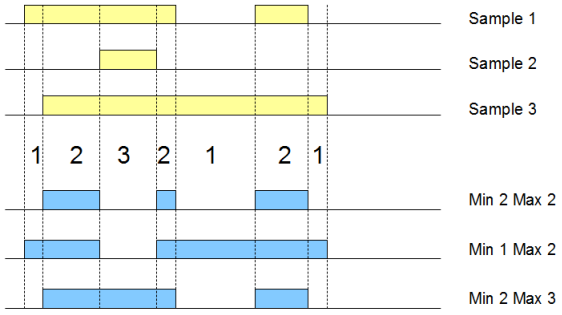

Fig. 4. Accumulation index and COVER results with three different minAcc and maxAcc values.

reference sample and three experiment samples, the output consists of three samples with the same regions as the reference sample, whose features corresponds to the number of mutations which intersect with those regions. The result can be interpreted as a (3×3) genome space.

2.3 Cover

The COVER operation applies to a single dataset and computes a single sample from several input samples by taking into account region intersections. In the basic COVER operation, each resulting region r is the contiguous intersection of at least minAcc and at most maxAcc regions r_i in the input samples; minAcc and maxAcc are called **accumulation indexes**[9].

Resulting regions may have new attributes Ar, calculated by means of aggregate expressions over the attributes of the contributing regions. Jaccard Indexes[10] are standard measures of similarity of the contributing regions r_i, added as default attributes. Three variants of the basic COVER are biologically relevant:

- The HISTOGRAM variant returns the nonoverlapping regions contributing to the cover, each with its accumulation index value, which is assigned to the **AccIndex** region attribute.
- The FLAT variant returns the union of all the regions which contribute to the COVER (more precisely, it returns the contiguous region that starts from the first end and stops at the last end of the regions which would contribute to each region of the COVER).
- The SUMMIT variant returns only those portions of the result regions of the COVER where the maximum number of regions intersect (more precisely, it returns regions that start from a position where the number of intersecting regions is not increasing afterwards and stops at a position where either the number of intersecting regions decreases, or it violates the max accumulation index).

Example. Fig. 4 shows three applications of the COVER operation on a small portion of the genome; the figure shows the regions resulting from setting the MinAcc and MaxAcc parameters respectively to $(2, 2)$, $(1, 2)$, and $(2, 3)$.

3 Implementation of Operators

Several general features apply to all operators and contribute to the generation of an highly parallel implementation:

- Use of **by-pair parallelism**, generated by splitting operations into independent computations over pairs of samples.
- Use of **by-chromosome parallelism**, generated by partitioning the operations by chromosome; as result, regions are only produced from input regions with matching chromosomes.

Partitioning by chromosome is suboptimal due to the difference in size of chromosomes, but it is a "natural" partition, as no region spans across chromosomes. The potential degree of parallelism depends on the product of the pairs of samples and of chromosomes[11]. However, parallelism is not sufficient for the domain-specific GMQL operations, as the number of regions that are candidate to result from a pair of samples within a chromosome is potentially very large (it can reach billions of candidates in practical cases).

In the first implementation we translated GMQL to PIG and we used simple user-defined Java functions for computing results by taking advantage of the ordering of regions along the reference genome, with limited parallelism; we adapted methods developed for temporal databases [25]. In the current implementation we use binning, i.e. the subdivision of the genome into much smaller, identical partitions or **bins** currently in use by databases of annotations of the UCSC Genome Browser [30] in order to speed up the search for portions of the genome that must be loaded within the same browser window. To the best of our knowledge, binning has not been used for parallelizing genomic operations; the use of time portions of equal size for temporal interval joins is studied in [3] in the context of map-reduce. We next focus on the implementation of domain-specific operations.

9. The keyword ANY can be used as maxAcc, and in this case no maximum is set (it is equivalent to omitting the maxAcc option); the keyword ALL stands for the number of samples of the operand, and can be used both for minAcc and maxAcc; these can also be expressed as arithmetic expressions built by using ALL (e.g., ALL-3, ALL+2, ALL/2); cases when maxAcc is greater than ALL are relevant when the input samples include overlapping regions.

10. The JaccardIntersect index is calculated as the ratio between the lengths of the intersection and of the union of the contributing regions; the JaccardResult index is calculated as the ratio between the lengths of the result and of the union of the contributing regions.

11. The number of chromosomes is fixed, e.g., they are 23 in computations on humans.

3.1 Join

Before discussing the join implementation, we discuss the clause evaluation order, the binning strategy, and the interaction between the binning strategy and the query-specific search space.

3.1.1 Evaluation Steps

As we discussed in Section 2.1, the order of execution of distal conditions influences the result; this depends on the fact that the min distance clause (`MD`) clause is not commutative with the greater distance clause (`GLE`) and with the stream clause (`UP/DOWN`); the less distance clause `DLE` is commutative with all other clauses, and stream and greater distal clauses are commutative with each other. Thus. the evaluation of a genometric predicate requires a sequence of 3 steps, where clauses within each step are commutative and each step can be missing:

- *Step 1* includes the `DLE` clause of the query and the stream and greater distal clauses which precede the `MD` clause; if a query-specific `DLE` clause is not present, then `DLE(Max)` is added, where `Max` denotes the maximum biological distance[12].
- *Step 2* includes the `MD` clause.
- *Step 3* includes the stream and greater distal clauses after the `MD` clause.

Examples. The genometric predicate:

```
DGE(500), MD(10), UP
```

produces the following three steps:

```
Step 1: DGE(500), DLE(Max)
Step 2: MD(10)
Step 3: UP
```

The genometric predicate: `DOWN, MD(10), DGE(2000), DLE(5000)` produces the following three steps:

```
Step 1: DOWN, DLE(5000)
Step 2: MD(10)
Step 3: DGE(2000)
```

Some simpler predicates may require a single step, e.g. `DGE(50000), UP, DLE(100000)` is mapped to Step 1.

3.1.2 Binning and Search Space

In cloud computing, efficient execution is achieved through parallelism; in genomic computing, such parallelism is achieved by means of **binning**, i.e., partitioning the genome into disjoint sections so that large computations can be split and assigned to bins. The process of binning splits every chromosome of the genome into several bins of equal size S; for each chromosome, bins are progressively numbered starting from 0 and the *i-th* bin spans from $S \times i$ to $S \times (i + 1) - 1$. For a given bin size S, a point placed at i bases from the chromosome start is assigned to the bin $b(i) = \lfloor i/S \rfloor$. Intervals between a left end l_i and a right end r_i are assigned to the bins between $b(l_i)$ and $b(r_i)$.

In order to effectively evaluate distal clauses, each anchor regions is associated with its **search space**, consisting of intervals of bins that may include matching regions of the experiment; search spaces are built according to the distal conditions of Step 1; it includes all potential matches, as Steps 2 and Step 3 are filters of the regions produced by Step 1. Consider an anchor region with left end l and right end r; let M be the maximum distance and let B_c denote the last bin of each chromosome c [13]. Then:

- If the clause is $LTE(d)$, then the search space is the interval of bins between $b(l - d)$ (excluding bins with $i < 0$) and $b(r + d)$ (excluding bins with $i > B_c$).
- If the clause is $LTE(d_1)$ and $GTE(d_2)$, with $d_1 > d_2$, then the search space is the two intervals of bins between $b(l - d1)$ and $b(l - d2)$ (excluding bins with $i < 0$) and between $b(r + d_2)$ and $b(r + d_1)$ (excluding bins with $i > B_c$).
- If the clause is $GTE(d_1)$, then the search space is the two intervals of bins between $b(l - M)$ and $b(l - d_1)$ (excluding bins with $i < 0$) and between $b(r + d_1)$ and $b(r + M)$ (excluding bins with $i > B_c$).

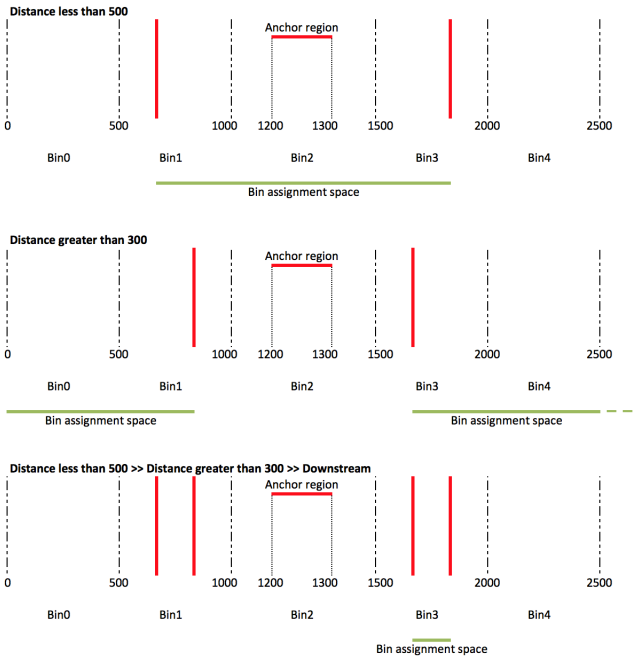

Fig. 5. Search spaces for three distal clauses, Step 1.

When the `UP/DOWN` clause is present, the search space is limited to the upstream/downstream directions of the genome. A representation of the search space for the anchor region as effect of the `DLE` and `DGE` clauses is shown in Fig. 5 (cases 1 and 2); the third case shows the effects of combining the `DLE`, `DGE` and `DOWN` clauses.

3.1.3 Evaluation of Distal Clauses in Step 1

This construction allows a parallel evaluation of join predicates. In particular, the following theorem holds due to the way in which search spaces are constructed:

Theorem 3.1. *The join predicate between an anchor region and any experiment region falling outside of its search space is false.*

12. If a query includes the clause `DLE(M1)` and $M1 > Max$, the clause is turned into `DLE(M)` by the execution engine; users can set the maximum biological distance of each query execution.

13. Given that chromosomes have different sizes, B_c is a specific number for each chromosome.

In addition, we would like to evaluate the Step 1 join predicate between given regions of the anchor and experiment in a given bin only, so as to generate the corresponding result region only once, avoiding duplicates. The following theorem provides a solution of this problem.

Theorem 3.2. *If the Step 1 predicate between an anchor region and an experiment region is true, it can be tested in a given bin, denoted as* **testing bin**.

Proof. We build the proof by considering four cases which exhaustively cover the relationships between anchor and experiment regions, and defining the testing bin for each of them.

- Assume that *the experiment is at the left of the anchor*, i.e. the experiment's left end is strictly less than the anchor's left end and the experiment's right end is less than or equal to the anchor's right end. Then, the testing bin is the experiment bin with greatest number (the one at the smallest distance from the anchor); the predicate can be true only if the portion of experiment region within the testing bin intersects with the search space. Some examples are shown in Fig. 6, where the testing bin is denoted by a thicker trait. The predicate can be true in case (a) (when the testing bin falls within the search space) and is false in case (b) (as the region is too close to the anchor) and (c) (as the region is too distant from the anchor).
- The case when *the experiment is at the right of the anchor* is symmetric; in such case, the experiment's right end is strictly greater than the anchor's right end and the experiment's left end is greater than or equal to the anchor's left end. Then, the testing bin is the experiment bin with the smallest number; also in such case, the predicate can be true only if the portion of experiment region within the testing bin intersects with the search space.
- Assume that *the experiment is included within the anchor*. Recall that by construction the search space either properly includes the anchor region or does not overlap with it. Thus, the experiment can satisfy the join predicate only if it intersects with the search space in anyone of its bins; conventionally, we may use as testing bin the experiment bin with the smallest number. This case is illustrated in Fig. 7.
- Finally, assume that *the anchor is included in the experiment*. Then, the anchor is at negative distance from the experiment, and again the search space either properly includes the anchor region or does not overlap with it; it follows that the join predicate between the region and the anchor can be true only if the search space includes the anchor. Conventionally, we may use as testing bin the anchor bin with the smallest number. This case is illustrated in Fig. 8.

□

Thanks to Theorem 3.2, at each bin B we evaluate Step 1 conditions just for those pairs of experiment and anchor regions such that B is their testing bin; thus, we either discard the pair of regions, or produce the resulting regions exactly once. This result is used by the parallel execution strategy which is next discussed.

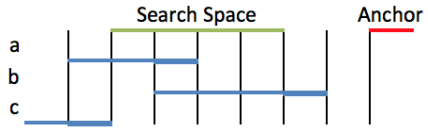

Fig. 6. Experiment regions at the left of the search space.

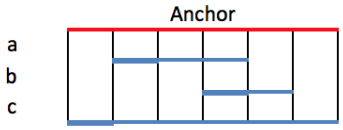

Fig. 7. Experiment regions enclosed within the anchor region.

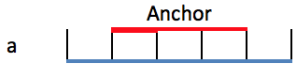

Fig. 8. Anchor region enclosed within the experiment region.

3.1.4 Join Execution Strategy in Flink and Spark

Fig. 9 illustrates the flow of Flink and Spark operators for implementing a join operation. We recall that joins require first to select the pairs of samples that need to be joined, using a metadata predicate, and then to compute the result regions, using a genometric predicate. The operation applies to two datasets, respectively called *anchor* and *experiment*; as a running example we consider the join with:

```
Step 1: DGE(140), DLE(500)
Step 2: MD(1)
Step 3: DOWN
```

computed on:

```
Anchor: Id, chromosome, start, stop
1 C1 150 160
1 C1 285 390
Experiment: Id, chromosome, start, stop
2 C1 10  20
2 C1 430 550
2 C1 750 780
```

Throughout the examples of this section, we do not consider strands; in reality, join predicates evaluation is defined only

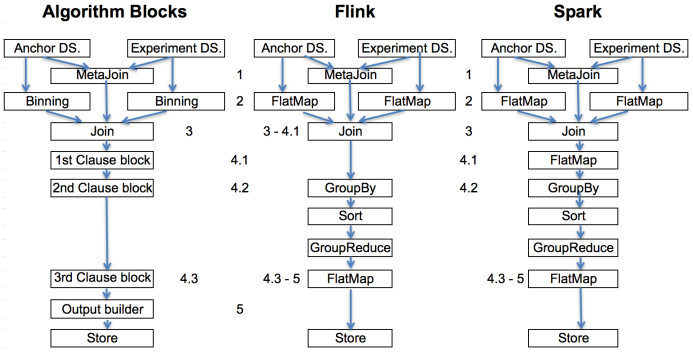

Fig. 9. Operators for encoding the Join algorithm in Flink and Spark.

between regions with compatible strand[14]. We also do not consider region values, they are carried along with each region and concatenated in the result[15].

- Block 1 (Metajoin) produces in output, for each anchor sample, the *join list* of the experiment samples that must be joined to it.
  Example. The join list of sample 1 is [2].
- Block 2 (FlatMap) is responsible of copying regions to the bins:
  - For every anchor region and bin b intersecting with the search space, it generates a copy of the anchor region for every bin b of the search space, by adding to it the attribute `Bin` (b) and the attribute `SBin` (the bin where the anchor region starts.)
  - For every sample of the join list and for every bin b intersecting with each experiment region, it generates a copy of the experiment region, by adding to it the attribute `Bin` (b) and the attributes `SBin` (the bin where the anchor region starts) and `EBin` (the bin where the anchor region ends.)

  Note that anchor regions are replicated at the bins of their search space, computed in this block, and experiment regions are replicated at the bins which intersect with them. The added attributes allow to test with a simple predicate if the current bin b is the testing bin of a given pair of anchor and experiment regions, based on the four cases of Theorem 3.2.
  Example. With a bin size $B = 100$, the first anchor region is copied to the bins $0, 2-6$, the second anchor region is copied to the bins $0-8$; the experiment regions is copied to the bins 0, $4-5$, and 7.
- Block 3 (Join) joins the anchor and experiments by `chrom` and `bin`. In this way, for any pair of anchor and experiment samples to be joined and for any of their anchor and experiment regions, all the relevant data are available at all bins, hence also at their testing bin. This operation is the most expensive, as it may join millions of regions to millions of regions; it is effectively computed by the `Join` operator, available in both frameworks. Performance depends on the bin size, as smaller bin size increases both replication and parallelism, therefore in Section 4.2 we study its optimal tuning as a function of data sizes and query parameters.
  Example. The following pairs are produced:

```
Bin  Chr   Id1 SB1 L1 R1    Id2 L2 R2
B0   C1    1 B1 150 160     2 10  20
B4   C1    1 B1 150 160     2 430 550
B5   C1    1 B1 150 160     2 430 550
B7   C1    1 B1 150 160     2 750 780
B0   C1    1 B3 285 390     2 10  20
B4   C1    1 B3 285 390     2 430 550
B5   C1    1 B3 285 390     2 430 550
B7   C1    1 B3 285 390     2 750 780
```

- Block 4.1 (Join in Flink, FlatMap in Spark) performs Step 1, by computing the distance between the regions in each row and then selecting only the rows of the testing bins where the distal conditions hold; testing bins are determined as indicated in the four cases of the proof of Theorem 3.2. This step is computed in parallel in each bin, in Flink is included in the `Join` of step 3, in Spark is a `FlatMap`.
  Example. The following pairs are produced:

```
Bin Chr   Id1 SB1 L1 R1    Id2 SB2 EB2 L2 R2 D
B4  C1    1 B1 150 160     2 B4 B5 430 550 270
B0  C1    1 B3 285 390     2 B0 B0 10 20 265
B7  C1    1 Bin3 285 390   2 B7 B7 750 780 360
```

- Block 4.2 (GroupBy, Sort, GroupReduce) performs Step 2, by selecting experiment regions based upon their minimal distance from anchor regions; it is implemented by the `GroupBy`, `Sort` and `GroupReduce` operators, but it requires data shuffling for collecting the experiment regions at nodes where sorting by distance and $top-k$ selection can be performed. We can reduce data shuffling with an alternative implementation, which adds a sort operation at each bin, producing at each bin the $top-k$ regions; these needs to be moved, while all other regions can be discarded. We discuss pros and cons of this alternative implementation in Section 4.2.2.
  Example. The following pairs are produced:

```
Bin Chr   Id1 SB1 L1 R1    Id2 SB2 EB2 L2 R2 D
B4  C1    1 B1 150 160     2 B4 B5 430 550 270
B0  C1    1 B3 285 390     2 B0 B0 10 20 265
```

- Block 4.3 (FlatMap) performs Step 3, by further reducing the filtered regions according to the distal conditions of Step 3. It uses the `FlatMap` operator.
  Example. In the example, the condition `DOWN` filters one pair, producing:

```
Bin Chr   Id1 SB1 L1 R1    Id2 SB2 EB2 L2 R2 D
B4  C1    1 B1 150 160     2 B4 B5 430 550 270
```

- Block 5 (FlatMap) is responsible of outputing the resulting pairs, by computing their sample identifier and their region coordinates according to the coordinate composition option and is executed together with block 4.3
  Example. We finally obtain the following result, where a new sample identifier is generated as a hash function of the identifiers of the two operands, and the resulting region is obtained by concatenating the operand regions:

```
Id          Chr  Start Stop
Hash(1,2)   Chr1 150   550
```

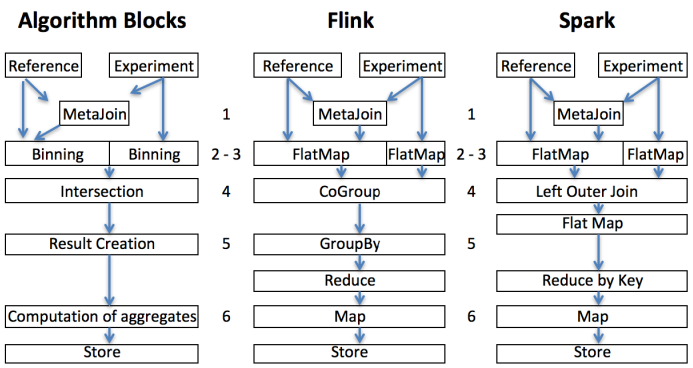

Fig. 10. Operators for encoding the Map algorithm in Flink and Spark.

3.2 Map

The encoding of this problem as a sequence of operations for Spark and Flink is shown in Fig. 10. The algorithm requires

14. Positive and negative strands are not compatible, and they are both compatible with undefined strands.

15. With big value sizes, it is convenient to project the values prior to Block 1 and then join them to resulting regions within Block 5.

to bin the two datasets, to group them by sample pair, chromosome and binning, to compute intersections within the bins, to compute aggregate functions, and output the results for each sample pair. The complexity of this problem grows quadratically with the sizes of the reference and experiment dataset. In the example, we count the experiment regions intersecting with reference regions; we consider:

```
Anchor: Id, chromosome, start, stop
1 C1 150 235
Experiment: Id, chromosome, start, stop
2 C1 10  230
```

- Block 1 (Metajoin) produces in output, for each reference sample, the *map list* of the experiment samples that must be mapped to it.
- Block 2 (Experiment Binning) is responsible of copying experiment regions to the bins. For every experiment region and bin b intersecting with the experiment, it generates a copy of the region for every bin b; only the attributes which are used by aggregate functions are copied.
  Example. With bins of size 100, the following copies are generated:
  ```
  Id Chr Bin Start Stop
  2  c1  0 10 230
  2  c1  1 10 230
  2  c1  2 10 230
  ```
  Note that a list of attribute values is generated, but no attribute value is needed for computing the `COUNT`.
- Block 3 (Reference Binning) is responsible of copying reference regions to the bins. For every reference region of a given sample, for every bin b intersecting with the reference, and for every experiment samples in its map list, a copy of the reference region is built, having as attributes the concatenation of `Id, Chr, Bin, Start, Stop` of the reference with the `Eid` of the experiment and with a new attribute `H` obtained by hashing all the attributes except the bin; this attribute is later used for assembling all copies relative to the same reference and experiment regions.
  Example. The following copies are generated:
  ```
  Id Chr Bin Start Stop Eid H
  1  c1  1 150 235 2 567
  1  c1  2 150 235 2 567
  ```
- Block 4 (LeftJoin) is responsible of computing a partial map within each bin. It joins references and experiment by `Eid`, `Chr` and `Bin`; if the join succeeds, it further selects resulting tuples by considering only the bins where either the reference region or the experiment region start (note that this bin exists and is unique by construction). At each selected pair, a portion of the aggregate function is computed. A new region is built, having as attributes the concatenation of `Chr, Bin` with `Rid, Start, Stop, H` of the reference and `EId, EStart, EStop, V` of the experiment; `V` stores the experiment values to be used by the aggregate functions (in the case of `Count`, it stores `1`.) If the join fails, thanks to the left join constructor, all the reference information is stored to the result, with null values stored for the experiment; in this way, all reference regions are correctly accounted.
  Example. The following copies are generated, and the second one is then filtered:
  ```
  Chr Bin RId Start Stop H EId EB EStart Estop V
  c1  1   1 150 235 567    2 c1 1 10 230 [1]
  c1  2   1 150 235 567    2 c1 2 10 230 [1]
  ```
- Block 5 (Assembling) is responsible of assembling all copies corresponding to the same reference and experiment at one node, through data shuffling; the operation is performed thanks to a `reduce phase` which uses the `Hash` attribute. Partial sums are performed for computing `COUNT`, and lists of attribute values are concatenated within a bag.
  Example. In the example, the two regions are reduced to one, as they have the same hash attribute. The following region is generated:
  ```
  Rid Chr Start Stop Val
  567 chr1 150 235 1
  ```
- Block 6 (Aggregating) is responsible of computing aggregate functions, by applying them to the bag of values built at block 5. This step does not apply to the running example.

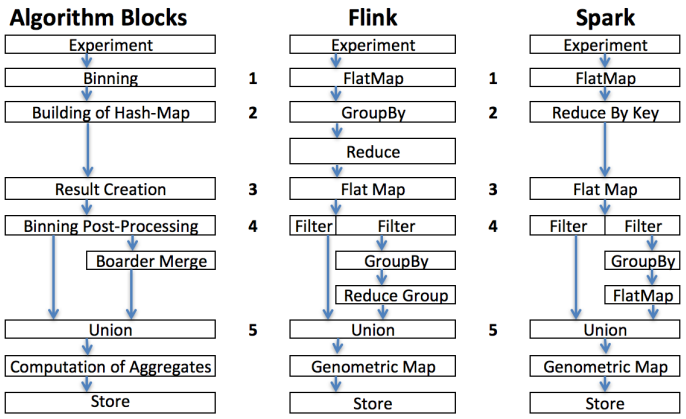

Fig. 11. Operators for encoding the Cover algorithm in Flink and Spark.

3.3 Cover

We discuss the computation of the `Histogram`, i.e. of the accumulation index, as discussed in Section 2.3; all other properties of the `Cover` are easily derived from that index. A sequential algorithm for solving this problem consists of scanning the genome from left to right and maintain the accumulation count. At every start of a region the count is incremented, and at every stop is decremented; the result is given by every consecutive pairs of region ends with a positive counter. In the following, we propose a parallel version of this algorithm which relies on partitioning the genome into bins. The operation flow is shown in Fig. 11.
Example. We show only three regions, with the peculiarity that the first region stops where the second region starts and that the third region intersects with two bins, whose size is set to 500.

```
Id, Chr, Start, Stop
1 chr1 154 237
1 chr1 237 450
1 chr2 460 600
1 chr2 580 700
```

- Block 1 (FlatMap) is responsible for the binning. For each region, it emits a new tuple for each bin it intersects. The output tuple contains the chromosome, the bin and a hash-map; in the hash map, we associate every region

start with +1 and every region stop with -1. In the case a region crosses the border between two bins, we split it into two contiguous regions; one from the start to the border and one from the border to the stop (if the regions spans for more than two bins, the same procedure is repeated).

```
Chr , Bin, HashMap[Int,Int]
chr1 0 {154->+1, 237->-1}
chr1 0 {237->+1, 450->-1}
chr2 0 {460->+1, 500->-1}
chr2 1 {500->+1, 600->-1}
chr2 1 {580->+1, 700->-1}
```

- Block 2 (GroupBy, Reduce) is responsible of grouping the output dataset of the previous block by chromosome and bin. Then, on each partition an associative function is applied by the Reduce, which builds a single tuple for each chromosome and bin containing a hash-map with all the starts and stops of the regions in the bin; notice that in the worst case, the size of this hash-map is the same as the the length of the bin, therefore it fits in memory.

```
Chr, Bin, HashMap[Int,Int]
chr1 0 {154->+1, 237->0, 450->-1}
chr2 0 {460->+1, 500->-1}
chr2 1 {500->+1, 580->+1, 600->-1, 700->-1}
```

- Block 3 (FlatMap) returns the list of produced regions, along with their accumulation value, with each region placed within a bin, thus creating a raw histogram:

```
Chr, Start, Stop, Count
chr1 154  450  1
chr2 460  500  1
chr2 500  580  1
chr2 580  600  2
chr2 600  700  1
```

- Block 4 (Filter) starts with two filters that separate the regions properly contained in the bins (left filter) from the regions overlapping with bins (right filter). The latter regions must be merged when they are adjacent and with the same count. This processing requires a GroupBy and a ReduceGroup. In the specific example, the right filter is applied to the regions of chromosome 2, producing the region:

```
Chr, Start, Stop, Count
chr2 460 580 1
```

- Finally, Block 5 (Union) performs the union of the regions separately produced, Block 6 computes the aggregate (if any) using a Genometric Map operation and then (DataSink) writes them to the disk; it generates:

```
Chr, Start, Stop, Count
chr1 154 450 1

chr2 460 580 1
chr2 580 600 2
chr2 600 700 1
```

3.4 Code Examples in Flink and Spark

GMQL is supported by in the context of a large project for genomic data management; in particular, GMQL queries are interpreted by a Scala compiler which produces either a Flink or Spark execution plan by invoking the respective APIs. The full translation and excution architecture of the system is outside the scope of this paper.

In this section, we describe Scala-like pseudocode for the implementation of a portion of the Map operation in Flink and Spark. Specifically, Fig. 12 shows the Flink implementation of Block 4 of Section 3.2, where a `coGroup` operation

```
0   //Intersection phase – Block 3
    val coGroupResult = binnedRef // expID, bin, chromosome
1     .coGroup(binnedExp).where(1,3,5).equalTo(2,1,3){
2     (reference , experiments, out) => {
3       val refCollected : List[RefRegion] = references.toList
4       for(ref <- refCollected){
5        for(exp <- experiments){
6         if ( /* space overlapping */
7          (ref.start < exp.stop && exp.start < ref.stop)
8           && /* same strand */
9          (ref.strand.equals('*') || exp.strand.equals('*') ||
10          ref.strand.equals(exp._7))
11          && /* first check */
12         (ref.bin.equals( ref.start/binsize ) ||
13          exp.bin.equals(exp.start / binsize))
14        ) {
15         ref.count += 1
16         ref. expData = ref. expData :+ exp._8
17       }
18      }}
19  // Output ref region with the exp's values(intersected).
20      refCollected.foreach((r) => {
21            val extraData = r.expData.reduce((a,b) =>
22       a.zip(b).map((p) => p._1 ++ p._2))
23            val res = (r.ID, pr.Chrom, r.start, r.stop, r.strand,
24       r.values, extraData, r.count, r.RUniqueID)
25            out.collect(res)
26          })
27        }
28       }
```

Fig. 12. Pseudocode for Block 4 of the Map algorithm in Flink.

is applied. Inside the operation, invoked at line 2, we iterate on all the regions of one reference sample (line 5) and then on all the regions of the experiment samples (line 6); if an overlap is found along compatible strands (lines 8-15), then the counter of regions is incremented (line 16). Then, regions are outputed by means of a Flink collector operation (line 22). Note that the Scala code embedding Flink operations is executed in each node's core in parallel for every bin; given that we control the bin size, we make sure that the memory size at each node is not exceeded.

The Spark implementation has a very similar pseudocode, based on `CoGroup` and `Reduce`; in the binning phase (Block 2), the fields of the reference and experiment are organized as key/value pairs, where the key is the tuple `(Eid, Chr and Bin)` and the value is the rest of the fields; then, line 2 is replaced in Spark implementation with `.cogroup(binnedExp).flatMap` (i.e. the cascade of two Spark operations `coGroup` and `flatMap`), without the need of a collect command. However, Spark allows for a second implementation, based on the `LeftJoin` operator, that supports persistence, and therefore works with any bin size[16]. In general, we kept the translation strategy to Flink and Spark aligned, so as to allow for a fair performance comparison (see Section 4.3); however, we also support alternative implementations, so as to selectively take advantage of the different features of the two frameworks.

Figure 13 shows the Flink implementation of Block 5 of Section 3.2, with a `grouBy` operation followed by a `reduce` operation; as discussed in Section 3.2, this phase is required to reduce all the regions with the same hash attribute to just one region. Spark supports a `reduceByKey` operator, that

16. The two implementations have no significant difference in performance; `leftJoin` is currently missing in Flink.

```
0
    // boarder regions assembly by ref region Unique ID. Block 4
1   val aggregationResult = coGroupResult.groupBy(8)
2     .reduceGroup((a, out) => {
3       out.collect(
4         a.reduce(
5           (r1,r2) => {
6             val out = (r1.ID, r1.Chrom, r1.start, r1.stop,
7               r1.strand, r1.values,
8               r1.extra.zip(r2.extra).map((a) => a._1 ++ a._2),
9               r1.count + r2.count, r1.RUniqueID)
10            out
11          }
12        )
13      )
```

Fig. 13. Pseudocode for Block 5 of the Map algorithm in Flink.

ie equivalent to the cascade of `groupBy` and `reduce`[17].

4 Experiments

4.1 Experimental Setup

We performed our experiments on the Amazon Cloud, in most cases with a small cluster of one master and five slaves (m3.x2large); we used Flink-0.9.1[18] and Spark 1.52 [19]. We concentrate on Join and Cover (Map is very similar to Join). The datasets are: `TSS` for references (one sample of 131780 short regions from UCSC transcription start sites), and `NARROW` for experiments (1999 samples from Encode Narrow Peak Dataset, which has a total of 143 million regions and an average of 71915 regions per sample).

4.2 Join Optimization and Tuning

4.2.1 Optimal Bin Size

As discussed before, the rationale of binning is to reduce the number of regions to be considered within each bin, instead of computing chromosome-wide cross product; however, regions that cross the binning borders must be replicated. Large bins reduce replication of regions, but they lead to producing and matching many pairs of regions within each bin; conversely, short bins increase replication and therefore the generation of matching regions that should not be produced in output.

The choice of a good bin size depends on many factors:

- Physical characteristics of the cluster: if the executors have a large amount of memory, then larger bins can be used, as the cost of computing larger cross products in-memory is generally less than the cost of shuffling regions.
- Total number of regions: when regions increase in number, smaller bins are needed in order to avoid huge cross products;
- Average region length: when regions are longer than bins, they are going to produce many replicates, thus increasing the cost of data shuffling and the number of useless tests.

17. Thus, in the the pseudocode of Fig. 13, lines 2-5 should be substituted by `coGroupedResult.reduceByKey(`.

18. flink-0.9.1/bin/yarn-session.sh -n 5 -jm 768 -tm 10752 -s 4; yarn.heap-cutoff-ratio = 0.15.

19. default EMR parameters. spark-submit –master yarn –deploy-mode client –num-executors 20 –executor-memory 5G.

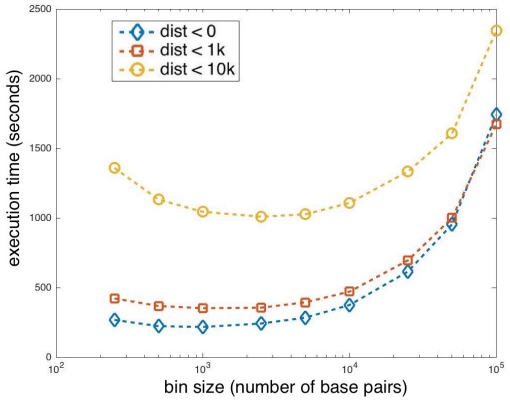

Fig. 14. Execution time of Join as a function of bin size in logarithmic scale for Flink.

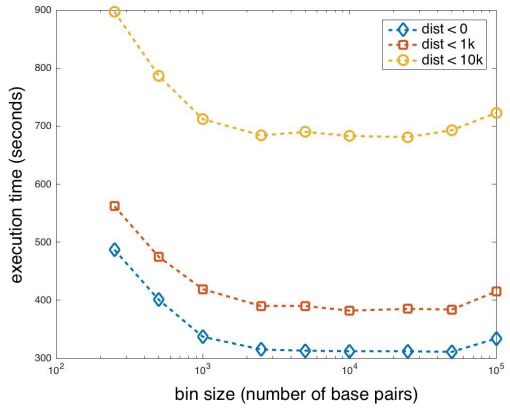

Fig. 15. Execution time of Join as a function of bin size in logarithmic scale for Spark.

Figure 14 shows the execution time of the join using the Flink engine, for different choices of the distal predicate (note that if the distance is less than 10K bases many more resulting regions are produced w.r.t. distances of 1K or of zero bases, yielding to longer execution times). In these cases, bin size between 1K and 10k bases are optimal. Figure 15 shows the same experiment using the Spark engine; in these cases, bin size close between $0.5 \times 10K$ and $5 \times 10K$ are optimal.

Figure 16 shows the execution time of the cover using the Flink engine, for different choices of the minimum and maximum accumulation indexes; note that the performance does not depend on accumulation indexes: once the histogram is computed (Block 3), then the extraction of result regions (Blocks 4-6) has very similar costs for any choice of accumulation indexes.

4.2.2 Data Shuffling and Ordering with MinDistance

As explained in Section 3.1.2, each reference region is replicated to all the bins whose distance from the anchor region is less than the query constant; this is implemented by Block 2 of Section 3.1.4, and may generate a large number of regions satisfying the join condition at each bin, especially when the query constant is set to `Max`, the maximum biological region length. This in turn may cause a lot of data shuffling

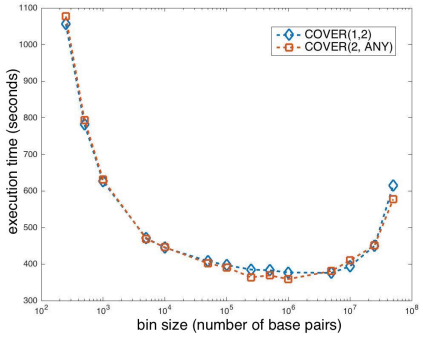

Fig. 16. Execution time of Cover for Flink.

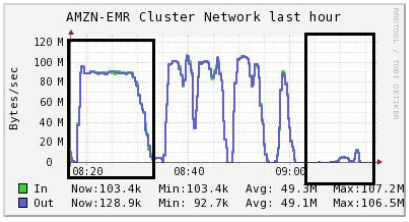

Fig. 17. Comparison of data shuffling strategies.

for extracting the top-k regions of a minimal distance join; to reduce overhead, we suggested an alternative implementation which adds an intermediate sorting step (see Block 4.2 of Section 3.1.4). Figure 17 shows the network statistics (retrieved using Ganglia[20]); the area in the first rectangle corresponds to the standard execution and shows heavy data shuffling, while the area in the second rectangle refers to the alternative implementation which includes the intermediate sorting step and shows a much lighter data shuffling. We plan to further study shuffling optimizations, along the work of [51].

4.3 Framework Comparison

Flink and Spark are both general-purpose data processing platforms and top level projects of the Apache Software Foundation (ASF). They have a wide field of applications and are usable for dozens of big data scenarios. They support several extensions, e.g. to SQL-like queries (Spark: Spark SQL, Flink: MRQL), graph processing (Spark: GraphX, Flink: Spargel (base) and Gelly(library)), machine learning (Spark: MLlib, Flink: Flink ML) and stream processing (Spark Streaming, Flink Streaming). Both are capable of running in standalone mode, yet most usage occurs on top of Hadoop (YARN, HDFS). For what concerns their differences[21]:

- Flink uses dataset variables and is optimized for cyclic or iterative processes by using iterative transformations on collections. This is achieved by an optimization of join algorithms, operator chaining and reusing of partitioning and sorting. However, Flink is also a strong tool for batch processing. Flink streaming processes data streams as true streams, i.e. data elements are immediately "pipelined" though a streaming program as soon as they arrive. This allows to perform flexible window operations on streams.
- Spark is based on resilient distributed datasets (RDDs), (mostly) in-memory data structures giving to Spark the power of functional programming paradigms. Spark is capable of big batch calculations by binning memory; Spark streaming wraps data streams into mini-batches, i.e. it collects all data that arrives within a certain period of time and then runs a regular batch program on the collected data. While the batch program is running, the data for the next mini-batch is collected.

20. http://ganglia.sourceforge.net/
21. A thorough comparison of Flink and Spark can be found in http://stackoverflow.com/questions/28082581/what-is-the-differences-between-apache-spark-and-apache-flink.

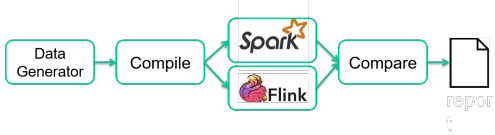

Fig. 18. Comparing Flink and Spark execution engines.

4.3.1 Approach

Figure 18 illustrates our approach; the *data generation* module produces the input for several runs, then for each input we invoke the *compiler* that produces two versions of Scala code embedding calls to both Flink and Spark (along the method discussed in section 3.4), then the two codes are invoked, then the results are *compared* [22], testing that results are identical and extracting as well the execution time for each run.

4.3.2 Performance and Bin Size

Join execution times of Flink and Spark are compared in Fig. 19. For small bin sizes and restrictive clauses (less matching regions) Flink has better performance, whereas for large bin sizes and more permissive clauses (more matching regions) Spark has better performance. In the Cover, pipeline parallelism results in faster execution times for Flink; Fig. 20 shows that the cover execution times are rather similar for an optimal choice of the bin size, but Flink outperforms Spark for either small or large bins. These results are coherent with our findings in [12], although more accurate.

A comparison of execution stages in Flink and Spark is also presented in [12]; Flink supports pipeline parallelism that may grant better performances to Flink than to Spark in some cases; in general, Flink pipeline parallelism is blocked by operations such as `Sort` or `Collect`[23].

4.4 Performance Scaling with more AWS Nodes

We considered the execution of several joins with the same reference and an increasing number of samples, using the Flink execution engine, and scaling the size of the AWS network from 5 to 10, 15, and 19 nodes[24] Fig. 21 shows that

22. We developed about 150 short programs which test every feature of GMQL.
23. http://data-artisans.com/apache-flink-new-kid-on-the-block/
24. Our AWS grant allows for configurations of at most 20 nodes.

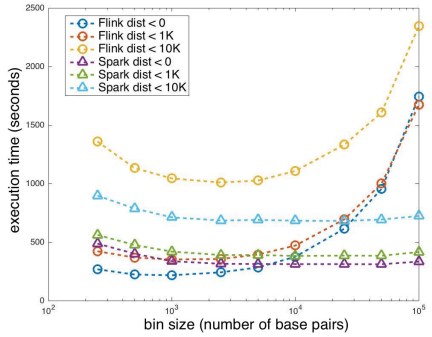

Fig. 19. Comparison of Join execution times as a function of bin size and join clause in logarithmic scale for Flink and Spark.

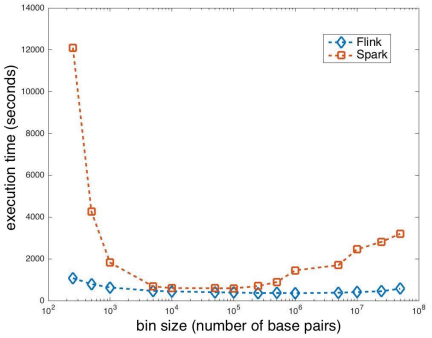

Fig. 20. Comparison of Cover execution times as a function of bin size in logarithmic scale for Flink and Spark.

the performance improves with largest networks with up to 500 samples, but in the case of 1000 samples the performance decreases; this is due to an excess of communication overhead with the addition of nodes.

Table 1 shows the unit cost per query using AWS, which charges a fixed price per node and time unit[25]. We note that the cost per second of execution increases with the growth of the network size, but this is compensated by a decreased execution time (see Fig. 21). However, with 1000 samples and 19 slave nodes, we note both an increase of cost and of execution time. An *elastic system* could benefit of constant monitoring of execution times, by dynamically shutting down nodes when such behavior occurs[26].

25. The hour cost of an Elastic Map Reduce (EMR) instance is the sum of the cost of nodes and the cost of EMR service. In our case we used M3.XLarge instances which cost 0,266$/h plus 0,070$/h for EMR service, yielding a total of 0,336$/h; therefore, a cluster of 6 nodes (5 slaves + 1 coordinator) of EMR costs 2,016$/h. Instances are paid hourly but we run batches of several experiments, thus we can redistribute execution costs to each query weighted by the execution time in seconds. For example, if a query takes 20 seconds to execute, then the cost of that query is 0,0112$.

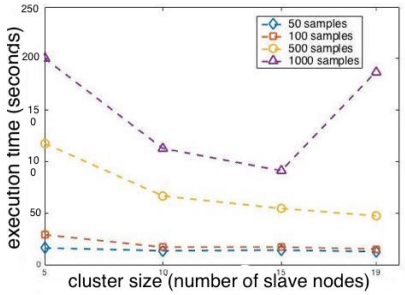

Fig. 21. Scaling of execution time by increasing the number of AWS nodes.

# of samples	6 nodes	11 nodes	16 nodes	20 nodes
50 samples	0,0112	0,0205	0,0299	0,0373
100 samples	0,0168	0,0236	0,0403	0,0467
500 samples	0,0644	0,0770	0,0896	0,0933
1000 samples	0,1120	0,1232	0,1419	0,3640

TABLE 1
Unit cost of execution per sample with different cluster sizes.

5 Related Work

Several organizations are considering genomics at a global level. *Global Alliance for genomics and Health*[27] is a large consortium of over 200 research institutions with the goal of supporting voluntary and secure sharing of genomic and clinical data; their work on data interoperability is producing a conversion technology for the sharing of data on DNA sequences and genomic variation [23]. *Google* recently provided an API to store, process, explore, and share DNA sequence reads, alignments and variant calls, using Google's cloud infrastructure [24].

We compare our work with recent papers on genomic data management. Works by Röhm and Blakeley [40], by Tata, Patel et. al. [47], and by Bafna et al. [11], address the querying of genomic data using either SQL (in the former case) or SQL extensions (in the latter two cases). [40] highlights the performance bottlenecks of conventional SQL optimization when dealing with domain-specific functions and parallelization. Tata, Patel et. al. developed *Periscope* [47], a system supporting matching operators over DNA sequences, encoded as character strings; they report fast execution times. Several domain-specific extensions are embedde within SQL in [11], to overcome SQL limitations in expressing genomic computations. SCORE [15] supports the embedding of user-defined functions within SQL and generates highly parallel execution plans on clusters.

Other works have proposed the embedding of query processing functions within libraries that can be integrated within programs [14], [38]. In particular, [38] presents a rather elegant mathematical formalism, based on set algebra, delivered as the *Genomic Region Operation Kit* (GROK) library. In comparison, GROK supports lower-level abstractions than GMQL and some low-level operations (e.g., flipping regions) that are not directly supported by GMQL, but they must be embedded into C++ programming language code. Furthermore, high-level declarative operations, such as `JOIN` and `MAP`, can be encoded in GROK, but they must be invoked from line editors or C++ programs. GROK shows excellent performance on desktop systems, but it is unsuitable for parallelization and does not deal with metadata.

26. However, such query-specific elastic system cannot be easily developed in AWS, as configuration switching is time expensive.

27. http://genomicsandhealth.org/

Several other tools focus on specific data formats or tackle specific needs and processing requirements. Among them, *BEDTools* [39] and *BEDOPS* [34] apply to the BED format, i.e. to a format based on regions; they either process regions of individual samples or compare regions of two samples, therefore multi-sample processing requiring verbose scripts. A functional comparison of these tools with GMQL is published as supplemental material to [33], where we illustrate how biologists would comparatively build a long query with the three approaches. BEDTools and BEDOPS can be used from within software environments for bioinformatics (e.g., *BioPerl*, *BioPython*, *R* and *Bioconductor*), but are not designed for cloud computing.

A recent work by Nordberg et al. [36] presents *BioPig*, a set of extensions for specific NGS analysis tasks to the *Pig Latin* data processing language [37]. BioPig includes three modules that can be used in the early phase of NGS data analysis for processing the raw read data files produced by NGS machines. *SeqPig* [44] is another collection of similar modules to manipulate, analyze and query sequencing datasets. The work by Weiwiorka et al. [53] presents analogous analysis tasks implemented on *Apache Spark* [54]. All these works are complementary to our, as they apply on NGS read data instead of on processed data.

One approach to efficiently compute interval joins in the Hadoop ecosystem frameworks uses indexed structures [19], [13]; this approach is not feasible in our application, where we have to directly read from experimental data files. An alternative way, more suitable for GMQL, is to implement algorithms which partition the join operands in order to speed up the evaluation. In [16] the authors propose an algorithm based on data binning; our algorithms differ from their proposal in the way we check the intersection and avoid output duplicates. Recently, Afrati et al. [3] further analysed binning-based algorithm in order to assess computation bounds. One other main difference with all of those approaches is that our algorithms are implemented upon frameworks at a higher than MapReduce. For Spark, similar problems have been addressed by the GeoSpark [49] and Magellan [32] projects. To the best of our knowledge, spatial joins have not been tackled by the Flink community.

6 Conclusions

In this paper, we described three domain-specific operations of GMQL; we introduced binning as a general approach to the parallelization of genomic operations; we identified the trade-offs due to the bin dimensions; we proved that, although multiple bins may carry the result of region comparison, the result can be safely extracted from a single bin; and we explained how the binning logic is implemented by using Flink and Spark.

Experiments demonstrate that the bin size is a critical parameter for the overall performance of domain-specific oparations; based on experiments, we will adopt a bin size of 10K bases for map and join and of 1M bases for cover both in Spark and Flink; we will also continue the development of GMQL on both systems, as the experients do not demonstrate a clear winner between the two engines.

We also showed the scaling of performance in a multi-node cloud computing network; adding computing power increments the performance but at an increasing cost per sample; moreover, above a given threshold, an increase in the number of nodes causes a loss in performance, given the complexity of multi-node computation.

The experiments demonstrate that domain-specific GMQL operations scale extremely well when challenged by very large datasets; therefore, GMQL is an ideal formalism to cope with big queries of today's and tomorrow's genomic computing. We are using GMQL in advanced biological research, for understanding how *topological domains*, i.e. recently discovered functional subdivisions of the genome [17], include genes which are highly expressed in either normal or tumor cells. This problem is addressed by a very simple GMQL program, which computes a MAP of RNAseq expression data for highly expressed genes over the regions corresponding to each topological domain; we use TCGA [52] expression datasets, having up to one thousand samples for each normal tissue or cancer types; each such query runs in about 3 minutes, hence a complete pipeline iterating over 20 tissues and normal vs tumor types runs in about 2 hours.

We are planning to turn GMQL into an incubated project within Apache, so as to provide a strong community of users and developers. We recently deployed our system in a public network at Cineca[28]. We also plan to deliver custom services and to provide access to a large repository of public data, constructed by integrating and curating data from ENCODE [21], TCGA [52], 1000 Genomes Project [1] and other sources.

Acknowledgments

We acknowledge the essential contributions of Gianpaolo Cugola, Vahid Jalili, Marco Masseroli, Matteo Matteucci, Heiko Muller, and Fernando Palluzzi, who designed GMQL with us. Research is supported by the ERC Advanced Grant *GeCo* (Data-Driven Genomic Computing), by the PRIN project *GenData 2020* (funded by the Italian Ministry of the University and Research), and by a grant from Amazon Web Services.

References

[1] 1000 Genomes Consortium, An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491, 56-65, November 2012.
[2] Adam. http://www.bdgenomics.org/
[3] F. Afrati and J. Ullman, Bounds for Overlapping Interval Join of Map Reduce. *Workshop Proceedings, EDBT/ICDT*, 2015.
[4] A. Alexandrov et al. The Strathosphere platform for big data analytics. *VLDB Journal* 23(6), 939-964, 2014.
[5] Anonymous paper, Accelerating Bioinformatics Research with New Software gor Big Data to Knowledge (BD2K), Paradigm4 Inc., 2015 downloaded from: www.paradigm4.com).
[6] Apache Flink. http://flink.apache.org/
[7] Apache Lucene. http://lucene.apache.org/core/
[8] Apache Pig. http://pig.apache.org/
[9] Apache Spark. http://spark.apache.org/
[10] Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[11] V. Bafna et al. Abstractions for genomics.*Commun. ACM*, 56(1):83-93, 2013.
[12] M. Bertoni, S. Ceri, A. Kaitoua, P. Pinoli. Evaluating cloud frameworks on genomic applications. In *Proc. IEEE Conference on Big Data Management 2015*, Santa Clara, CA, 2015.

28. http://www.bioinformatics.deib.polimi.it/GMQL/interfaces/

[13] J. Buck et al. SciHadoop: Array-based Query Processing in Hadoop. In *Super Computing 2011*, Seattle, WA, 2011.
[14] M. Cereda et al. GeCo++: a C++ library for genomic features computation and annotation in the presence of variants. *Bioinformatics*, 27(9):1313-1315, 2011.
[15] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets, *Proc. VLDB 2008*, pp. 24-30.
[16] B. Chawda. Processing Interval Joins On Map-Reduce. In *Proc. EDBT 2014*, pp. 463-474.
[17] Dixon J. R. et al. Topological domains in mammalian genomes identified by analysis of chromatin interactions. Nature, 485:376-380 (2012).
[18] J. Ekanayake et al. MapReduce for data intensive scientific analyses. In *Proc. IEEE eScience*, 277-284, 2008.
[19] A. Eldawy, M. F. Mokbel A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data. *Proc. Very Large Data Bases*, Riva del Garda, Trento, Italy, August 2013.
[20] S. Ewen et al., Spinning fast iterative data flows. *PVLDB 2012*, 1268-1279.
[21] ENCODE Project Consortium. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57-74, 2012.
[22] Galaxy. http://galaxyproject.org/
[23] Global Alliance Genomics API. http://ga4gh.org/#/documentation
[24] Google Genomics Could Platform. https://cloud.google.com/genomics/
[25] H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *Proc. IEEE ICDE*, 336-344, 1991.
[26] P.H. Guzzi, G. Agapito, and M. Cannataro, coreSNP: Parallel Processing of Microarray Data, *IEEE - Transactions on Computers*, vol. 63 no. 12, Dec. 2014.
[27] T. Hey at al. Jim Gray on eScience: a Transformed Scientific Method, In *The fourth paradigm. Data-intensive scientific discovery*, Microsoft Research, Redmond, WA, XVII-XXXI, 2009.
[28] Hadoop 2. http://hadoop.apache.org/docs/stable/
[29] F. Hueske et al. Opening the black boxes in dataflow optimization. *PVLDB 2012*, 1256-1267.
[30] W.J. Kent, The human genome browser at UCSC. *Genome Res.*, 2002 Jun;12(6):996-1006.
[31] C. Kozanitis et al. Using Genome Query Language to uncover genetic variation. *Bioinformatics* 30(1):1-8, 2014.
[32] http://spark-packages.org/package/harsha2010/magellan.
[33] M. Masseroli, P. Pinoli, F. Venco, A. Kaitoua, V. Jalili, F. Paluzzi, H. Muller, S. Ceri. GenoMetric Query Language: A novel approach to large-scale genomic data management. *Bioinformatics*, 2015, doi: 10.1093/bioinformatics/btv048.
[34] S. Neph, et al. BEDOPS: high-performance genomic feature operations. *Bioinformatics*, 28(14):1919-1920, 2012.
[35] NIH National Human Genome Research Institute, "DNA Sequencing Costs." http://www.genome.gov/sequencingcosts/
[36] H. Nordberg et al. BioPig: a Hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, 29(23):3014-3019, 2013.
[37] C. Olston et al. Pig Latin: A not-so-foreign language for data processing. *ACM-SIGMOD*, 1099-1110, 2008.
[38] K. Ovaska et al. Genomic Region Operation Kit for flexible processing of deep sequencing data. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 10(1):200-206, 2013.
[39] A. R. Quinlan and I. M. Hall. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841-842, 2010.
[40] U. Röhm and J. Blakeley. Data management for high-throughput genomics. In *Proc. CDIR*, 1-10, 2009.
[41] C. E. Romanoski, C. K. Glass, H. G. Stunnenberg, L. Wilson, and G. Almouzni, "Epigenomics: Roadmap for regulation," *Nature*, vol. 518, no. 7539, pp. 314-316, 2015.
[42] S. C. Schuster, Next-generation sequencing transforms today's biology. *Nat Methods.*, vol. 5, no. 1, pp. 16-18, 2008.
[43] SciDB. http://www.scidb.org/
[44] A. Schumacher et al. SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop. *Bioinformatics*, 30(1):119-120, 2014.
[45] J. Shendure, and H. Ji, Next-generation DNA sequencing. *Nat Biotechnol.*, vol. 26, no. 10, pp. 1135-1145, 2008.
[46] K. Shvachko et al. The Hadoop distributed file system. In *Proc. MSST*, 1-10, 2010.
[47] S. Tata et al. Periscope/SQL: Interactive exploration of biological sequence databases. In *Proc. VLDB*, 1406-1409, 2007.
[48] R. C. Taylor. An overview of the Hadoop MapReduce HBase framework and its current applications in bioinformatics.*BMC Bioinformatics*, 11(12):S1, 2010.
[49] J. Yu, GeoSpark: "A Cluster Computing Framework for Processing Large-Scale Spatial Data". *Proc. ACM SIGSPATIAL GIS 2015*, Seattle, WA, USA November 2015.
[50] R. Xin et al. Shark: SQL and Rich Analytics at Scale. In *Proc. ACM-SIGMOD*, June 2013.
[51] W. Yu, Y. Wang, X. Que, C. Xu, Virtual Shuffling for Efficient Data Movement in MapReduce, *IEEE Transactions on Computers*, vol.64, no. 2, pp. 556-568, Feb. 2015, doi:10.1109/TC.2013.216.
[52] J. N. Weinstein et al. The Cancer Genome Atlas Pan-Cancer analysis project. *Nat Genet.*, 45(10):1113-1120, 2013.
[53] M. S. Weiwiorka et al. SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, 30(18):2652-2653, 2014.
[54] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX*, 15-28, 2012.

Abdulrahman Kaitoua received his B.E. degree with high distinction in Computer Systems Engineering from Mamoun Private University (MUST), Syria, in 2009. He received his Masters in Electrical and Computer Engineering department from the American University of Beirut (AUB), Lebanon, in 2013. He is currently a Ph.D. student in Information Technology at Politecnico di Milano. He is doing research in the areas of cloud computing, data mining, and bioinformatics. His research interests include cloud computing, software engineering, data mining, distributed computing, and big data processing.

Michele Bertoni got his BSc in 2013 and his MSc in 2015, both summa cum laude, in Engineering of Computing Systems at Politecnico di Milano. He was involved in GenData 2020 project while working on his master thesis, when he designed and developed the GMQL algorithms using Apache Flink framework. His main interests spread over the area of databases, big data, data mining and Web services.

Stefano Ceri is Professor at the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano. He was Visiting Professor at the Computer Science Department of Stanford University (1983-1990), Chairman of the Computer Science Section of DEI (1992-2004), Director of Alta Scuola Politecnica (ASP) of Politecnico di Milano and Politecnico di Torino (2010-2013). In 2008 he has been awarded an advanced ERC Grant on Search Computing (2008-2013). He is co-founder (2001) of WebRatio (http://www.webratio.com/). focused on building query and data analysis systems for genomic data as produced by fast DNA sequencing technology. He is the recipient of the ACM-SIGMOD "Edward T. Codd Innovation Award" (2013), and an ACM Fellow and member of the Academia Europaea.

Pietro Pinoli received the BS/MS degree in Computer Science and Engineering in 2012 from Politecnico di Milano, Italy. He is currently a PhD candidate at the DEIB of Politecnico di Milano, Italy, and a member of the DEIB Bioinformatics Group. He is a collaborator of the Istituto Europeo di Oncologia (IEO). His research interests include databases, big data, machine learning and bioinformatics.