Multi-Dimensional Genomic Data Management for Region-Preserving Operations

Olha Horlova
Politecnico di Milano
olha.horlova@polimi.it

Abdulrahman Kaitoua
DFKI & TU-Berlin
abdulrahman.kaitoua@dfki.de

Volker Markl
DFKI & TU-Berlin
volker.markl@tu-berlin.de

Stefano Ceri
Politecnico di Milano
stefano.ceri@polimi.it

***Abstract*—In previous work, we presented GenoMetric Query Language (GMQL), an algebraic language for querying genomic datasets, supported by Genomic Data Management System (GDMS), an open-source big data engine implemented on top of Apache Spark. GMQL datasets are represented as genomic regions (i.e. intervals of the genome, included within a start and stop position) with an associated value, representing the signal associated to that region (the most typical signals represent gene expressions, peaks of expressions, and variants relative to a reference genome.) GMQL can process queries over billions of regions, organized within distinct datasets.**

In this paper, we focus on the efficient execution of region-preserving GMQL operations, in which the regions of the result are a subset of the regions of one of the operands; most GMQL operations are region-preserving. Chains of region-preserving operations can be efficiently executed by taking advantage of an array-based data organization, where region management can be separated from value management. We discuss this optimization in the context of the current GDMS system which has a row-based (relational) organization, and therefore requires dynamic data transformations. A similar approach applies to other application domains with interval-based data organization.

***Index Terms*—Big data processing, data management, cloud computing, genomic computing.**

I. Introduction

Thanks to Next Generation Sequencing (NGS), the new technology for reading the DNA, data production for genomics has exploded; by 2025, genomic data will exceed in size (by about two orders of magnitude) data available on YouTube or other Web-based video sources [22]. Most bio-informatic efforts are concerned with primary and secondary data analysis, focused on processing raw data from DNA sequencing machines and on extracting signals from them. Typical signals describe DNA regions which show variants from the reference genome (e.g., germline or somatic mutations) or which are most expressed in specific experimental conditions (i.e., where RNA is mostly produced or where proteins bind to the DNA). Genomic signals in such region-based format are also called *processed datasets* and are collected within huge repositories, open to the public for secondary research.

With the growth and diversity of available genomic signals, bio-informatics is now also targeting *tertiary data analysis*, concerned with data analysis and mining for processed datasets (see Fig. 1). Along this trend, we are currently developing a new, holistic approach to genomic data modelling and querying. Our approach is based on GenoMetric Query Language (GMQL) [14], a high-level query language for tertiary data analysis, concerned with filtering, aggregating and joining heterogeneous genomic signals in order to answer biological and clinical queries over large repositories of tertiary data. GMQL data model assumes a region-based data organization, where the

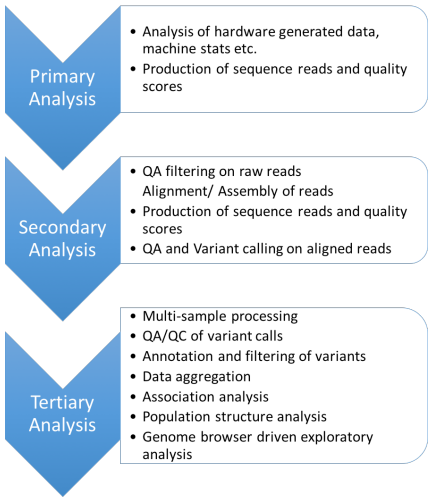


Fig. 1. Phases of genomic data analysis

elementary genomic information is the signal associated to a given region. Regions produced by the same experiment share the same data format and are assembled in a GMQL **sample**; several samples are assembled in a GMQL **dataset**.

GMQL has been designed with two objectives. The first one is to improve the expression of biological queries, by means of new abstractions for tertiary data management; examples of typical queries concern the search of regions of the genome (e.g. genes) with high number of mutations, or of genes that are under- or over-expressed in specific conditions, e.g. in normal and tumor cells; Section V-E shows a complete example of use of GMQL[1]. The second one is the efficient management of massive data volumes which are required to cope with the explosion of processed data. For this second reason, we implemented GDMS, a Genometric Query Management System, which translates GMQL queries to operations of general-purpose cloud-based engines [11]. GDMS was initially implemented on top of Apache Spark [25], Apache Flink [5] and SciDB [2] - Spark and Flink are classic data engines for Hadoop-based clouds with a row-based data model, SciDB is a multi-dimensional database designed for scientific data analysis. After an initial evaluation of these engines, we focused on one of them: the current GDMS implementation, described in [11], uses Spark. Our choice was influenced by our domain-specific comparative analysis of Flink and Spark [7] and of Spark and SciDB [8].

[1]For an account of biological queries supported by GMQL, see the Biological Examples in the Documentation Section at http://www.bioinformatics.deib.polimi.it/geco/?try.

In this work, we present an important improvement of GDMS in light of a distinctive property of genomic operations, which is of general nature - and as such is applicable also to other interval-based application domains. Specifically, we observe that most GMQL operations are **regions preserving**; by this term, we denote the operations in which all the regions of the result are a subset of the regions of one of the operands. Of course, this property does not hold in general, as the number of result regions can grow quadratically with the number of regions of the operands.

For executing region-preserving operations, we adopt an array-based structure which is quite popular in Spark libraries managing big datasets in preparation for machine learning - where a specific dimension is used for partitioning big data into subsets, organized in the format of large matrices, so that the dimension entries can be efficiently selected/compared/clustered.

While the adoption of an array engine for GDMS was discarded [8] (due to lack of performance in many operations over classic row-based engines) in this paper we demonstrate that an array-based approach implemented in Spark is commended for executing **chains of region-preserving operations**. In such condition, the benefits of the optimization pays off the cost of transforming some specific datasets from a row-based model to an array-based model and back. Therefore, our approach consists of detecting suitable parts of a query where the optimization is applicable, and then apply an improved Spark implementation to them.

Overall, we make the following contributions:

- We introduce a new multi-dimensional data model for GMQL, which uses regions, samples and attributes as dimensions and considers in the cells the values of each attribute.
- We introduce the property of region-preservation and then classify the GMQL operations according to how they manage the dimensions. In this way, we analyze orthogonal algebraic transformations from the point of view of the multi-dimensional model.
- We identify chains of region-preserving operations and we describe the invariant properties of queries over such chains.
- We define a method for computing chains of structure-preserving operations which consists of an operation-independent workflow pattern which embeds operation-specific tasks, with suitable abstractions for taking advantage of region-preserving operations in terms of large-scale map-reduce steps.
- We provide an Apache Spark implementation of the method that exploits classical Spark structures contiguous to machine learning.
- We present an experimental evaluation which shows the advantages of our solution.

The organization of this paper is as follows: Section 2 describes the two data models used in this paper, the background GMQL data model (with associated row-based physical model) and the multi-dimensional Genomic data model (with associated array-based physical model). Section 3 analyzes how the significant subset of region-preserving GMQL operations can be seen from a multi-dimensional perspective. Section 4 describes the algorithm for the efficient execution of chains of multi-dimensional operations; its main aspect is the ability of performing a *reduction by regions* that substitutes more complex tasks. Section 5 shows that the new approach outperforms our reference implementations by factors that increases significantly with the depth of iterations and are not too influenced by region sparsity; it also compares our SPARK implementation with a native array-based implementation using SciDB. Section 6 presents a complexity analysis, Section 6 describes related work and Section 7 concludes.

II. Model

A. Background: Genomic Data Model (GDM)

The genome can be considered as a long sequence of positions, divided into sub-sequences or chromosomes; thus, each region belongs to a **chr**omosome, **starts** at a specific position and **stops** at a specific position; **strand** denotes the reading direction of the chromosome and can be missing. GMQL has a domain-specific Genomic Data Model (GDM). Each dataset consists of several samples, which describe genomic regions extracted after a specific process (e.g., RNA expression of given genes or variants of a given donor). Regions of the same dataset have the same structure: all regions have coordinates, denoted by attributes `chr, start, stop` and `strand`. Each region is further characterized by a signal, consisting of an array of typed values. Regions of the same dataset have the same signature, also called the **dataset schema**.

Fig. 2 shows a small dataset with two samples, each region schema has the coordinates and then three attributes `pvalue, qvalue`, denoting the accuracy of the process used for producing the regions, and `score`, denoting the read accuracy from the sequencing machine. Note that two regions can have identical coordinates, e.g. in Fig. 2 the second sample has 2 regions with the same coordinates. Each sample is separately stored in hadoop distributed file system (HDFS) as a separate file; its format is compatible with classical tab-delimited or BED formats used by bio-informaticians.

SAMPLE1.BED

chr,	start,	stop,	strand,	pvalue,	qvalue,	score
chr1,	100,	200,	+,	0.0002,	0.01,	500
chr1,	200,	300,	-,	0.0003,	0.02,	300
chr2,	300,	400,	+,	0.002,	0.03,	400
chr2,	400,	500,	-,	0.0001,	0.01,	500

SAMPLE2.BED

chr,	start,	stop,	strand,	pvalue,	qvalue,	score
chr1,	100,	200,	+,	0.002,	0.02,	300
chr2,	300,	400,	+,	0.0002,	0.01,	100
chr2,	400,	500,	-,	0.0002,	0.015,	600
chr2,	400,	500,	-,	0.0002,	0.015,	300
chr3,	600,	700,	+,	0.0002,	0.005,	300

Fig. 2. Dataset with 2 samples in GMQL format.

In addition to regions, each GMQL sample has also metadata, in the form of semi-structured attribute-value pairs. They are smaller in size and describe the experimental conditions; in this paper, which is focused on region management, we omit to deal with metadata and their processing (which is unchanged as effect of the optimization).

When files are loaded by GDMS to the Spark engine, typically after a selection used to filter only few samples out of a larger dataset, they are stored with a row-based schema, shown in Fig. 3.

B. Multi-dimensional Genomic Data Model (MGDM)

Conceptually, we organize the multi-dimensional genomic data model (MGDM) along three dimensions.

sid,	chr,	start,	stop,	strand,	pvalue,	qvalue,	score
1,	chr1,	100,	200,	+,	0.0002,	0.01,	500
1,	chr1,	200,	300,	-,	0.0003,	0.02,	300
1,	chr2,	300,	400,	+,	0.002,	0.03,	400
1,	chr2,	400,	500,	-,	0.0001,	0.01,	500
2,	chr1,	100,	200,	+,	0.002,	0.02,	300
2,	chr2,	300,	400,	+,	0.0002,	0.01,	100
2,	chr2,	400,	500,	-,	0.0002,	0.015,	600
2,	chr2,	400,	500,	-,	0.0002,	0.015,	300
2,	chr3,	600,	700,	+,	0.0002,	0.005,	300

Fig. 3. Samples in row-based format supported by GDMS.

- The *first dimension* is associated to genomic **coordinates** C; it is obtained by considering all the regions of a dataset.
- The *second dimension* is associated to **samples** S_x.
- The *third dimension* is associated to **attributes** V.
- *Cells* include **attribute values**.

Regions may be duplicates inside a sample; therefore, each cell is an array having as many values as the replicas of each region. The conceptual multi-dimensional model is described in Fig. 4.

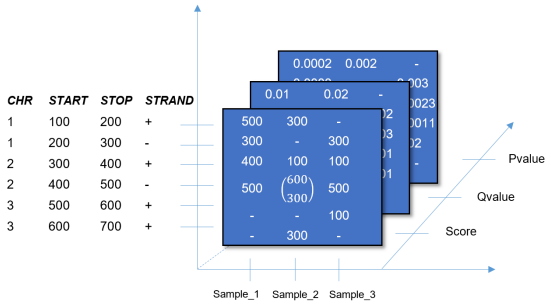


Fig. 4. Multi-dimensional genomic data model

In Spark, we represent the multi-dimensional model as an array structure <K, S>, where:

- The key *K* represents a quadruple *<chr, start, stop, strand>* of region coordinates; each key is unique.
- *S* is a table collecting all the features related to the same region coordinates. Rows of *S* correspond to attributes, columns of *S* correspond to samples. Cells in this matrix can be missing, when a given region is not present in a sample, or can be represented by arrays, when the region is replicated within the sample.

Fig. 5 shows the array structure for the dataset illustrated in Fig. 4. Note that the tables can be *sparse*, as many cells may be empty.

Two simple transformations τ_1 and τ_2 produce the conversion from row-based format (GDM in Fig. 3) to array-based format (MGDM in Fig. 5) and back. Informally, the former requires first to extract all unique regions K and then build the table S associated to K by suitable nesting. The latter flattens all internal tables and adds a sample identifier to each row.

The advantage of array-based storage is that algebraic operations can be separated into a first part which applies to keys and a second part which applies to tables; this separation is particularly useful with region-preserving operations, as no new key can be generated during the operation execution.

CHR	START	STOP	STRAND	ATTRIBUTES		
				S1	S2	S3
1	100	200	+	<500; <0.01; <0.0002;	300; 0.02; 0.002;	Ø>, Ø>, Ø>
1	200	300	-	<300; <0.02; <0.0003;	Ø; Ø; Ø;	300>, 0.02>, 0.003>
2	300	400	+	<400; <0.03; <0.002;	100; 0.01; 0.0002;	100>, 0.03>, 0.0023>
2	400	500	-	<500; <0.01; <0.0001;	{600; 300}; {0.015; 0.015}; {0.0002; 0.0002};	500>, 0.01>, 0.0011>
3	500	600	+	<Ø ; <Ø; <Ø;	Ø; Ø; Ø;	600>, 0.01>, 0.02>
3	600	700	+	<Ø; <Ø; <Ø;	300; 0.025; 0.0004;	Ø>, Ø>, Ø>

Fig. 5. Array-based representation of 3 samples in MGDM model

III. Language

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. Most GMQL operations are extension of classic relational algebra operations, twisted to the needs of genomics: SELECT, PROJECT, UNION, DIFFERENCE, GROUP, ORDER, MERGE and EXTRACT. Three domain-specific operations, called Genometric-JOIN, MAP and COVER, significantly extend the expressive power of classic relational algebra, as they perform region manipulations that are typically used in biology[2]. All GMQL operations are region-preserving except UNION and COVER; operations PROJECT and JOIN are region-preserving in most cases, but are not region-preserving when they use certain options, that are next discussed. These operations have a limited use, therefore **most of GMQL queries are region-preserving**[3]. We next characterize region-preserving operations by showing their effect upon the operands directly with the multidimensional model; new attributes are represented using the green color. The main features of region-preserving operations from the point of view of the multi-dimensional model are summarized in Table 6. The table describes the effects of each operation on the regions, samples, and attributes; column SAME indicates that the output is identical to the input, REDUCE indicates that the output may include less samples and attributes than the input, ADD indicates that the output may include new attributes.

A. Unary operations

1) PROJECT: The operator creates, from the input dataset, a new dataset with all the samples (with their regions and region values) in the input one, but keeping for each sample in the input dataset only those region attributes expressed in the operator parameter list. Therefore it filters attributes, as illustrated in Fig. 7. PROJECT can be used also to create new regions (`update option`), in such case it is not region-preserving.

[2]For the syntax and semantics of GMQL, see the *Introduction to the Language* at http://www.bioinformatics.deib.polimi.it/geco/?try.

[3]In the *Biological Examples* of GMQL, available at http://www.bioinformatics.deib.polimi.it/geco/?try and collected in the last two years, queries Q1, Q2, Q3, Q7, Q8 are region-preserving; query Q4 is partially region-preserving; queries Q5, Q6 and Q9 are not region-preserving, as they use the COVER operation.

OPERATIONS	REGIONS		SAMPLES		ATTRIBUTES		
	SAME	REDUCE	SAME	REDUCE	SAME	ADD	REDUCE
PROJECT	✗		✗				✗
SELECT		✗	✗		✗		
ORDER	✗		✗			✗	
GROUP	✗		✗			✗	✗
MERGE	✗			✗	✗		
EXTEND	✗		✗		✗		
DIFFERENCE		✗	✗		✗		
MAP	✗		✗			✗	
JOIN	✗		✗			✗	

Fig. 6. Effect of region-preserving operations upon regions, samples and attributes.

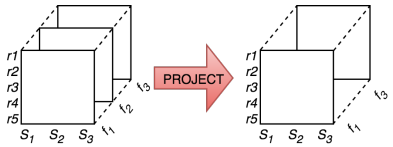


Fig. 7. PROJECT

2) SELECT: The operation extracts a subset of regions and values from the input dataset. A selection may create empty samples, i.e. samples without regions; during the evaluation of an operation chain, such empty samples are recognized just at the time of the row structure reconstruction.

3) ORDER: The operator is used to order regions of a sample according to some of their attributes; these may include the whole list of region coordinates, when the query asks for producing regions in the natural genome ordering. Ordering may be followed by a `TOP` clause restricting regions to be in a limited number. Ordering is rendered by adding a specific value that indicates, for each region, its position in the ordering, as illustrated in Fig. 8; cell positions of regions which do not satisfy the `TOP` clause are left empty. During the evaluation of a region-preserving chain, region ordering is left as final step when reconstructing the row-based model.

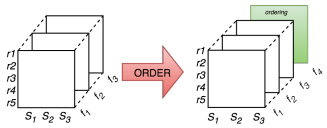


Fig. 8. ORDER

4) MERGE: The operator merges all samples into one. It builds a new dataset consisting of a single sample having as regions, all the regions of all the input samples, with the same attributes and values. It reduces the samples dimension to one, as shown in Fig. 9.

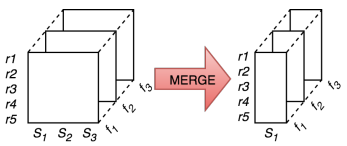


Fig. 9. MERGE

5) GROUP: When the group operator applies to region coordinates, it is implicitly managed by the multi-dimensional model, that groups together regions with the same coordinates; possible aggregate functions evaluated for each group are added within a new attribute, in a way which is similar to `ORDER`. When it applies to regions and to other additional attributes A, it changes the attribute structure of samples, by dropping all existing attributes, creating new attributes A, and creating one cell for each distinct combination of values of attributes A; this is illustrated in Fig. 10.

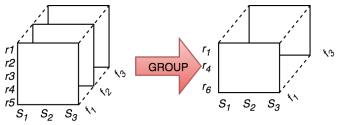


Fig. 10. GROUP

6) EXTEND: deals only with metadata - can therefore be considered as region-preserving.

B. BINARY operations

1) DIFFERENCE: It is a non-symmetric operator that produces one sample in the result for each sample of the first (left) operand, by keeping only those regions (with their attributes and values) of the first operand which do not intersect with any region in the second operand (also known as negative regions) (fig. 11).

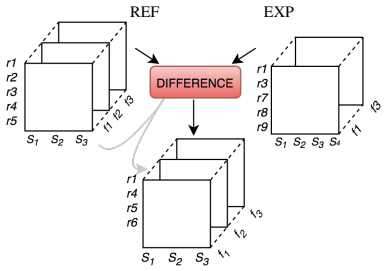


Fig. 11. DIFFERENCE

2) MAP: It is a non-symmetric operator over two datasets, respectively called REFERENCE and EXPERIMENT. It computes, for each region R of each sample in the reference dataset and for each sample in the experiment dataset, aggregates over the values of the experiment regions that intersect with R; we say that experiment regions are mapped to the reference regions R. The number of generated output samples is the Cartesian product of the samples in the two input datasets, but the output has the **same regions as the input reference dataset**, with their original attributes and values, plus the attributes computed as aggregates over experiment region values (Fig. 12).

3) JOIN: The operation applies to two datasets, respectively called REFERENCE and EXPERIMENT; the operation produces a result sample for every pair of samples of the operand datasets. The regions within each result sample are built from either REFERENCE, or the EXPERIMENT, or from both of them:

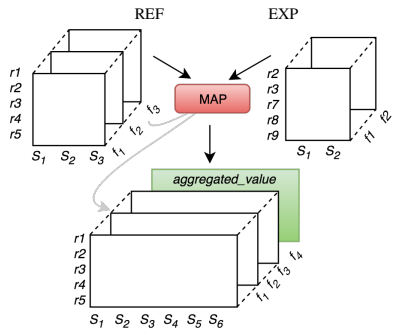
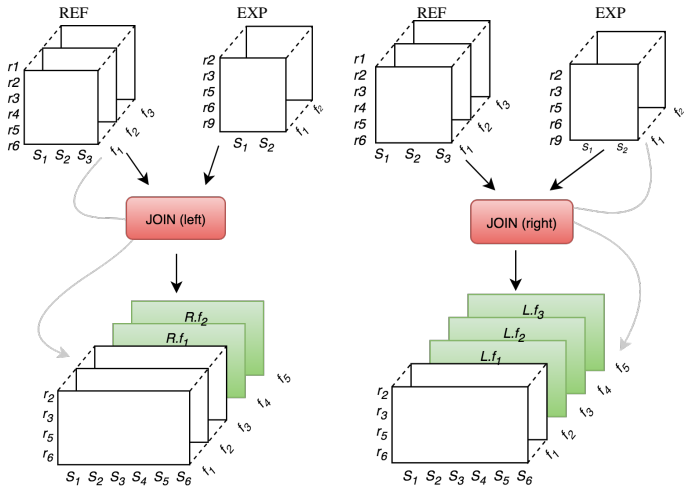


Fig. 12. MAP

- With the *Left* option, the JOIN result keeps only those regions from the left operand that intersect with regions of the right operand (Fig. 13(a)) [4].
- With the *Right* option, the JOIN result keeps only those regions from the right operand that intersect with regions of the left operand (Fig. 13(b)).

With the `intersect` and `contig` options, the JOIN result creates new regions composed respectively by the concatenation or the intersection of regions from the operands; with such options, JOIN is not region-preserving.



(a) JOIN left (b) JOIN right

Fig. 13. JOIN

C. Other Operations

The other GMQL operations include the COVER, which creates a single sample having new regions built by contiguous intersections of at least `min` and at most `max` operand regions, where `min` and `max` are suitable parameters; and the UNION, which by unifying all the samples includes regions from both left and right datasets. These operations, which generate new regions, are not further considered.

[4]Note that the MAP and JOIN LEFT operations are similar as they both use region intersection with the REFERENCE, however the former operation computes aggregate functions over the intersecting EXPERIMENT regions, while the latter generates one result region for each intersecting EXPERIMENT region. These operations are widely used in GMQL queries.

D. Chains of region-preserving operations

Region-preserving operations are widely used within queries; for maximizing efficiency, chains of region-preserving operations should be considered together, as they guarantee the advantage of factoring regions from the first to the last operation in the chain. Therefore, the method that will be discussed in the next section is invoked when a chain of two or more region-preserving operations is detected during the execution of arbitrary queries. The region-preserving chain is illustrated in Fig. 14; operations start with a transformation from the row-based to the array-based structure, continues with the execution of operations and ends with a transformation back to the row-based structure. Binary operations have one operand which adopts the array-based model and the other operand which adopts the row-based model.

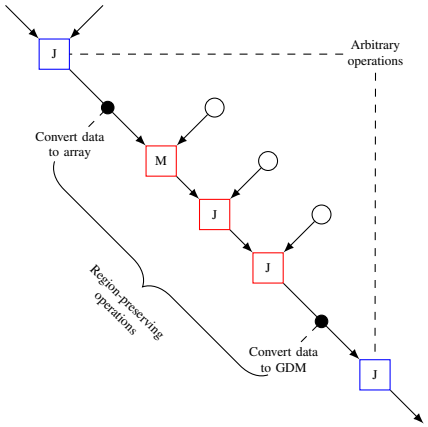


Fig. 14. Chain of region-preserving operations

IV. Method

We illustrate the region-preserving implementation of a chain of operations. Unary operations are applied to region-preserving chains one after the other, in essence by performing a reduce operation over regions so as to assemble at one processing node all the data items that allow reconstructing the matrix corresponding to a given region. The central body of the method is concerned with the Spark implementation of three binary GMQL operations: `MAP`, `GenoMetric-JOIN` and `DIFFERENCE`. The proposed method embodies a generic algorithm for interval-based processing, which is applicable whenever new intervals cannot be created by the computation.

A. Running Example

We consider two samples S1 and S2 for the Reference dataset (Left operand) and two samples S3 and S4 for the Experiment dataset (Right operand). The schema of samples S1 and S2 includes, in addition to coordinates, a `pValue`, denoting the quality of region processing; samples S3 and S4 include a `score`, denoting the accuracy of the sequencing process. Note that samples S1 and S2 have two regions with the same coordinates.

- The `MAP` result has four samples (one for each combination of its input samples); each result region includes the regions of its reference sample which overlap with some regions

of its experiment input sample; result regions include the counts of overlapping experiment regions.
- The `JOIN` result has also 4 samples and each result region includes the regions of its reference samples that overlap with some region of its experiment sample.
- The `DIFFERENCE` result has two samples that include those regions of the reference sample with no overlap with regions of the experiment sample; note that the second sample has no regions.

Resulting regions after the MAP, JOIN, and DIFFERENCE operations are shown in Fig. 15.

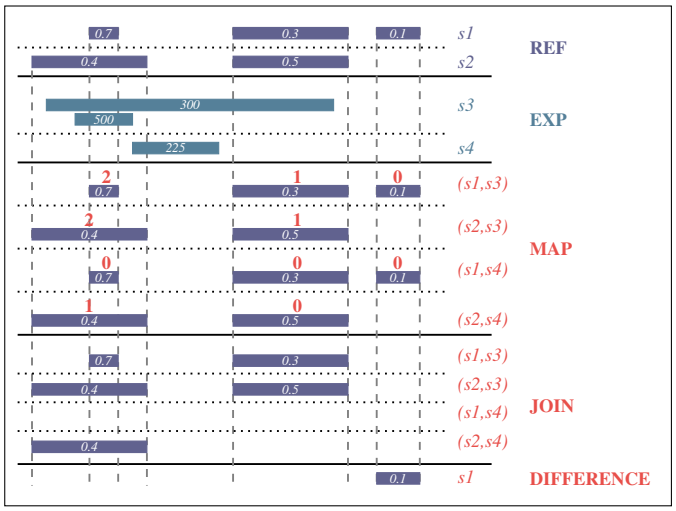


Fig. 15. Region manipulations produced by the three operations MAP, JOIN and DIFFERENCE.

B. Binning

Genome binning refers to the subdivision of the genome into small, identical partitions or **bins**; binning algorithms partition operands in order to speed up binary operations whose elementary operation is region intersection. The process of binning splits every chromosome of the genome into several bins of equal size b; for each chromosome, bins are progressively numbered starting from 0; the *i-th* bin spans from $b \times i$ to $b \times (i+1) - 1$; a genome position placed at i bases from the chromosome start is assigned to the bin $\beta(i) = \lfloor i/b \rfloor$.

Binning was introduced in [18] in order to speed up spatial joins, and is used in the UCSD genome browser to speed up the loading of genomic regions to the browser [12]. Binning is also used by the GenoMetric Query System in the row-based implementation [11] and is also used by the algorithm which is discussed next.

C. Algorithm

Fig. 16 illustrates the algorithm for implementing a chain of binary operations, consisting of a workflow of steps of low-level Spark operations. The algorithm preserves, after an arbitrary number of iterations, the regions of its first `Reference` dataset; white steps are common to all operations, while gray steps are performed in a different way for each GMQL operation. The reference dataset is initially translated to the array-based representation; the result of the last operation in the chain is translated back to a row-based representation.

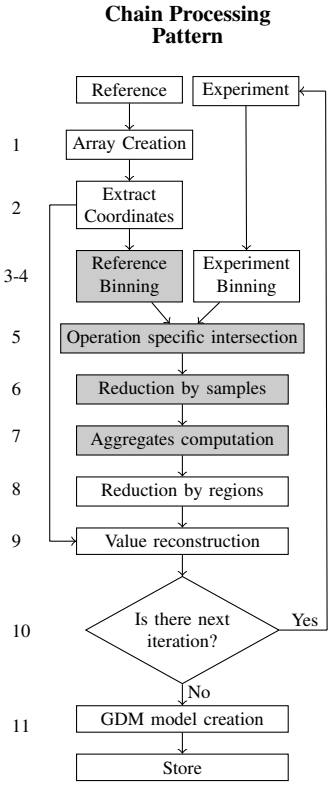


Fig. 16. Workflow of low-level Spark operations

In the step-by-step workflow description we focus on the `MAP` operation which computes both a `count` and the minimum `Score`. The **Reference** dataset in a row-based format is:

sid	chr	start	stop	strand	pValue
1	c1	50	70	+	0.7
1	c1	150	230	+	0.3
1	c1	250	280	+	0.1
2	c1	10	90	+	0.4
2	c1	150	230	+	0.5

The **Experiment** dataset in a row-based format is:

sid	chr	start	stop	strand	score
3	c1	20	220	+	300
3	c1	40	80	+	500
4	c1	80	140	+	225

D. Steps

1) Transformation to array-based model: Performs a transformation of reference dataset from the row-based to the array-based model.

Example. The following array is generated from the Reference dataset:

Chr	Start	Stop	Strand	V
c1	10	90	+	⟨ Ø; 0.4 ⟩
c1	50	70	+	⟨ 0.7; Ø ⟩
c1	150	230	+	⟨ 0.3; 0.5 ⟩
c1	250	280	+	⟨ 0.1; Ø ⟩

2) Extract Coordinates: Extracts the coordinates dimension from the input array.

Example. The following coordinates are retrieved:

Chr	Start	Stop	Strand
c1	10	90	+
c1	50	70	+
c1	150	230	+
c1	250	280	+

3) Reference Binning: It is responsible for copying reference regions to the bins. For every bin *b* intersecting with some reference region and for every experiment samples, a copy of the reference region is built, having as attributes the concatenation of the S_{id} of the experiment with `Chr, Bin, Start, Stop, Strand` of the reference.

Example. With bins of size 100, the following tuples are generated:

Sid	Chr	Bin	Start	Stop	Strand
3	c1	0	10	90	+
4	c1	0	10	90	+
3	c1	0	50	70	+
4	c1	0	50	70	+
3	c1	1	150	230	+
3	c1	2	150	230	+
4	c1	1	150	230	+
4	c1	2	150	230	+
3	c1	2	250	280	+
4	c1	2	250	280	+

This step is operation-specific because, in case of Genometric Join operation, the binning is performed by translating the reference regions according to the specific genometric predicate used in the operation; the complete binning algorithm is described in [11].

4) Experiment Binning: is responsible for copying experiment regions to the bins. For every bin *b* intersecting with experiment region, 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 tuples are generated:

Sid	Chr	Bin	Start	Stop	Strand	V
3	c1	0	20	220	+	300
3	c1	1	20	220	+	300
3	c1	2	20	220	+	300
3	c1	0	40	80	+	500
4	c1	0	80	140	+	225
4	c1	1	80	140	+	225

5) Operation specific intersection: It is responsible for computing a partial map within each bin. It joins references and experiments by `Sid, 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 starts (note that this bin exists and is unique by construction, as discussed in [11]).

At each selected pair of regions, a portion of the aggregate function is computed. A new region is built, having as attributes the concatenation of a new attribute `K` and `Sid` of the experiment with `Chr, Start, Stop, Strand` of the reference; `V` stores the experiment values to be used by the aggregate functions; `Count` stores 1 if regions intersect, otherwise it stores 0. The attribute `K` is obtained by hashing `Chr, Start, Stop, Strand`; this attribute is later used for assembling all copies relative to the same reference.

If the join fails, as we use the left join constructor, the reference information is stored into the result, with null values stored for the experiment; in this way, all reference regions with zero intersections are correctly accounted.

Example. The following regions are generated:

K	Sid	Chr	Start	Stop	Strand	V	Count
362	3	c1	10	90	+	[300]	[1]
362	3	c1	10	90	+	[500]	[1]
362	4	c1	10	90	+	[225]	[1]
613	3	c1	50	70	+	[300]	[1]
613	3	c1	50	70	+	[500]	[1]
613	4	c1	50	70	+	[null]	[0]
425	3	c1	150	230	+	[300]	[1]
425	4	c1	150	230	+	[null]	[0]
425	3	c1	150	230	+	[null]	[0]
425	4	c1	150	230	+	[null]	[0]
712	3	c1	250	280	+	[null]	[0]
712	4	c1	250	280	+	[null]	[0]

This step is operation-specific, therefore:

- In case of JOIN, the result includes the region values of the experiment; for what concerns matching regions of the reference, a copy of each matching region is produced for each intersection with an experiment region; if a region has no matching, then the region is dropped from the result.
- In case of DIFFERENCE, the step returns only those regions that have no intersection with the experiment dataset. It does not create any value.

6) Reduce by samples: It is responsible for assembling all copies corresponding to the same reference region and experiment sample at one node; the operation is performed thanks to a reduce phase which uses the `K` and `Sid` attributes. Partial sums are performed for computing `COUNT`, and lists of attribute values are added to a `Bag`.

Example. In the example, the twelve regions are reduced to eight, as some regions belong to the same sample (they have the same hash and Sid attributes). The following regions are generated:

K	Sid	Chr	Start	Stop	Strand	Bag	COUNT
362	3	c1	10	90	+	[300; 500]	[2]
362	4	c1	10	90	+	[225]	[1]
613	3	c1	50	70	+	[300; 500]	[2]
613	4	c1	50	70	+	[null]	[0]
425	3	c1	150	230	+	[300]	[1]
425	4	c1	150	230	+	[null]	[0]
712	3	c1	250	280	+	[null]	[0]
712	4	c1	250	280	+	[null]	[0]

This step is operation-specific and it takes place only when the cross-product should be performed between samples, therefore:

- in case of MAP and JOIN, the block concatenates all the values related to the same experiment sample (see below).
- in case of DIFFERENCE, the block does not take place.

7) Aggregates Computation: is responsible for computing aggregate functions, by applying them to the values built at block 6.

Example. The following regions are generated after applying the aggregate function `MIN(score)`:

K	Sid	Chr	Start	Stop	Strand	MIN(score)	COUNT
362	3	c1	10	90	+	[300]	[2]
362	4	c1	10	90	+	[225]	[1]
613	3	c1	50	70	+	[300]	[2]
613	4	c1	50	70	+	[null]	[0]
425	3	c1	150	230	+	[300]	[1]
425	4	c1	150	230	+	[null]	[0]
712	3	c1	250	280	+	[null]	[0]
712	4	c1	250	280	+	[null]	[0]

This block is operation-specific, and only takes place with a MAP operation.

8) Reduce by regions: It is responsible for assembling all copies corresponding to the same reference coordinates; the operation is performed thanks to a reduce phase which uses the `Chr, Start, Stop, Strand`.

Example. In the example, the eight regions are reduced to four by their `Chr, Start, Stop, Strand` attributes. The following regions are generated:

Chr	Start	Stop	Strand	MIN(score)	COUNT
c1	10	90	+	[300; 225]	[2; 1]
c1	50	70	+	[300; null]	[2; 0]
c1	150	230	+	[300; null]	[1; 0]
c1	250	280	+	[null, null]	[0; 0]

9) Value reconstruction: It joins by coordinates those coordinates that are produced at this stage with the progressively produced dimensions of the reference array.

Example. The obtained `MIN(score)` and `COUNT` values are added to the initial value dimensions (compare with Step 1). The following regions are generated:

Chr	Start	Stop	Strand	V: pValue	V: MIN(score)	V: COUNT
c1	10	90	+	⟨[Ø; 0.4],	[300; 225],	[2; 1]⟩
c1	50	70	+	⟨[0.7; Ø],	[300; null],	[2; 0]⟩
c1	150	230	+	⟨[0.3; 0.5],	[300; null],	[1; 0]⟩
c1	250	280	+	⟨[0.1; Ø],	[null, null]	[0; 0]⟩

Note that columns of `MIN(score)` and `COUNT` were evaluated by using the experiment samples, while columns of `pValue` correspond to the reference samples.

10) Next iteration test: If the current operation is the last operation in a chain step 11 is executed, else step 2 is executed for the next operation in the chain.

11) Row-based data transformation: It is responsible for the transformation from the array-based model to the row-based model.

Example. The following regions are generated after transforming the output to GDM model:

sid	Chr	Start	Stop	Strand	pValue	MIN	COUNT
13	c1	50	70	+	0.7	300	2
13	c1	150	230	+	0.3	300	1
13	c1	250	280	+	0.1	null	0
14	c1	50	70	+	0.7	null	0
14	c1	150	230	+	0.3	null	0
14	c1	250	280	+	0.1	null	0
23	c1	10	90	+	0.4	300	2
23	c1	150	230	+	0.5	300	1
24	c1	10	90	+	0.4	225	1
24	c1	150	230	+	0.5	null	0

Note that MAP operation produces the cross product of samples and thus after the transformation to the GDM model we have 4 samples. Their ids are obtained by hashing reference sample id with experiment sample id.

E. Structure of the result in the three examples

Fig. 17 illustrates how the result of the three operations is progressively assembled. One can note that the region-preserving operations indeed have the effect of adding dimensions (in the case of MAP and JOIN) and possibly dropping regions (in the case of DIFFERENCE), as discussed in Section III.

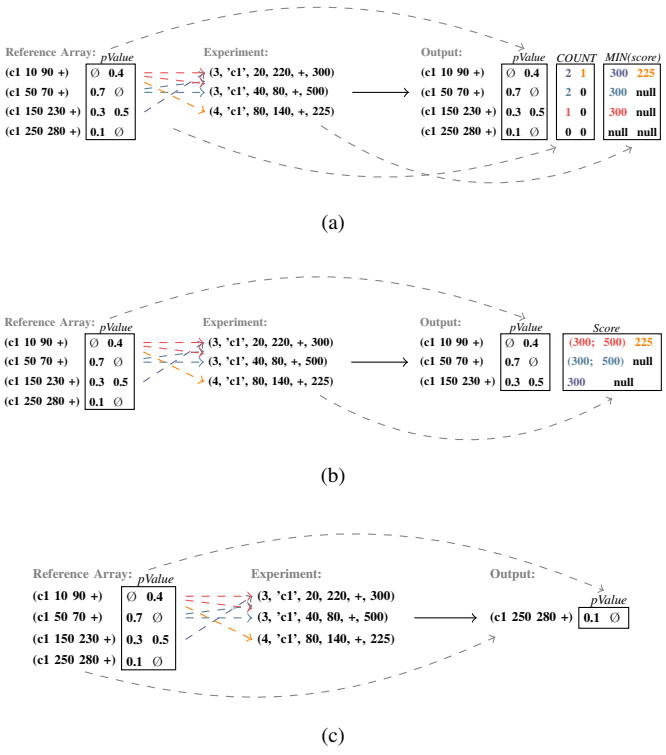


Fig. 17. Results of (a) MAP, (b) JOIN and (c) DIFFERENCE, showing that dimensions are added in JOIN and MAP and that regions are dropped in DIFFERENCE.

V. Performance and Scalability Evaluation

We performed our experiments on our server, an Intel® Xeon® Processor with CPU E5-2650 at 2.00 GHz, 32 hyper-threads, RAM of 192 GB, hard disk of 4x2 TB, and the engines Apache Hadoop 2.7.2 and Apache Spark 2.2.0. Our experimentation platform for the scalability test, discussed in Section V-C, is a cluster comprised of machines containing Intel E5620 processors with 8 hyper-threads and 47GB memory.

A. Performance Analysis

1) Chains of operations: We consider chains of MAP and JOIN operations. For the first iteration, we consider two input datasets as described in Table I; REFERENCE regions are RefSeq genes [5], EXPERIMENT regions are from Encode BrodaPeak Datasets [9]. For the following iterations, we used the result of the previous iteration as REFERENCE; a new EXPERIMENT is built by selecting 10 similar samples from ENCODE, with a similar number of regions.

TABLE I
Features of the real datasets used in the experiments

	Reference	Experiment
Size (MByte)	1.63	110.70
Regions	49052	901254
Samples	1	10

The size of generated output at each iteration is shown in Table II; sizes are relative to the row-based format obtained after materialization. As the number of samples grows of a factor ten

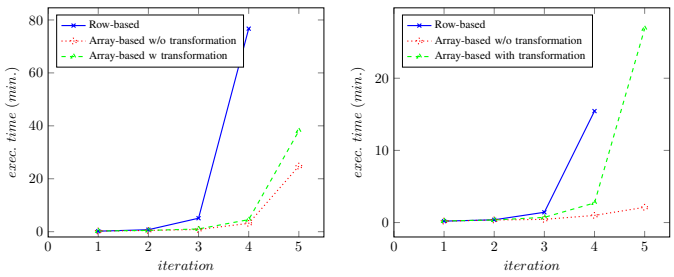
[5]From: https://www.ncbi.nlm.nih.gov/refseq/.

at each iteration, sizes become soon very significant. At the fifth iteration, we have 4 billion regions over 100 thousand samples and about 200 Gigabytes.

TABLE II
OUTPUT SIZE AFTER EACH ITERATION

Step	Size	Regions (Million)	Samples
1	17.3 MB	0.49	10
2	182.95 MB	4.9	100
3	1.92 GB	49.05	1000
4	20.17 GB	490.5	10000
5	211.78 GB	4905.2	100000

The fifth iteration can only be reached with the region-preserving algorithm, as shown in Fig. 18. Execution of the reference model becomes too expensive after three iterations with the MAP and four iterations with the JOIN operations. We also separately considered the cost of performing the final transformation to the row-based model; such cost is quite small in the MAP operation but becomes the most significant fraction of execution time in the last iteration of the JOIN; this is due to the number of copies of the LEFT operands that are generated (due to the semantics of the JOIN, that keeps one copy whenever the join predicate is true). If we omit the final model transformation, then the execution time of joins grows almost linearly with the number of iterations, which is expected thanks to the behavior of region preservation.



(a) (b)

Fig. 18. Performance of chains of operations: (a) MAP, (b) JOIN

Table III shows the execution times in seconds after each iteration for the MAP operation.

TABLE III
PERFORMANCE OF A CHAIN OF MAP OPERATIONS

Iteration	Array with transformation (sec.)	Array without transformation (sec.)	Row-based (sec.)
1	16.12	14.74	11.39
2	28.74	28.13	44.83
3	62.38	50.19	305.21
4	272.96	188.64	4602.88
5	2285.41	1486.44	–

B. Effect of sparsity

In order to assess how sparsity of regions influence performance, we built synthetic datasets with controlled region replication, ranging between no replication, full replication, and an intermediate value where each region is duplicated. Table IV describes the features of the synthetic datasets used in the test.

TABLE IV
FEATURES OF THE SYNTHETIC DATASETS USED IN THE EXPERIMENTS

	Reference	Experiment
Size (MByte)	30	57
Regions	1M	2M
Samples	5	10

We then considered a chain of MAP operations over synthetic datasets, and observed its execution time as a function of region sparsity; Table V summarizes the results. We note that the direct execution on the row-based model has better execution time with one iteration, but the array-based model has better execution times with two or three iterations. Further, we note that increased sparsity reduces the speed-up of the array-based solution, however such speed-up is between 6x and 10x when we perform three iterations (see Fig. 19). Thus, the performance of array-based approach is superior to the row-based approach also if the region matrix is sparse.

TABLE V
PERFORMANCE COMPARISON OF MAP OPERATION WITH DIFFERENT SPARSITY LEVEL

Sparsity	*Iteration*	Array with transformation (sec.)	Array without transformation (sec.)	Row-based (sec.)
0%	1	63.67	61.83	63.65
	2	171.99	126.73	280.52
	3	579.38	227.93	2344.23
50%	1	79.48	76.58	70.57
	2	211.04	183.03	262.03
	3	641.20	308.28	2620.31
100%	1	86.88	81.02	61.91
	2	222.69	184.49	251.28
	3	720.14	319.59	2140.46

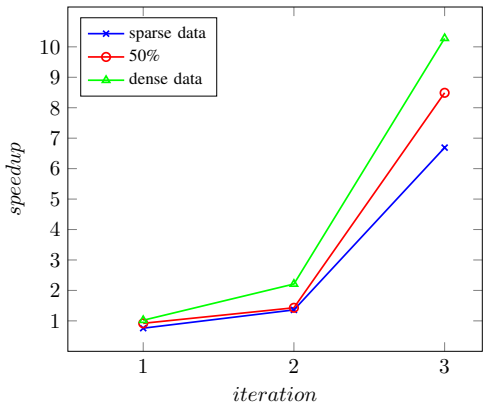


Fig. 19. Speedup

C. Scalability Analysis

To show the performance in respect to the number of machines in the cluster, we perform a test with one reference sample and fifteen experiment samples with three consecutive map operations (see Fig. 20). As expected, by increasing the number of machines up to 8 nodes the performance improves,

but a further increase of execution nodes produces a decrease of performance, as the communication overhead of the map operations becomes more dominant than the advantage caused by greater parallelism.

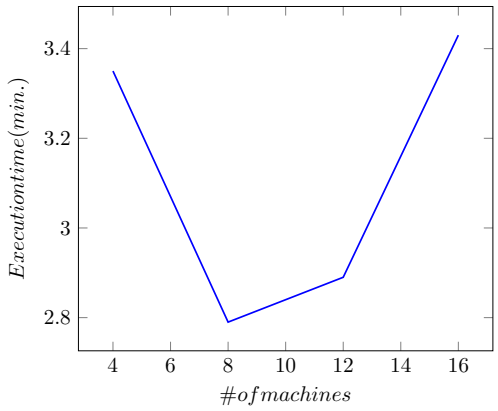


Fig. 20. Scalability of array-based model representation with a three consiqutive map operations and a data size of 1 reference sample and 15 experiment samples.

D. Comparison with SciDB

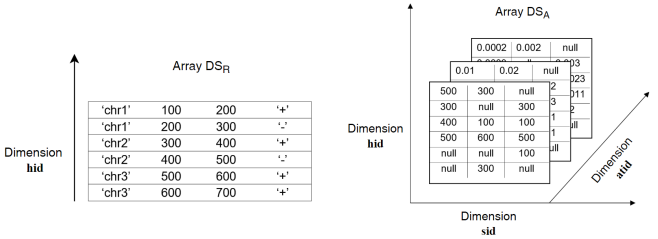
In this section we compare our Spark-based solution with an equivalent implementation using Sci-DB, a native array-based approach. With SciDB, we store a dataset DS using two arrays: $\texttt{DS}_\texttt{R}$ for region coordinates and $\texttt{DS}_\texttt{A}$ for attributes. These arrays have one common dimension - a hashed value of region coordinates $<$ `chr`, `start`, `stop`, `strand` $>$ as `hid`. Thus, $\texttt{DS}_\texttt{R}$ has one dimension `hid` and four attributes: `chr`, `start`, `stop` and `strand`.

$$\texttt{DS}_\texttt{R} =< \texttt{chr} : \texttt{string}, \texttt{start} : \texttt{int64}, \texttt{stop} : \texttt{int64}, \texttt{strand} : \texttt{string} > [\texttt{hid}] \quad (1)$$

$\texttt{DS}_\texttt{A}$ has one attribute `value` and three dimensions: hash id, sample id and attribute id.

$$\texttt{DS}_\texttt{A} =< \texttt{value} : \texttt{string} > [\texttt{hid}, \texttt{sid}, \texttt{atid}] \quad (2)$$

Fig. 21 shows the representation of the two arrays $\texttt{DS}_\texttt{R}$ and $\texttt{DS}_\texttt{A}$.



(a) DS_R (b) DS_A

Fig. 21. Reference dataset in SciDB

We implemented our optimized solution for chains of MAP operations (without model mapping) and repeated the experiment reported in Section V-A on the Amazon Web Services (AWS) cloud, using a conguration with i3.16xlarge machine, 64 cores, 488 GB of RAM. Fig. 22 shows that both Spark implementations outperform SciDB up to the third iteration, then SciDB's array-based implementation outperforms the row-based implementation; SciDB performance is almost linear because the array $\texttt{DS}_\texttt{R}$, that stores region coordinates, does not grow.

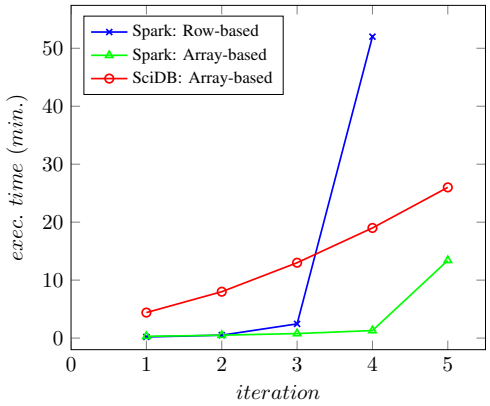


Fig. 22. Performance of chains of operations in SciDB and Spark

E. Complete Example

We next provide a complete example that shows all the three operations at work. The query is a classic computation in genomics: given a set of genes, (a) find those genes which overlap with given experimental regions (using a JOIN), (b) then subtract those which overlap with another experiment (using a DIFFERENCE), (c) then count over resulting genes the mutations of patients with no clinical indication (using a MAP). The query is:

```
GENES = SELECT(provider == "RefSeq") ANNOTATION;
EXP1 = SELECT(assay == "ChIP-seq" AND
    output_type == "peaks" AND
    experiment_target == "CTCF_human" AND
    biosample == "GM12878") ENCODE_NARROW;
EXP2 = SELECT(assay == "ChIP-seq" AND
    output_type == "peaks" AND
    experiment_target == "H3K4me1-human" AND
    biosample == "GM12878") ENCODE_NARROW;
MUT = SELECT(clinical == "NO") TCGA_SOMATIC_VAR;
JGE = JOIN(distance<0; output:left) GENES EXP1;
DIFF = DIFFERENCE() JGE EXP2;
MAPPED = MAP() DIFF MUT;
MATERIALIZE MAPPED INTO RESULT;
```

This query was executed on the input dataset featured in Table VI; execution times are reported in Table VII. The array-based method has clearly a shorter execution time. Three regions of the result, in row-based format, are shown in Fig. 23.

TABLE VI
INPUT DATA

Dataset Name	# of Regions	# of Samples	Size (MB)
RefSeq	50653	1	7.3
EXP1(TF)	2136849	10	126.47
EXP2(HM)	328949	3	22.34
Mutations	6121	14	1.37

VI. Complexity

We compare memory usage and execution time of row-based and array-based organization.

TABLE VII
EXECUTION TIME

Row-based	Array-based
379 sec.	49 sec.

chr,	start,	stop,	strand,	name,	COUNT
chr1	197170591	197447585	+	NM_001257965	4
chr11	22359666	22401046	+	NM_020346	4
chr2	215276460	215440653	+	NM_001080500	3

Fig. 23. Regions of the query result

A. Memory

We focus on the memory required for storing a dataset, consisting of multiple samples. Let I denote sample identifiers C denote coordinate attributes, V denote the signal attributes; let n denote the number of regions, n_D the number of distinct regions, with replication factor $\beta = n/n_D$. In the row-based organization, the memory usage is simply the storage used by each row:

$$Mem_{row} = n \times (size(I) + size(C) + size(V))$$

In the array model, every distinct region is used as a key; for each such key, we store two arrays. The former one contains the identifiers of the samples that include that region, the latter is the array of values that are associated to that region. Although keys depend on the number of distinct regions, samples and values reflect the real region replication, hence their cost includes the replication factor β. We use an indexing scheme from keys to arrays that can be regarded as a fixed overhead F, for a given array implementation. Hence the memory usage is given by:

$$Mem_{array} = n_D \times (size(C) + \beta \times size(I) + \beta \times size(V) + 2 \times F)$$

The difference between the two memory usages is simply:

$$\begin{aligned} Delta &= Mem_{row} - Mem_{array} \\ &= n \times size(I) + n \times size(C) + n \times size(V) \\ &\quad - n_D \times size(C) - n_D \times \beta \times size(I) \\ &\quad - n_D \times \beta \times size(V) - 2 \times n_D \times F \\ &= (n - n_D) \times size(C) - 2 \times n_D \times F \end{aligned}$$

In GDM, coordinates require 57 bytes, while the fixed overhead F in the Java implementation requires 12 bytes, hence $Delta = 57 \times n - 81 \times n_D$; when $Delta$ is positive, the array model requires less memory. In terms of the replication factor, $Delta$ is positive when $\beta > 1.4$; replication factor of datasets is typically higher, especially after the evaluation of operations.

B. Computational complexity

The dominant operation for complexity is step 5 of Figure 16, where the two datasets are intersected; this operation has a complexity which grows quadratically with the size of data, whereas the complexity of the other operations grows linearly. In the original row implementation, after binning both the reference and experiment datasets, the complexity of this step is $\mathcal{O}((n/Bin) \times (m/Bin)) = \mathcal{O}((n \times m)/Bin^2))$, where n, m respectively denote the number of regions of the reference and experiment dataset, and where Bin is the average number regions in each bin (assuming uniform distribution of regions to bins). In the array representation, which also uses binning, the regions in the reference do not have duplicates, hence the complexity of this step is $\mathcal{O}((n_D/Bin) \times (m/Bin)) = \mathcal{O}((n \times m)/(\beta \times Bin^2)))$. Thus, the array-based solution has lower complexity when β is greater than 1.

In summary, with high replication factors, the array-based solution is superior to the row-based solution both for the memory usage and computational complexity. Note that β grows very fast as effect of a chain of operations, e.g. in the example of section V-A β grows from 1.5 in the first iteration to 10 for the second iteration and 100 for the third iteration.

VII. RELATED WORK

A. Background

In the Spark implementation of GMQL, described in [11], we developed reference algorithms for join and map (with an optimized, three-step approach to join evaluation) and we developed a theory of binning strategies as a generic approach to their parallel execution (which allows a simplication of the parallel processing). In [8], we compared the evaluation if our reference algorithm with an alternative implementation using SciDB [2], array-based scientific database, as database engine; we focused on few but representative operations (filter, aggregate, map, join). The paper showed that SciDB performs better when it directly uses filtering and aggregation over an array-based physical data organization, but Spark performs better on massive region mapping operations (maps and joins). This work convinced us that the array-based approach has virtues but also limitations, and inspired us in our current work, which is using the array-based paradigm on top of a strong row-based data engine.

B. Array Databases

Our work is indebted to several contributions given to join algorithm optimization on the context of distributed database systems ([4], [20], [21]) and of parallel dataflow systems like Hadoop [1] and Spark [25] ([3], [15], [19], [24], [26]). We next review the work focused on array-based implementations.

1) BlockJoin [13]: It is an optimized join algorithm which fuses relational joins with blocked matrix partitioning, with the objective of building suitable data partitioning for later use in machine learning algorithms. It nicely support linear algebra programs in the context of relational algebra; although we focus on data extraction, we borrow from this approach the idea of using array-based data representation for machine learning.

2) SciSpark [16]: It extends Spark for scaling scientific computations. It introduces the Scientific Resilient Distributed Dataset (sRDD), a distributed-computing array structure which supports iterative scientific algorithms for multi-dimensional data. SciSpark converts structured files (e.g. NetCDF) into a collection of Spark-readable data frames named *sciTensors*, each including key/value pairs and array data. Therefore, Spark can repetitively manipulate multiple array datasets at runtime. SciSpark requires users to provide partitioning and file-loader functions.

3) SparkArray [23]: It extends Spark with a multi-dimensional array data model and a set of common used array operations (e.g., filter, subarray, smooth and join).

Comment: The focus of these papers is only on the array model, while we propose switching between relational and array models on-the-fly based on the optimal data representation for each operation. Both SciSpark and SparkArray do not provide an optimized solution for a chain of operations like the ones mentioned in this work.

4) RasDaMan [6]: It is one of the earliest next-generation array DBMS for multi-dimensional discrete data, supporting an extended SQL query language. It stores its data as tiles, i.e., possibly non-aligned sub arrays, and resorts to blobs in an external DBMS. While their optimizer provides a rich set of heuristic-based rewrites, to the best of our knowledge, RasDaMan does not perform joint optimization over relational and array data.

5) SciDB [2]: It is an array database that, in contrast to RasDaMan, provides its own shared-nothing storage layer. This allows SciDB to store and query tiles more efficiently. It provides a variety of optimizations, like dealing with overlapping chunks and data compression. However, as it was shown in [8], it does not efficiently perform massive operations like mapping and joins.

6) TileDB [17]: It is a system that stores multi-dimensional array data in fixed size data tiles, which is optimized for both dense and sparse multi-dimensional arrays. GenomicsDB [10] is built on TileDB and it is used by the Broad Institute to store genomic variant data in 2D arrays, where columns and rows correspond to genome positions and samples, respectively.

Comment: All the aforementioned technologies provide a general solution for array databases; they build indexes for the dimensions to facilitate fast access which in turn speed up range queries and equi-joins on the dimensions. However, domain-specific join operations in Genomics are similar to theta-join which are not optimized by a dimensional organization. Coding genomic operations using SQL or SQL-like languages that are provided in RasDaMan and SciDB is rather difficult.

VIII. Conclusions

In this paper, we introduce a scalable algorithm for structure-preserving genomic operations; we showed that our approach has a strong potential for performance improvement. Our approach is applicable to other interval-based domains; in our research work, we used the GDM model to represent twitter accounts, meteorological measures, and github commits. Structure-preservation occurs when no new coordinate regions are created by the query; such situation is common in these interval-based domains.

Our future work will focus on evaluating the extension of the array-based model to arbitrary operations, by looking for classes of array-based optimization beyond region-preservation; we will also consider a stronger integration of the array-based approach with machine learning, by investigating the direct application of machine learning methods to the array-based model.

Acknowledgment

This research is funded by the ERC Advanced Grant project 693174 "GeCo" (Data-Driven Genomic Computing), 2016-2021.

References

[1] P. 4. Apache hadoop, oct 2017.
[2] P. 4. Scidb array database, oct 2017.
[3] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.
[4] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.
[5] Apache. Flink, oct 2017.
[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *Acm Sigmod Record*, volume 27, pages 575–577. ACM, 1998.
[7] M. Bertoni, S. Ceri, A. Kaitoua, and P. Pinoli. Evaluating cloud frameworks on genomic applications. *IEEE Big Data Conference, Santa Clara*, Nov. 2015.
[8] S. Cattani, S. Ceri, A. Kaitoua, and P. Pinoli. Evaluating big data genomic applications on scidb and spark. *Proc. Web Engineering Conference*, June 2017; Rome.
[9] E. Consortium. An integrated encyclopedia of dna elements in the human genome. *Nature*, 2012.
[10] Intel. Intel. genomicsdb.
[11] A. Kaitoua, P. Pinoli, M. Bertoni, and S. Ceri. Framework for supporting genomic operations. *IEEE Transactions on Computers*, 66(3):443–457, 2017.
[12] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at UCSC. *Genome Res.*, 12(6):996–1006, Jun 2002.
[13] A. Kunft, A. Katsifodimos, S. Schelter, T. Rabl, and V. Markl. Blockjoin: efficient matrix partitioning through joins. *Proceedings of the VLDB Endowment*, 10(13):2061–2072, 2017.
[14] M. Masseroli, P. Pinoli, F. Venco, A. Kaitoua, V. Jalili, F. Palluzzi, H. Muller, and S. Ceri. Genometric query language: a novel approach to large-scale genomic data management. *Bioinformatics*, 31(12):1881–1888, 2015.
[15] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 949–960. ACM, 2011.
[16] R. Palamuttam, R. M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. McGibbney, and P. Ramirez. Scispark: Applying in-memory distributed computing to weather event detection and tracking. *IEEE International Conference on Big Data (Big Data)*, pages 2020–2026, 2015.
[17] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The tiledb array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360, 2016.
[18] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. *ACM SIGMOD Record*, 25(2):259–270, 1996.
[19] O. Polychroniou, R. Sen, and K. A. Ross. Track join: distributed joins with minimal network traffic. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1483–1494. ACM, 2014.
[20] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1194–1205. IEEE, 2016.
[21] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1345–1354, 1993.
[22] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J.Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.
[23] W. Wang, T. Liu, D. Tang, H. Liu, W. Li, and R. Lee. Sparkarray: An array-based scientic data management system built on apache spark. *IEEE International Conference on Networking, Architecture and Storage (NAS)*, August 2016.
[24] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.
[25] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.
[26] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 1060–1071. IEEE, 2010.