Array-based Data Management for Genomics

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

Abdulrahman Kaitoua
TU-Berlin & DFKI
abdulrahman.kaitoua@outlook.com

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

***Abstract*—With the huge growth of genomic data, exposing multiple heterogeneous features of genomic regions for millions of individuals, we increasingly need to support domain-specific query languages and knowledge extraction operations, capable of aggregating and comparing trillions of regions arbitrarily positioned on the human genome. While row-based models for regions can be effectively used as a basis for cloud-based implementations, in previous work we have shown that the array-based model is effective in supporting the class of region-preserving operations, i.e. operations which do not create any new region but rather compose existing ones.**

In this paper, we remove the above constraint, and describe an array-based implementation which applies to unrestricted region operations, as required by the Genometric Query Language. Specifically, we define a wide spectrum of operations over datasets which are represented using arrays, and we show that the array-based implementation scales well upon Spark, also thanks to a data representation which is effectively used for supporting machine learning. Our benchmark, which uses an independent, pre-existing collection of queries, shows that in many cases the novel array-based implementation significantly improves the performance of the row-based implementation.

I. Introduction

Among life science applications, genomics is the most relevant for the database community. The recent development of massive genome sequencing technologies is producing a huge amount of genomic datasets: by 2025, the global size of genomic data is expected to exceed the size of all YouTube videos by two orders of magnitude [1]. World-wide international data sequencing efforts are continuously producing important results in terms of curated repositories; classical examples include the Encyclopedia of DNA Elements (ENCODE, [2]) the Cancer Genome Atlas (TCGA, [3]), the 1000 Genomes Project [4], the 100000 Genomes Project [5], the International Cancer Genome Consortium [6], and others. Their common aspect is to present genomic data by means of efficient region-based formats, where region sizes may range from a single basis (in the case of single nucleotide variations - SNVs) to huge genome segments describing large genes or genome copies created by rearrangements. According to many biologists and clinicians, a wealth of information is already undisclosed within such repositories; their discovery requires a combination of data extraction, analysis, and exploration tools.

So far, very few systems are specifically dedicated to region-based genomic calculus. Among them, the GenoMetric Query Language (GMQL) [7] provides a high-level, declarative approach to data extraction and a cloud-based implementation for managing region-based genomic repositories, which have the typical big data dimensions. 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*. The GMQL system is best suited to support batch queries over large datasets, typically part of complex pipelines. After testing several cloud engines [8], [9], the current implementation uses Apache Spark [10]; depending on data sizes executions can require hours of computing time, hence performance optimization is important.

With a cloud-based relational engine as target, the most obvious data model maps each region to a table row, with genomic coordinates representing the region's position on the genome and a feature vector representing the region's properties. As in classic relational engines, this representation quickly hits performance bottlenecks when regions of different datasets are combined by using binary operations, and specifically join, which is the heaviest binary operation also in the genomic domain. For scalability, join algorithms use binning [11], a partitioning of the genome into segments of equal size, such that each bin is processed in parallel. Optimal binning strategies, discussed in [12], highly improve the join performance, but the scale up is limited due to intrinsic synchronization requirements of the method: contiguous bins may produce replicated regions in the results, their pruning induces a need for data shuffling, and at some point the data shuffling overhead becomes predominant.

In our previous work [13] we discovered the potential of using a multi-dimensional representation of genomic data on top of data flow engines. Our approach was concerned only with *region-preserving* operations, i.e. operations in which all the regions of the results are a subset of the regions of some of the operands. As region-preserving chains of operations frequently occur in GMQL programs, we used data model transformations from row to array prior to executing the chains, and from array back to rows at the end of the chains.

In this paper, we present a full implementation on the array-based model of all GMQL operations, by including also the operations which are not region-preserving. We store array-based genomic datasets by taking advantage of a novel organization of Spark RDDs inspired by a solution already in use in MLSpark, a library for managing big datasets in machine learning applications. In previous works, we experienced the use of a pure array-based engine such as SciDB (see [14] and more recently [13]); we found that the use of an array-based engine does not outperform the combination of an array-based model and Spark. Our choice of concentrating on

Spark has also some pragmatic motivations, as we reuse many architectural components. Overall, we make the following contributions:

- We describe region-based abstractions in terms of the array-based model.
- We present a novel structure for managing array-based genomic datasets using Spark RDDs.
- We present an implementation of the most critical region-based genomic abstractions on the Spark engine.
- We provide a thorough experimental evaluation which shows the advantages of our solution.

In particular, we use as benchmark a set of queries defined in the context of the STQL language [15], already encoded in GMQL (see [16], supplemental material) in order to compare languages both in terms of performance and of expressive power. For complex queries, we show that the array-based solution has significant speedup (up to 35) over the row-based solution. As discussed in the evaluation, the most significant factor which explains such huge gain of performance is a high region replication factor; note that many datasets have hundreds or thousand of samples, and each sample is typically associated to a different patient or tissue or biological condition. In these cases, samples may have many regions with identical coordinates, e.g. corresponding to genes, or to highly recurrent mutations, or to protein binding sites.

The paper is organized as follows. Section II presents background material, Section III presents how the array-based model supports all GMQL abstractions, Section IV describes the management of the array-based data model and operations in Spark, and Section V presents a comparative performance evaluation, including both operation-by-operation evaluation on synthetic data and a benchmark consisting of fourteen queries defined in [15]. We then present related work and our conclusions.

II. Background

A. GenoMetric Query Language (GMQL)

GMQL is a data management and query system for genomic data management. The main components of the system are: (a) a repository where we import large public datasets as well as private data; (b) a query language, which combines classical relational algebra operators with domain specific ones; (c) a Web accessible interface for browsing the repository, composing and launching queries and retrieving the results; and (d) an implementation based on cloud computing technologies to cope with the big data nature of genomic datasets.

In the repository, data are organized using a data model based on the concepts of dataset and sample; a dataset is a collection of homogeneous samples (e.g., samples produced by the same technology, hence exposing the same schema). Each sample is typically associated to a distinct biological tissue (that in turn may have a human donor) and consists of two components: region data (for representing features such as genomic annotations, DNA variants, genome rearrangements, and results of experiments measuring expression levels or peaks of expression) and the associated metadata (providing clinical and biological features of the sample as well as information on how the experiment has been performed). Region data can be very large, e.g. including trillions of regions. Metadata are typically much smaller. GMQL is an algebraic language whose operations apply to either one or two datasets and produce a result dataset; a GMQL query is invoked by requiring the materialization of its result, which in turn causes the recursive computation of all the datasets building intermediate results, up to the source datasets stored in the repository.

B. Row-Based Genomic Data Model

GMQL currently supports a row-based data model, which is next described. 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; a binary **strand** denotes the reading direction of the chromosome and can be missing. 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. Metadata are in the form of semi-structured data, with an `sid` to indicate the sample to which each metadata information refer to, and then `attribute` and `value` corresponding to specific factual information. Our work is focused on region management; we disregard metadata structure and their processing, whose organization in not affected by the choice of data model.

Fig. 1 shows an example of row-based data model for a small dataset with regions and metadata. The region schema has a `sid` to indicate the sample, then the coordinates, and then two attributes `signal` and `pvalue`, representing the signal. The example describes three samples `S1, S2` and `S3`, each containing 3 regions. Metadata describe the properties of the experiments producing the three samples.

Regions table

sid,	chr,	start,	stop,	strand,	pValue,	signal
s1,	chr1,	50,	70,	*,	0.1,	50
s1,	chr1,	50,	70,	*,	0.3,	30
s1,	chr7,	25,	100,	*,	0.1,	15
s2,	chr2,	30,	90,	*,	0.9,	30
s2,	chr7,	100,	150,	*,	0.9,	10
s2,	chr7,	100,	150,	*,	0.4,	25
s3,	chr1,	50,	70,	*,	0.5,	35
s3,	chr2,	30,	90,	*,	0.5,	95
s3,	chr7,	100,	150,	*,	0.5,	90

Metadata table

sid,	attribute,	value
s1,	cell,	blood
s1,	antibody,	CTCF
s2,	cell,	brain
s2,	antibody,	CTCF
s3,	cell,	blood

Fig. 1. Samples in row-based model supported by GMQL system.

C. Array-Based Genomic Data Model

In this paper, we investigate the use of the multi-dimensional data model, expanding on our previous work [13].The array-based representation of genomic datasets has three dimensions:

- The first dimension is associated to genomic **coordinates**; it includes a distinct entry for each distinct region of the dataset.

- The second dimension is associated to **samples**; it includes a distinct entry for each sample.
- The third dimension is associated to **signals**; it includes a distinct entry for each attribute of the dataset schema.
- Cells include **attribute values** corresponding to a specific triple of region/sample/attribute; regions within one sample may have several replicas, therefore cells may include arrays of values.

Fig. 2 shows an example of array organization; the figure highlights that cells are provided with a fast associative index access, which enables the extraction of specific cells by providing specific values of their dimensions. Arrays can also be effectively segmented by slice and dice operations, also effectively supported by means of the index accesses.

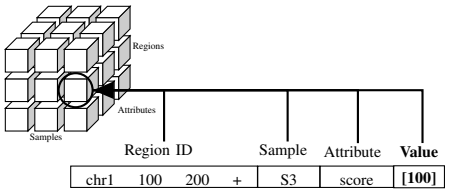


Fig. 2. Array-based genomic data model.

III. Genomic Operations upon Arrays

Genomic operations on arrays are classified in Table I; the main distinction is between unary operations (which apply to one dataset) and binary operations (which apply to two datasets); in addition, operations are **region preserving** when no new regions are created during processing, as discussed in [13]. Another important quality of operations is being **space-localized**; this occurs when the processing of each region occurs independently from other regions. In contrast, we call **space-rearranged** those operations which require merging the contribution of several input regions to create the result regions. It turns out that all unary operations except histogram and cover are both region-preserving and space-localized, and that some binary operations are region-preserving but none of them is space-localized, as shown in Table I.

TABLE I
Array Operations

	Region-preserving		**Arbitrary**
	Space-Loc	**Space-Rearr**	**Space-Rearr**
Unary	*Select, Project, Merge, Group*		*Cover, Histogram*
Binary		*Left/Right Join, Map Difference*	*Intersect/Contig Join, Union*

A. Simple Unary Operations

We first discuss four unary operations: *Select, Project, Merge* and *Group*. They are classified as simple because they are both region-preserving and space-localized.

1) Select: The selection predicate is a conjunction of two parts, which apply respectively to coordinates or to features. The region predicate selects the regions that belong to the result; for those regions, the feature predicate then selects the features that belong to the result. When a sample remains empty as result of the selection, it is not included in the result dataset. Fig. 3 shows the effect of applying the selection predicate `chr=2 and A>50` to a given array, where the condition on the coordinates eliminates some regions and the condition upon features eliminates some values from the array.

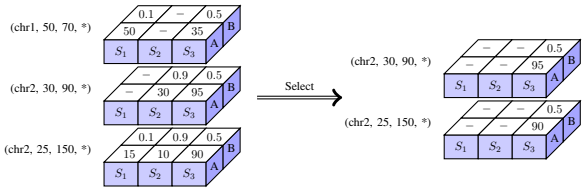


Fig. 3. Example of Select operation with the predicates "`chr=2`" and "`A>50`".

2) Project: The operator filters attributes away; from the input array it creates a new array with all the regions and samples as the input one, but keeping only those attributes expressed in the operator parameter list. The operator can also be used to create new attributes, resulting from applying arbitrary expressions to the input attributes. For instance, the projection shown in Fig. 4 includes in the result tha attributes `B` and `newA`, where `newA` is a new region attribute whose value is obtained by applying the expression $0.45 \times A$ to the `A` and attribute.

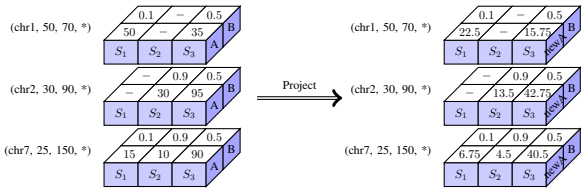


Fig. 4. Example of Project operation that creates a new attribute named `newA` and includes in the result `B` and `newA` attributes.

3) Merge: The operator merges all samples of the input array into one. It builds a new array consisting of a single sample, having as regions all the regions of all the input samples, with the same attributes and values. Fig. 5 illustrates the example of merging an input array into one sample; the resulting array has a single sample, with a new attribute name S_{new}, storing in its cells the vector of values from all samples.

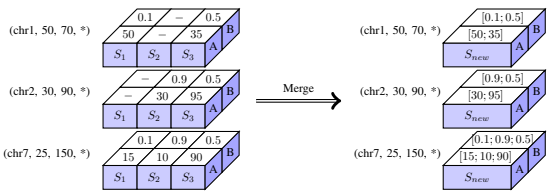


Fig. 5. Example of Merge operation.

4) Group: This operation groups the input array by regions and computes aggregate values on each group. An example is show in Fig. 6, which applies to the array resulting from the merge operation of Fig. 5 and computes the two functions `MIN(A)` and `MAX(A)`. The resulting array has two distinct attributes, one for each computed aggregate function.

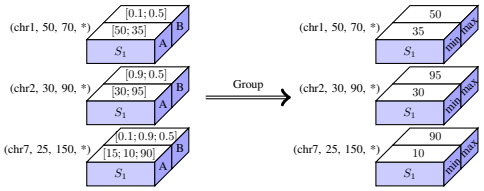


Fig. 6. Example of Group operation that computes the two functions *MIN(A)* and *MAX(A)*.

B. Histogram and Cover

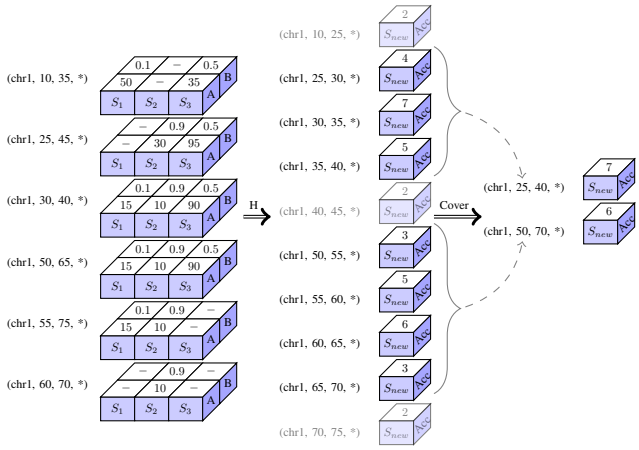


Fig. 7. Example of Historgam and Cover operations with `minAcc=3` and `maxAcc=ANY`.

Histogram is a domain-specific genomic operation that applies to a multi-sample dataset and produces a single sample having as regions the intersection of the input regions; the attributes are results of aggregate functions applied to contributing regions. In the example provided in Fig. 7 we show the `Count` (also called **accumulation index**). Computing the counts is facilitated in the array-based model, as the region coordinates can be used to access the input cells; however, the operation is space-rearranging and the output regions have new coordinates. Note that in Fig. 7 we start with six regions and we end up with ten regions. Histograms can be filtered based on the minimum and maximum accumulation index; if we filter the result with the condition `minAcc=3` and `maxAcc=ANY`, then the regions having counters less than 3 are removed from the result[1].

Cover connects all the regions in the histogram that satisfy the condition on the accumulation filter; technically, it is the contiguous intersection of at least `minAcc` and at most `maxAcc` contributing regions. A typical property of the region is the Jaccard Index, measuring the degree of overlap of contributing regions, which is computed as default. Other values may be given by computing aggregates over the connected regions. In the example of Fig. 7, we show the result of a cover operation also expressed with the condition `minAcc=3` and `maxAcc=ANY`, which produces two regions in the result, from 25 to 40 and from 50 to 70. In the specific example we compute the `Max` aggregate function, hence the region's values are respectively 7 and 6.

[1]The keyword `ANY` can be used as `maxAcc`, and in this case no maximum is set; the keyword `ALL` stands for the number of samples of the operand, and can be used for `minAcc` or for `maxAcc`.

C. Binary Operations

Binary array operations include: Join, Map, Difference and Union. Operations are space-rearranged (cells in the result generally depend on more than one cell); moreover, some join operations are not region-preserving.

1) Map: The *Map* operation applies to 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 regions which intersect with the Reference regions. Although the operation is defined for arbitrary arrays, its most typical application occurs when the reference consists of just one sample (e.g., a collection of genes) and then the aggregate function computes a property of its regions (e.g., the average gene expression over multiple experiments).

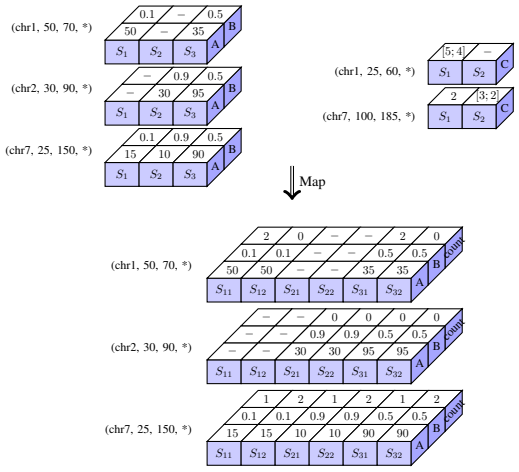


Fig. 8. Example of Map operation with `count` as aggregate function.

Fig. 8 illustrates an example of applying Map operation to 2 input arrays, with `count` as aggregate operation. Note that it builds the cross-product of samples, it replicates the attributes of the first operand, and adds a counter of the intersecting

regions. In the specific example, the only intersections occur between the first and last regions of the two operands; the counter is 2 when the second operand has 2 intra-sample replicates.

2) Join: The *Join* operation applies to two datasets, respectively called Reference and Experiment. It 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:

- With the *Left* option, the Join result keeps only those regions from the left operand that intersect with regions of the right operand.
- With the *Right* option, the Join result keeps only those regions from the right operand that intersect with regions of the left operand.
- With the *Contig* option, the Join result creates new regions composed by the concatenation of regions from the operands (on the same chromosome).
- With the *Intersect* option, the Join result creates new regions as the intersection of regions from the operands.

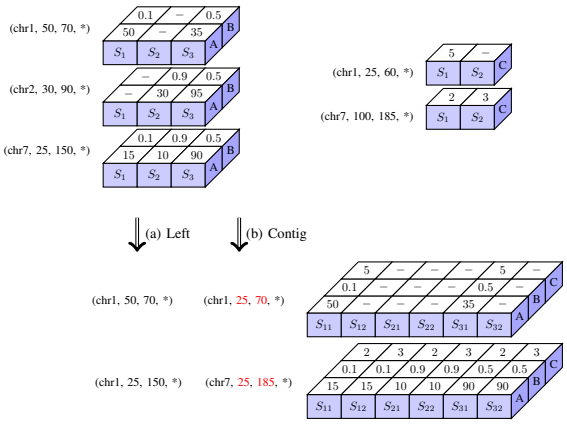


Fig. 9. Example of Join operation with (a) Left option and (b) Contig option.

The first two options above are region-preserving, while the last two options are not. Fig. 9(a) illustrates an example of Left Join operation; the only non-empty intersections occur between the first and last regions of the two operands. Fig. 9(b) illustrates the result of the same operation with the Contig option; note that the result is identical except for the regions, which are the intersection of the first and last regions of the operands.

3) Difference: This operation applies to two arrays and produces the result 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. 10 illustrates an example of applying Difference operation on two input arrays; the resulting array has only one region.

4) Union: This operation is used to integrate possibly heterogeneous samples of two datasets within a single dataset; each sample of both input datasets contributes to one sample of the result with merged region schema. The merging of two

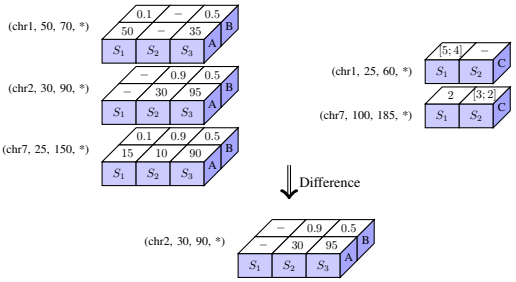


Fig. 10. Example of Difference operation.

schemas is performed by adding the schema of the second dataset to the schema of the first one; missing fields are set to NULL value. Fig. 11 illustrates how the resulting array is built after applying the Union operation. The two input arrays has one common region on chromosom 7, which is aggregated in the result.

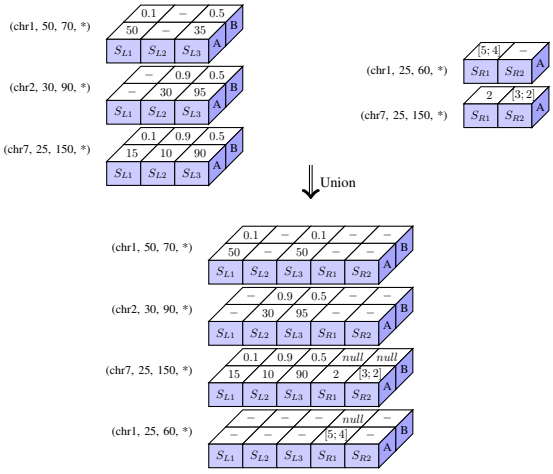


Fig. 11. Example of Union operation.

IV. Implementation

The implementation of each operation using the array data model is facilitated by the modular architecture of the GMQL system as consolidated in [11], whose main building blocks are shown in Fig. 12. Many language interfaces are all served by the same Scala API; the dataflow for a given query is an internal Acyclic Directed Graph (DAG), whose nodes correspond to calls to operation and whose arcs describe parameter passing. Typically, every GMQL operation is mapped into a workflow of 4-5 smaller granularity DAG operations, which are designed for optimizing sharing and reuse among GMQL operations. Using a Service Manager, the DAG operations are passed to the available implementations; in the past, we supported implementations in Flink [17] and SciDB [18], but our currently maintained implementation is based on Spark with the row-based implementation. In order to support the array-based data model, we developed a second implementation in Spark, shown as the red block of Fig. 12; availability of paired row-based and array-based implementations allows us

to highlight the differences between them (in this section) and then compare their performances (in the next section).

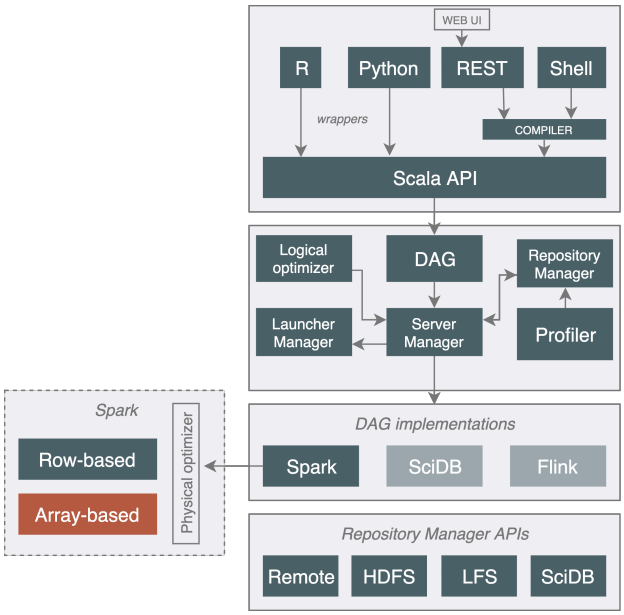


Fig. 12. Architecture of the GMQL system

A. Array Model Representation using Spark RDDs

In the design of an internal data structure for arrays, we were inspired by MLlib, a popular Apache Spark library for machine learning [19]. The main purpose of ML applications is to ingest huge matrixes of training sets so as to repeatedly train classifiers. Matrixes in MLlib are represented as vectors inside a Spark RDD; to overcome issues due to data sparsity, matrixes are represented as vectors of two arrays, where the former is an internal index pointing to the latter, which in turn is a compact data array.

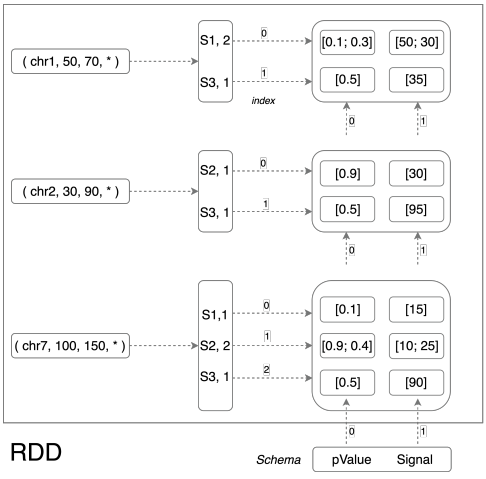


Fig. 13. Organization of Spark RDDs to support the array data model

The data organization used for storing the array data model within a Spark RDD is shown in Fig. 13; note that its data content is equivalent to the data content of Fig. 1, which uses the row-based model. In comparison, our data organization is more complex than the MLlib data representation, because regions can be replicated either inside a sample (intra-replication) or across samples (inter-replication).

The core of the structure is a key/value record in Scala, where the key (named *region key*) is a quadruple of region coordinates. The value is further sub-structured as two paired arrays, respectively called *replication vector* and *attribute vector*, where the former behaves as a map for accessing the latter. As shown in Fig. 13, each region key points to several entries of the replication vector (their number corresponds to the inter-sample replication). In turn, the replication vector contains pairs `(sample-id,repl)`, where the latter indicates the number of intra-sample replicates; each entry of the replication vector points to an entry of the attribute vector. Thus, the region denoted by cardinalities `<chr1,50,70,*>` is replicated within samples S1 and S3, and in turn sample S1 has two replicates of that region.

Data within the attribute vector is organized as arrays of arrays, where the enclosing array has a fixed number of entries, dictated by the schema; in the particular example of Fig. 13 the enclosing array has two coordinates, corresponding to attributes `pvalue` and `signal`. The internal array has a number of values for each attribute that depends on intra-replication, thus the internal array for the first entry pointed by `<chr1,50,70,*>` and sample S1 has two values - note that values must be paired to recover row values, e.g. values `<0.1,50>` and `<0.3,30>` correspond to the pvalue and signal of two regions of sample S1 with the above coordinates.

The array data model is implemented as a key/value record in Scala. The region coordinate act as key (denoted as RegionKey), the value (denoted as RegionData) is further structured as two arrays respectively storing the data replication map and the attribute values. RegionKey is used by the Spark implementation for data partitioning, according to a partitioning function. Listing 1 describes the Scala data types; GValue used for representing attribute values is a Trait in Scala with several implementations: GInt, GDouble, GString, and GNull.

```
1 ArrayModel(key:RegionKey,value:RegionData)
2
3 RegionKey (chrom:String, start:Long, stop:Long, strand:Char)
4
5 RegionData (Replication:Array[(Long, Int)],Attribute:Array[Array[Array[GValue]]])
```

Listing 1. Scala Types for the Array Data Model.

The implementation of several operations in GMQL requires binning [11]; the method assigns regions to bins (i.e., partitions of the genome) whose size typically ranges between few hundreds and several thousands of base pairs; through suitable algorithms, operations are executed in parallel at each bins, and then results are gathered and processed to recover the correct resulting regions. The effect of binning is to generate additional region replication, as regions which fall within many bins have as many replica as the number of overlapping bins. In order to speed up operations which involve binning, the bin number is added to the region key, which therefore becomes: `<chrom, start, stop, strand, bin>`. The

representation of Region Data is not changed – it is identical for every replicated region.

We next illustrate the most interesting aspects of the array-based implementation, by using the row-based implementation described in [11] as baseline; we discuss simple unary operations, then address Cover (which includes Histogram), then Join and Map, the most interesting binary operations.

B. Simple Unary Operations

Unary Operations are space-localized, hence they are executed by first using the region key to perform data access and then applying a function to its value, independently from the content of other regions, for producing the resulting region. For example, in case of *Select* operation, the selection predicate is separated into two components, respectively on the coordinate key and on the value; for each input region r_i we first check the region predicate; if it is satisfied, we produce an output region r_i' when some data in the region's value exists for which the value predicate is true, as illustrated in Algorithm 1. From the point of view of Spark, these operations consists of one **FlatMap** block that iterates over the input regions and applies an operation-specific function.

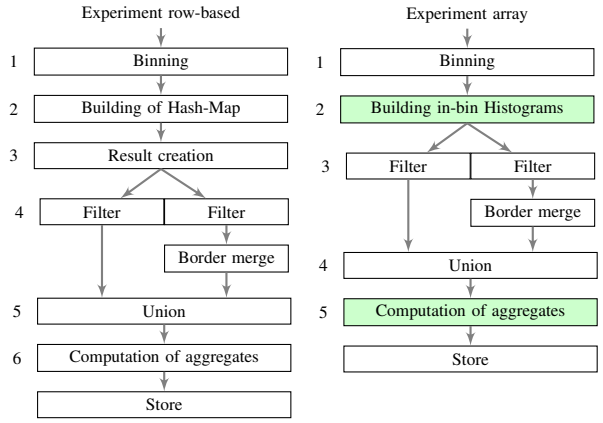
Algorithm 1: Select operation

Input : Array M and a predicate $P = Pc \wedge Pv$
Output: Array RES
$RES \leftarrow \varnothing$
foreach *region* $r_i \in M$ **do**
begin
  if *P(c_i) and* $\exists$ $v \in v_i$*: P(v_i)* **then**
    RES.insert(r_i)
end

C. Cover

In contrast to simple unary operations, all the other operations described in this section require complex processing by means of multiple Spark blocks. In all these operations, given the size of the genome, we opt for a parallel execution rather than a sequential one; this requires the use of *binning*, hence we perform a preliminary transformation of the data model, as discussed in Section IV-A - in a similar way, bin numbers are added to region coordinates in the row-based model, see [11]. Instead of using Spark listings, we describe them using logical blocks; Fig. 14 illustrates the algorithm blocks for implementing a cover operation. Since the purpose of this section is to illustrate the differences between row-based and array-based implementations, we start by describing the former, and then we describe how the latter differs; green blocks indicate substantially different implementations.

Computing the *Cover* requires computing the *Histogram*; for this, the row-based algorithm consists of scanning the genome from left to right and maintaining an accumulation count. At every start of a region the count is incremented, and at every stop is decremented; the result is given by intervals associated with the same counter, skipping all the bin's ends. Final steps



(a) Row-based (b) Array-based

Fig. 14. Operators for encoding the COVER algorithm

consist of selecting the *Cover* results out of the *Histogram* and associating its regions with aggregate functions as specified in the operation.

- Block 1 (Binning) 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.
- Block 2 (Building of Hash-Map) is responsible of grouping the output dataset of the previous block by chromosome and bin. Then 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.
- Block 3 (Result creation) returns the list of produced regions, along with their accumulation value, with each region placed within a bin, thus creating a raw histogram.
- 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.
- Block 5 (Union) performs the union of the regions separately produced.
- Block 6 (Computation of aggregates) computes aggregate functions as specified in the COVER operation (if any) using a Genometric Map operation.

The array-based algorithm takes advantage of the replication count available in the replication vector, that indicates the intra-replication factor of each region (see Fig. 13). The main difference between the row-based and the array-based method is in Block 2, which is executed within each bin by doing a sequential scan of regions and keeping a cache. If the cache

is empty, the current region is added to the cache. with an associated value equal to the replication count of the region. Otherwise the current region is compared with all the regions in the cache. When a cached region is found to be before the current one, it is removed from the cache and added to the output (given the ordering, if a region does not intersect with the current one, it will not intersect with any subsequent region). Instead, if a region in the cache is intersecting the current one, it is "split" into two, and the counters of the common part are summed up.

The array-based algorithm takes advantage of the key-value structure of the model also in Block 5, describing how aggregations are produced. With the array-based solution, the values of the contributing regions are accumulated within the value part while the computation proceeds, so aggregation functions can directly be applied. In the row-based implementation, this step requires a mapping to the input regions, to recover the original attribute values.

D. Binary operations

Next we discuss three binary operations: Join, Map and Difference. All these operations are based on the intersections of reference and experiment regions, so they have common steps, however the internals of such common steps require different algorithms; a gray shade indicates the steps that need to be performed differently. Fig. 15 illustrates the algorithm for implementing binary operations with (a) row-based and (b) array-based models.

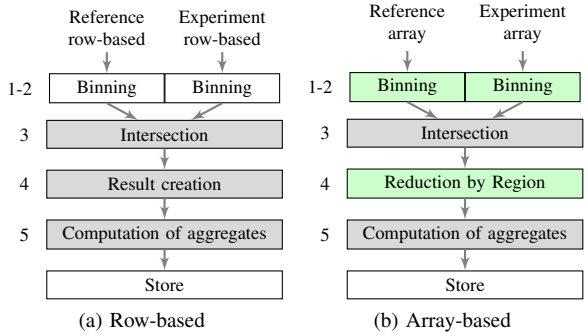


Fig. 15. Operators for encoding binary algorithms

The row-based method requires to bin the two datasets, to group them by sample pair, chromosome and bin number, to compute intersections within the bins, to compute aggregate functions (if necessary), and output the results. By contrasting the row-based and array-based implementation, we note that blocks 1-2 (Binning) and blocks 4 (reduction by region) are simplified by the array data model, as the region keys are not replicated. Thus, binning is simpler and bin reconstruction takes advantage of such simplification. The construction of region coordinates in block 3 is however more complex in the array-based algorithm for joins with the `intersect` and `contig` options, which are not region-preserving, as they require connecting regions in a way that is not directly supported by the region key.

E. Union

In a union, compatible attributes from two input datasets are merged, and then incompatible attributes are listed at the end, by putting the reference first and the experiment second. Fig. 16 illustrates the operations required for implementing a union operation using the two models.

The row-based method is very simple - it first updates the values of experiment regions (Block 1) and then just makes the union of the two datasets (Block 2). As the array-based model requires to have non replicated regions (reflected in their coordinates), the array-based method is more complex; it first requires a join of the two arrays by their region coordinates, and a rearrangement of region cells when the coordinates exactly match (Block 1) . Then, values of experiments are updated (Block 2) to become compatible with the reference array schema and then reference and experiment values are merged (block 3). Blocks 2-3 can be performed together.

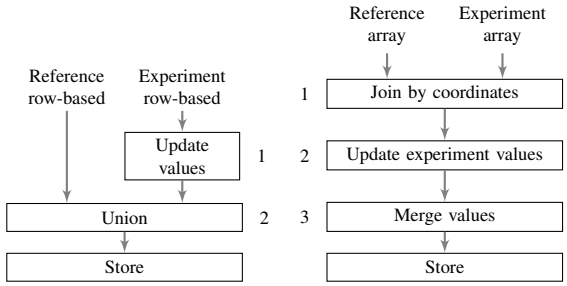


Fig. 16. Operators for encoding the Union algorithm

V. Experiments And Evaluation

A. Experimental Setup

We conducted our experiments on our server, which contains an Intel® Xeon® Processor E5-2650 at 2.00 GHz (32 hyper-threads), 192 GB of RAM, and 4x2 TB hard disks. The software stack includes Apache Hadoop 2.7.2 and Apache Spark 2.2.0.

B. Performance Evaluation at Operation Level

We first compare the row-based and array-based implementations for each operation separately. For unary operations we generated a dataset that consists of 5 samples, contains a total of 23 million regions, where each region consists of 5 attributes. For binary operations, which have Reference and Experiment input datasets, we generated a Reference dataset that consists of 5 samples with 3 attributes in the schema and a total of 5 million regions; we then used the dataset of unary operations as Experiment dataset.

Fig. 17 shows the performance comparison for unary operations. The most expensive unary operations are *Cover* and *Group*; in both cases, the array-based implementation outperform the row-based implementation, as the array-based takes advantage of the region key. The row-based implementation is slightly more efficient in the *Select, Project* and *Merge*

operations; these operations have an additional overhead due to the unfolding of array structures.

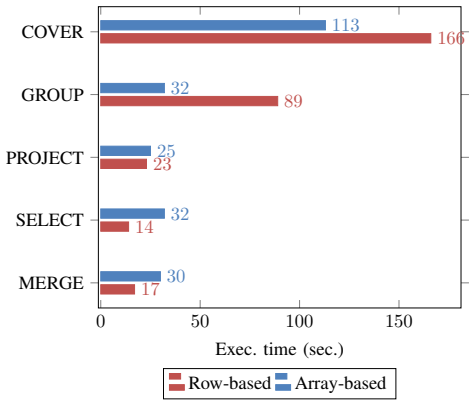
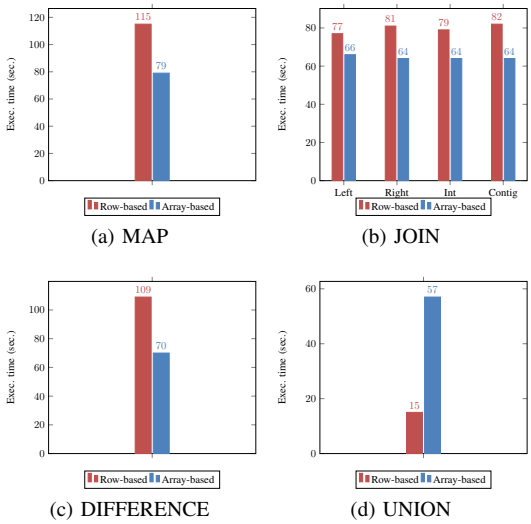


Fig. 17. Execution times of unary operations



(a) MAP (b) JOIN

(c) DIFFERENCE (d) UNION

Fig. 18. Execution times of binary operations

Fig. 18 shows the performance comparison of binary operations; each array-based operation except *Union* slightly outperforms its rival row-based implementation; the implementation of these operations uses region intersection, which is facilitated by the array model. The *Union* operation has a simpler implementation in the row-based algorithm, as discussed in Section 4.E.

C. Full Benchmark

A thorough performance comparison has been possible thanks to the availability of a set of queries defined in the context of the STQL language [15] and encoded in GMQL in the supplemental material of [16]; as STQL does not use metadata, the queries in the benchmark were only addressing the regions.

For the complex query CQ5 of the benchmark we show in Listing 1 its translation into calls to the Scala API; Fig. 19 shows the operation dataflow as a DAG. A query is

```
def CQ5(path: String): Unit = {
  val step1results = Import(path + "/GENCODE")

  val fun1 = DefaultRegionExtensionFactory.get(RELEFT(),Left("original_left"))
  val fun2 = DefaultRegionExtensionFactory.get(RERIGHT(),Left("original_right"))
  val fun3 = DefaultRegionExtensionFactory.get(RESUB(RESTART(),
     REFloat(1500)),Right(COORD_POS.START_POS))
  val fun4 = DefaultRegionExtensionFactory.get(READD(RESTART(),
     REFloat(500)),Right(COORD_POS.STOP_POS))

  val step2results = Project(None, Some(List(fun1, fun2, fun3, fun4)),
     step1results)
  val t2 = Import(path + "/T2")
  val t3 = Import(path + "/T3")
  val t4 = Import(path + "/T4")
  val t5 = Import(path + "/T5")
  val t2_on_step2 = GenometricMap(step2results, t2, 5000)
  val t3_on_step2 = GenometricMap(t2_on_step2, t3, 5000)
  val t4_on_step2 = GenometricMap(t3_on_step2, t4, 5000)
  val t5_on_step2 = GenometricMap(t4_on_step2, t5, 5000)

  val fun5 = DefaultRegionExtensionFactory.get(REDIV(RESUB(REPos(25), REPos(26)),
     RESUB(REPos(27), REPos(28))),Left("folding_value"))
  val step6results_0 = Project(None, Some(List(fun1, fun2, fun5)), t5_on_step2)
  val res = Select(Some(Predicate(29, REG_OP.GT, 3)), step6results_0)
}
```

Listing 2. CQ5 Scala GMQL complex query example.

executed by recursively traversing the DAG from the result leaf, producing the sequence of calls that generate it from the query input, and then executing operations in an arbitrary but consistent order, passing parameters and results according to that order. As shown in Fig. 12, operations inside the DAG are implemented using both the row-based and array-based methods.

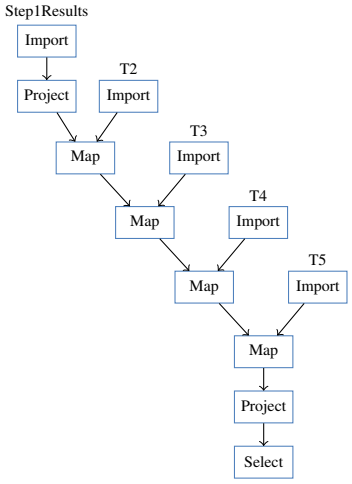


Fig. 19. DAG of CQ5 query

Benchmark queries are classified as simple and composite queries, based on their data size and complexity; Fig. 20 shows the performance of eight simple queries. Queries SQ3, SQ6, and SQ8 consist of a single operation running on biological data from ENCODE public repository [2]. The performance for most of the simple queries for both the array-based and row-based models is comparable. SQ4 shows the most significant performance improvement among for the array-based implementation, as it achieves 2.4× speedup. It consists of three nested *Cover* followed by a *Union* operation; the speedup of the the array-based implementation of the *Cover* compensates for the decrease of performance of the *Union* and grants an overall advantage to the array-based implementation.

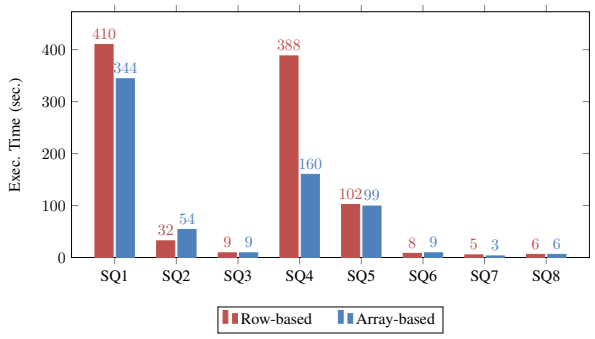


Fig. 20. Performance of 8 simple queries (SQ1-SQ8)

Fig. 21 shows the comparative performance of the row and array-based implementations upon the complex query set, consisting of three medium size queries (`CQ1`, `CQ2` and `CQ6`, whose execution time stays below 5 minutes) and three large size queries (`CQ3`, `CQ4` and `CQ5`, whose execution time is significantly more and approximates 6 hours in the case of `CQ5`). Thanks to the complex interplay of operations, the array-based implementation outperforms the row-based implementation in all cases, with speedup of $5\times$, $19\times$ and $35\times$ respectively for `CQ3`, `CQ4` and `CQ5`.

The query `CQ3` has a region-preserving chain of two *Map* and *Join* operations, then four *Cover* and one *Union*. The query `CQ4` consists of two *Map* operations with a *Select-Group-Merge* sequence between them. The `CQ5` query has a region-preserving chain of 4 *Map* operations followed by a *Project-Select* chain, as illustrated in Fig. 19. The huge speed-up is motivated both by the existence of many region-preserving operations and by the comparatively smaller size of intermediate data, which benefit from the efficient accumulation of cells with the same region coordinates.

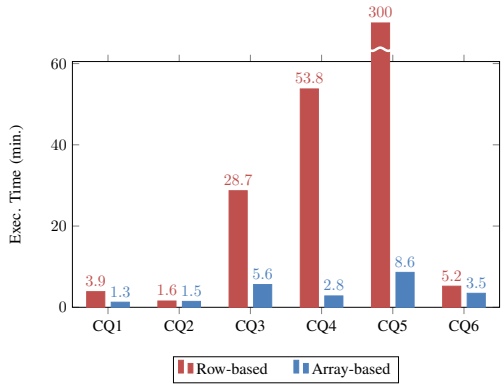


Fig. 21. Performance of 6 composite queries (CQ1-CQ6)

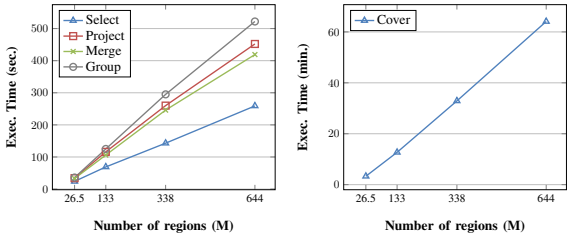
D. Scalability

For testing scalability, we designed four scenarios of use, respectively named *small*, *medium*, *big* and *large*, whose features are summarized in Table II. We run unary operations upon the 4 cases, and binary operations by using a new dataset with 2 million regions as Reference and the four above datasets as Experiments. We introduced replication in the row-based data model (using a fixed replication factor $\beta \approx 2$), regardless of the mapping of regions to samples; thus, replication is mostly inter-sample.

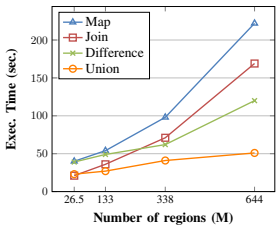
TABLE II
FEATURES OF THE DATASETS USED IN THE SCALABILITY TESTS ($\beta \approx 2$)

Name	Size	Total # regions	Regions in array	Samples
small	1.6 GB	54 M	26.5 M	400
medium	8.5 GB	285 M	133 M	2000
big	22.1 GB	740 M	338 M	5000
large	43.2 GB	1445 M	644 M	10 000

Execution times of unary operations are illustrated in Fig. 22 (a,b). Note that both space-localized and space-rearranged operations scale linearly. Execution times of binary operations are illustrated in Fig. 22 (c); scalability is no longer linear for *Map* and *Join*, but this is to be expected due to the greater complexity of these operations with sizes.



(a) Unary Space Localized (b) Unary Space Re-arranged

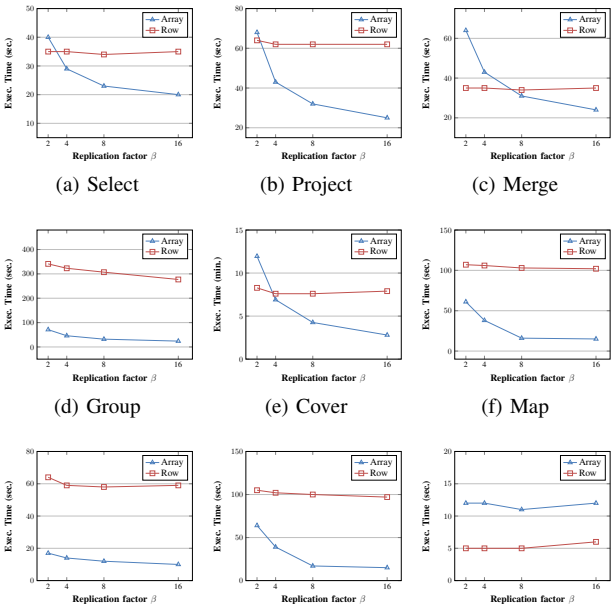


(c) Binary

Fig. 22. Execution times of array-based solution with increasing data sizes

Fig. 23 shows the execution times of array-based and row-based solutions using datasets with approximately equal number of regions and size but different replication factor β. The array-based solution dominates over the row-based solution in GROUP, MAP, JOIN and DIFFERENCE and is dominated by the row-based solution in UNION. In the other operations, the array-based solution has better performance than the row-based solution with large values of the replication factor; the break-even replication factor is different for the various operations, it is smaller (below 2) for SELECT and PROJECT than for COVER and MERGE.

We next consider a cluster of different size, ranging from 2 to 16 nodes. We performed our experiments on the Amazon Web Services (AWS) cloud, using a configuration with c5.9xlarge machines, each with 32 virtual CPUs, 72GB of



(a) Select (b) Project (c) Merge

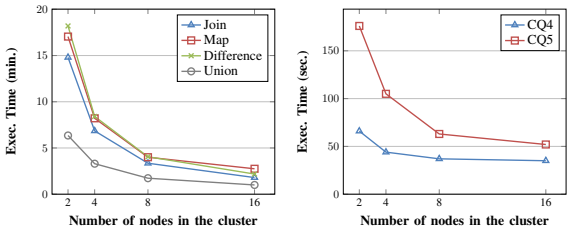
(d) Group (e) Cover (f) Map

(g) Join (h) Difference (i) Union

Fig. 23. Scalability evaluation of array-based solution with increasing replication

memory, and 576 GB of storage. The testing setup contained one driver node and four configurations of slave nodes, set at 2, 4, 8 and 16 nodes respectively.

Fig. 24 shows the scalability of binary operations on different cluster sizes. We generated two datasets, Reference and Experiment, each with replication equal to 10. Reference dataset has size of 43 MB and consists of 10 samples and 1 million regions in total. Experiment dataset has size of 93 GB and consists of 1000 samples and almost 2 billions regions in total. We also show the scalability of queries CQ4 and CQ5, the most demanding queries in the benchmark of Section V-C.



(a) Binary Operations (b) CQ4 and CQ5 applications

Fig. 24. Execution times of array-based solution with increasing nodes.

VI. Related Work

Our work can be considered as a follow-up of [13], presented at ICDE 2019. In [13] we limited our analysis to *region-preserving operations*; we used the array-based model only for the operands whose regions were preserved, whereas we used the row-based model for the other operands. Hence, we could not consider *Histogram, Cover, Union* and *Contig/Intersect Join*; moreover, we approached complex queries by converting from the row-based model to the array-based model prior to each chain of region-preserving operations, and from array-based to row-based at the end of each chain. The array-based data model provided significant speed-ups of queries with long region-preserving chains, but was not a full-fledged data model capable of encompassing any operation.

We next discuss array-based implementations for genomics, by distinguishing native and Spark-based implementations. As domain-specific join operations in genomics are similar to theta-join, they are not well optimized by array-based organizations, which typically use indexes for *slicing and dicing* arrays along dimensions. We also experienced that encoding genomic operations using SQL or SQL-like languages, e.g. as those provided in RasDaMan and SciDB, is rather difficult.

1) SciDB [18]: It provides its own shared-nothing storage layer; it provides a variety of optimizations, like dealing with overlapping chunks and data compressions. We considered SciDB as a possible target engine and built a full implementation of the main GMQL abstractions using it, presented in [14], focused on representative operations (filter, aggregate, map, join). At the time of our experiments, SciDB performed faster in filtering and aggregation over an array-based physical data organization, but Spark performed faster on massive binary operations (maps and joins). In [13] we compared two implementations of region-preserving array-based operations upon SciDB and Spark, showing that Spark had better performance than SciDB on the AWS cloud for a specific query consisting of a chain of Map operations (similar to query CQ5).

2) TileDB [20]: It stores multi-dimensional array data in fixed size data tiles, optimized for both dense and sparse multi-dimensional arrays. It supports the same data model as SciDB, but is currently a storage manager for array data rather than an array database. GenomicsDB [21] is built on top of the 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.

3) ChronosDB [22]: It is a distributed array DBMS for geospatial data with command-line tools; it provides a formal multidimensional array data model to abstract from the files and the tools. ChronosDB wraps the geospatial tools and neither supports Genomic data nor modifies the core algorithms of the tools to support the array format. Two extensions of Spark for supporting array data processing are not adapted to genomics.

4) SciSpark [9]: 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. Therefore, SciSpark can repetitively manipulate multiple array datasets at runtime. SciSpark requires users to provide custom partitioning and file-loader functions.

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

VII. Conclusion

In this paper, we show that by coupling the array data model with the Spark execution engine we obtain a high-performance solution for supporting region-based operations for genomics. While in previous work we considered the subset of region-preserving operations, in this work we cover the full spectrum of GMQL region-based operations; for each of them, we first intuitively describe the meaning of operations upon an array representation, then we discuss a method for their implementation. For managing arbitrary operations upon regions, we designed an internal representation adapted to the needs of supporting region-based calculus within Spark RDDs; performance analysis shows that our array-based solution is superior to a row-based solution in most operations and benchmark queries, especially with large replication factors which occur in many practical cases.

We plan to apply our multi-dimensional data management to spatial and temporal applications; as we can map genomic coordinates to the longitude and latitude of locations in spatial data or time intervals of temporal data, these applications are similar in data types and operations to genomic applications. In particular, the genomic join can be mapped to spatial/temporal batch join between two datasets of locations/time-intervals. Our multi-dimensional join shows promising results when it produces several matches which must be ranked; this directly maps to spatio-temporal queries where many equivalent options are available, e.g. find minimum distance offices of public or private organizations closest to given locations (e.g. for all banks, the closest bank office from home), or the closest time events in different countries when a certain climate event occurred (e.g., for each nation/state/region, the event closest in time to Xmas 2019 when temperature was higher than 40 degrees Celsius).

Spark is a very popular big data engine, and several ongoing efforts are focused upon improving its use for big data analysis; in particular, large in memory data structures can be exploited to improve the interconnection to machine learning algorithms (e.g. [24]). In our future research, we plan to investigate the direct use of arrays, as produced by genomic queries, for training machine learning models.

Acknowledgments

This research is funded by the ERC Advanced Grant project 693174 "GeCo" (Data-Driven Genomic Computing), and is also supported by an AWS Machine Learning Research Award[2].

References

[1] 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*, vol. 13, no. 7, p. e1002195, 2015.

[2] ENCODE Project Consortium, "An integrated encyclopedia of dna elements in the human genome," *Nature*, vol. 489, p. 57, 2012. [Online]. Available: https://doi.org/10.1038/nature11247

[3] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, J. M. Stuart, and Cancer Genome Atlas Research Network, "The Cancer Genome Atlas Pan-Cancer analysis project," *Nature genetics*, vol. 45, no. 10, pp. 1113–1120, Oct 2013. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/24071849

[4] N. Siva, "1000 Genomes project," *Nature Biotechnology*, vol. 26, no. 3, pp. 256–256, 2008. [Online]. Available: https://doi.org/10.1038/nbt0308-256b

[5] M. Caulfield, J. Davies, M. Dennys, L. Elbahy, T. Fowler, S. Hill, T. Hubbard, L. Jostins, N. Maltby, J. Mahon-Pearson, and et al., "The National Genomics Research and Healthcare Knowledgebase," Dec 2017. [Online]. Available: https://figshare.com/articles/GenomicEnglandProtocol_pdf/4530893/5

[6] ICGC. (2019, oct) International cancer genome consortium. https://icgc.org/. [Online]. Available: https://icgc.org/

[7] 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*, vol. 31, no. 12, pp. 1881–1888, 2015.

[8] M. Bertoni, S. Ceri, A. Kaitoua, and P. Pinoli, "Evaluating Cloud Frameworks on Genomic Applications," *IEEE Big Data Conference, Santa Clara*, Nov. 2015.

[9] 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)*, pp. 2020–2026, 2015.

[10] Apache. (2019, oct) Spark. https://spark.apache.org/. [Online]. Available: https://spark.apache.org/

[11] A. Kaitoua, P. Pinoli, M. Bertoni, and S. Ceri, "Framework for supporting genomic operations," *IEEE Transactions on Computers*, vol. 66, no. 3, pp. 443–457, 2017.

[12] A. Gulino, A. Kaitoua, and S. Ceri, "Optimal binning for genomics," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 125–138, Jan 2019.

[13] O. Horlova, A. Kaitoua, V. Markl, and S. Ceri, "Multi-Dimensional Genomic Data Managment for Region-Preserving Operations," in *35th IEEE International Conference on Data Engineering (ICDE)*, April 2019, pp. 1166–1177.

[14] 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.

[15] X. Zhu, Q. Zhang, E. D. Ho, K. H.-O. Yu, C. W. Liu, T. H.-M. Huang, A. S.-L. Cheng, B. Kao, E. Lo, and K. Y. Yip, "START: a system for flexible analysis of hundreds of genomic signal tracks in few lines of SQL-like queries," in *BMC Genomics*, 2017.

[16] M. Masseroli, A. Canakoglu, P. Pinoli, A. Kaitoua, A. Gulino, O. Horlova, L. Nanni, A. Bernasconi, S. Perna, E. Stamoulakatou, and S. Ceri, "Processing of big heterogeneous genomic datasets for tertiary analysis of Next Generation Sequencing data," *Bioinformatics*, 2018, dOI: http://dx.doi.org/10.1093/bioinformatics/bty688.

[17] Apache. (2019, oct) Flink. https://flink.apache.org/. [Online]. Available: https://flink.apache.org/

[18] P. 4. (2019, oct) SciDB array DataBase. http://www.paradigm4.com/. [Online]. Available: http://www.paradigm4.com/

[19] Apache. (2019, oct) Spark mllib. https://spark.apache.org/docs/latest/mllib-data-types.html. [Online]. Available: https://spark.apache.org/docs/latest/mllib-data-types.html

[20] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The TileDB array data storage manager," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 349–360, 2016.

[21] Intel. Intel. GenomicsDB. [Online]. Available: https://github.com/Intel-HLS/GenomicsDB/wiki

[22] R.A. Rodriges Zalipynis, "ChronosDB: Distributed, File Based, Geospatial Array DBMS," *PVLDB*, vol. 11, no. 10, pp. 1247–126, 2018. [Online]. Available: https://doi.org/10.14778/3231751.3231754

[23] W. Wang, T. Liu, D. Tang, H. Liu, W. Li, and R. Lee, "SparkArray: An Array-based Scientific Data Management System Built on Apache Spark," *IEEE International Conference on Networking, Architecture and Storage (NAS)*, August 2016.

[24] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann, "SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases," in *EDBT*, 2017.

[2]https://aws.amazon.com/aws-ml-research-awards/