Evaluating Genomic Big Data Operations on SciDB and Spark

Simone Cattani, Stefano Ceri, Abdulrahman Kaitoua, and Pietro Pinoli

Dip. Elettronica, Informazione e Bioingegneria - Politecnico di Milano
fistname.secondname@polimi.it

Abstract. We are developing a new, holistic data management system for genomics, which provides high-level abstractions for querying large genomic datasets. We designed our system so that it leverages on data management engines for low-level data access. Such design can be adapted to two different kinds of data engines: the family of scientific databases (among them, SciDB) and the broader family of generic platforms (among them, Spark). Trade-offs are not obvious; scientific databases are expected to outperform generic platforms when they use features which are embedded within their specialized design, but generic platforms are expected to outperform scientific databases on general-purpose operations.

In this paper, we compare our SciDB and Spark implementations at work on genomic abstractions. We use four typical genomic operations as benchmark, stemming from the concrete requirements of our project, and encoded using SciDB and Spark; we discuss their common aspects and differences, specifically discussing how genomic regions and operations can be expressed using SciDB arrays. We comparatively evaluate the performance and scalability of the two implementations over datasets consisting of billions of genomic regions.

1 Introduction

Next Generation Sequencing (NGS) is a technology for reading the DNA that is changing biological research and will change medical practice; thanks to the availability of millions of whole genome sequences, genomic data management may soon become the biggest and most important "big data" problem of mankind, and bringing genomics to the cloud is becoming more and more essential. In this context, we are currently developing a new, holistic approach to genomic data modelling and querying that uses cloud-based computing to manage heterogeneous data produced by NGS technology [12][1]. Our approach is based on a new, high-level query language, called GenoMetric Query Language (GMQL) [13], which enables building new datasets from a repository of existing datasets, using algebraic operations.

[1] Advanced ERC Grant, http://www.bioinformatics.deib.polimi.it/geco/

The current implementation of GMQL, described in [12], uses Flink [3] and Spark [4]. We recently considered another target for GMQL, opting for a scientific data management system. Scientific databases are known to for their efficient support of data aggregation over several dimensions, which is crucial for genomics. Among the various alternative systems, we selected SciDB, because it supports an add-on specifically dedicated to tertiary data analysis for genomics [1]; thus, it is an ideal alternative implementation framework for GMQL.

In this paper, we closely compare Spark and SciDB at work on genomic queries. We describe four widely used genomic abstractions: region selection, aggregation, histogram and mapping; by composing them, we obtain a significant subset of domain-specific operations of GMQL and of the other tools for genomic region management. We built a big data benchmark with a large dataset of regions and samples: the largest test compares half million regions to hundred millions regions, scattered over two thousand samples, corresponding to 50 trillion potential region intersections.

This paper demonstrates that both Spark and SciDB can manage such workload and qualify as relevant candidates for hosting tertiary genomic data analysis. Our benchmark demonstrates the superiority of SciDB in computations which perform selections and aggregations, but also shows that Spark outperforms SciDB in computations that perform genome-wise region comparisons; in such cases, both the SciDB and Spark computations use **binning**, a method for partitioning the genome into disjoint portions so as to enable parallelism.

The organization of this paper is as follows. Section 2 briefly introduces SciDB and Spark; Section 3 explains the Genomic Data Model and its mapping to an array database such as SciDB. Then, Section 4 provides a high-level description of the encoding of genomic abstractions using Spark and SciDB, and Section 5 provides their benchmark. Conclusions summarize our findings.

2 Platform Features

Apache Spark [18, 20, 21] is a general-purpose data processing engine providing high-level data operators and making a more efficient use of memory as compared with low-level map-reduce programming. SciDB [6, 16, 17] is a computational multi-dimensional database engine optimized for fast data selection and aggregations, required by most scientific applications.

Spark. The programming model of Spark is based on an abstraction called *resilient distributed datasets* (RDDs); a RDD is a distributed, fault tolerant data collection which can be processed on large servers or clusters. RDDs empower Spark with the support for in-memory data processing by allowing the state of the memory to be shared across different jobs. On contrary, for conventional MapReduce systems, sharing of intermediate data is only possible through write and read on persistent storage (e.g., the distributed file system), incurring significant cost for loading the data and writing it back at each stage.

Spark includes a rich set of operators, including *map*, *flatMap, mapPartition, reduce, repartition, filter, union, cartesian, coGroup, sortByKey, countByKey*;

the above operations are also denoted as *transformations*, as they produce RDDs from either RDDs or input files, whereas other operations are denoted as *actions*, as they do not produce RDDs, but instead they either pass a result set to the embedding program or write data to the disk We opted for RDDs over Spark's DataFrame so as to keep the Spark and Flink deploys similar, as Flink operators are similar to RDD operators.

SciDB. The database engine of SciDB is based on a native array data model. Each array is described by a list of dimensions and a list of attributes. Dimensions have the integer type and each combination of them defines a cell in the array; attributes have arbitrary types, and consequently each cell is an vector of correspondingly typed values. Each array is implemented as a specific data structure, managed by the SciDB engine. Arrays are divided into chunks, where the chuck size is an important parameter under the control of the database designer. An hash function uses the dimension values associated to each chunk in order to assign it to a specific node of the cluster; by using this method, called *Multidimensional Array Clustering*, every query processing operation is mapped to specific chunks and executed in parallel at the nodes where such chunks are allocated.

SciDB queries are programmed using the Array Functional Language (AFL), a query language where each operation is defined as a function that receives as input either one or two arrays and produces as output one array; operations can be nested. The operations of AFL include: `filter` for selecting the array elements that satisfy a Boolean condition, `between` for extracting a rectangular region of the array, `cross_join` for pairing arrays (using equi-join on dimensions to speed up the computation), `redimension` to promote attributes to dimensions or to deprecate dimensions to attributes.

3 SciDB representation of the Genomic Data Model

The Genomic Data Model (GDM), biologically motivated in [14], is based on the notions of datasets and samples. Datasets are collections of samples carrying the same region schema. Each sample consists of two parts, the *region data*, which describe portions of the DNA and their features, and the *metadata*, which describe general properties of the sample.

GDM Model. A genomic region r is a portion of the genome defined by the quadruple of values $\langle chr, left, right, strand \rangle$, called *region coordinates*, where *chr* is the chromosome, $left$ and $right$ are the two ends of the region along the DNA coordinates; strand is encoded as either $+$ or $-$, and can be missing. Formally, a *sample* s is a triple $\langle id, R, M \rangle$ where:

- id is the sample identifier of type *long*.
- R is the set of regions of the sample, built as pairs $\langle c, f \rangle$ of coordinates c and features f.Coordinates are arrays of four fixed attributes *chr*, *left*, *right*, *strand* which are respectively typed *string*, *long*, *long*, *string*. Features are arrays of typed attributes; we assume attribute names of features to be different, and their types to be arbitrary. The *region schema* of s is the list

of attribute names and types used for the identifier, the coordinates and the features.
- M is the set of metadata of the sample, built as attribute-value pairs $\langle a, v \rangle$, where we assume the type of each value v to be *string*.

A *dataset* is a collection of samples with the same region schema and with features having the same types; sample identifiers are unique within each dataset.

SciDB Representation of GDM. We store metadata into a cube where the three dimensions are: *attribute name*, *value* and *sample id*. Given that SciDB does not permit *string* as dimension type, we introduced hashing for attribute name and value as dimensions. As the hashing of strings into 64-bit integers (standard dimension type in SciDB) introduces a high risk of collision errors, we opted for a double hashing, i.e. we used both for attribute name and value two different hashing functions, specifically selecting two orthogonal hash functions [9]. In conclusion, metadata are stored into a single 5-dimensional array.

```
DS_MD = <name:String,value:String>[sid, nid_1, nid_2, vid_1, vid_2]
```

where `sid` is the sample id and (`nid_1`,`nid_2`) and (`vid_1`,`vid_2`) are produced by the double hashing for attribute name and value. The schema of metadata arrays is identical for all the GDM datasets imported into SciDB.

Regions of a dataset are also stored into a single array; they are organized according to the relative sample id and genomic coordinate. In order to use region coordinates as dimensions for the SciDB data model, it is required to cast to integers the chromosome and strand values. For chromosomes, we defined a global codification map table that provides chromosome ids shared among all the datasets. This indexing operation is natively supported by SciDB through `uniq` and `index_lookup` operators. For strand, we applied a static conversion. Using these transformations, regions data are mapped to a 6-dimensional array, where attribute fields are based on the specific dataset feature schema provided by the user. The `x` dimension is an enumeration value, required because each GDM sample could have more than one region with the same coordinates.

```
DS_RD = < feature_schema > [sid, chr, left, right, strand, x]
```

Fig. 1 shows the representation of the two arrays `DS_MD` and `DS_RD`; hash values are truncated for a better visualization. The physical arrays were designed using columnar storage with respect to each attribute, algebraic indexing (built by using a combination of hashing as well as lookup structures that are automatically maintained as the data sizes grow), and clustering of logically contiguous regions. This makes *slice* or *between* queries very efficient. Arrays are stored within fixed-size rectilinear **chunks** that partition the multidimensional space. Each chunk is then assigned to a computational node, using a hash function over the chunk's coordinates; the usage of region ends as coordinates allows their storage based on real region proximity, a fundamental property in order to speed up domain specific operations that use range intersection or range selection. According to [2], the optimal size for a chunk should be between 5MB and 50MB. In our example, with a single attribute (and a size of about 8 Bytes), chunks with a million of regions have size of about 8-10 MB.

DS_MD

{sid,nid_1,nid_2,vid_1,vid_2}	name,	value
{1, 1020, 6526, 8844, 3474}	'avg',	'0.0'
{1, 5139, 2589, 8864, 6221}	'type',	'chiapet'
{2, 5139, 2589, 7534, 3123}	'type',	'chipseq'
{1, 2984, 8763, 1123, 8232}	'cell',	'hela-s3'

DS_RD

{sid,	chr,	left,	right,	strand,	x}	score
{1,	4,	1129,	3425,	1,	1}	0.5867
{1,	3,	5120,	9253,	1,	2}	0.7632
{1,	4,	3342,	3544,	0,	3}	0.3324
{2,	1,	4212,	7676,	1,	1}	0.9981
{2,	2,	1112,	1745,	1,	2}	0.7783
{2,	2,	1112,	1745,	1,	3}	0.7783
{2,	3,	5142,	7435,	1,	4}	0.5741

Fig. 1. Metadata and regions arrays imported into SciDB.

4 Genomic Operations

We next present four basic abstractions which are composed in several ways within GMQL operations; given that regions are several orders of magnitude greater than metadata, we focus on the operations which apply to regions.

4.1 Region Filtering

We consider three selection predicates: (a) by chromosome (coordinate), (b) by a region attribute and (c) by a conjunctive expression on both chromosome and score. Both implementations are straightforward.

A. Spark In the Spark implementation, regardless of the condition type, we just need to filter the RDD of the regions:

```
regionsRDD.filter(predicate)
```

B. SciDB The SciDB implementation is based on a simple `AFL` selection. In the SciDB model, chromosome is a dimension while the region attribute is placed within a cell. Therefore, case (a) is implemented simply as a dimension lookup and case (c) as a dimension lookup followed by a filter. The query of case (b) requires instead to scan all the chunks and then test the condition.

4.2 Region Aggregation

We consider an aggregate operation, such as `COUNT, AVG`, which is applied to all the regions of each sample of a dataset; the operation returns pairs `<SampleId, Value>`.

A. Spark The *Spark* implementation is based on grouping samples based on `SampleID` and then calculate the aggregation for each sample separately. To calculate the count, we use the following code:

```
DS_RDD.groupBy(x=>x.sampleId).mapValues(x=>x.size)
```

B. SciDB The AFL language of SciDB provides the `aggregate` operator which takes as input a SciDB array, a list of aggregate functions and (optionally) a list of dimensions along which to compute the aggregates. The mapping of Region

Aggregation to `aggregate` is straightforward, where the dimension corresponds to the sample ID. As an example, the AFL code for computing the count aggregate looks like:

```
aggregate(dataset_array, count(*),SampleID);
```

where the first argument of the `aggregate` is the input dataset, the second is the aggregation function and the last one is the aggregation dimension.

4.3 Region Histogram

A classic operation in genomics is to compute the *accumulation index*, i.e. for each position in the genome the number of regions which overlap with that position; the operation applies to all the samples of a dataset. Note that the regions of the three input samples `S1, S2, S3` have several overlaps, and the accumulation index ranges between 1 and 3, as shown in Fig. 2.

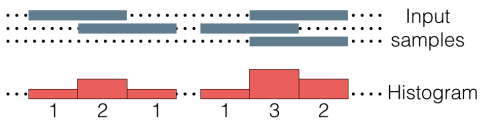


Fig. 2. Genomic histogram: computation of an accumulation index

A sequential algorithm for solving this problem consists of scanning the genome from left to right and maintain the accumulation count. Every time we meet the start of a region, we increment the count; conversely, every time we meet the stop of a region, we decrement the count. The result is made of all the consecutive couples of region ends (either starts or stops) between which the accumulation count is positive and does not change.

A. Spark A parallel and distributed Spark implementation for computing the histogram is described in detail in [5]. The implementation of histogram (and of most operations on genomic regions) is based on **mono-dimensional binning**, consisting in partitioning the genome into equal size, disjoint and consecutive segments, called **bins**, originally introduced for fast genome viewing on a browser (https://genome.ucsc.edu/). For each chromosome, the *i-th* bin spans from `i*BIN_SIZE` to `(i+1)*BIN_SIZE`. Then the histogram is computed in parallel for each bin, through a functional style algorithm based on the `reduce` operation; results of all the bins are merged in order to produce the histogram of the whole genome.

B. SciDB The computation of histograms by a SciDB program is complex and requires a good understanding of AFL primitives We provide a high-level description of the algorithm used for aggregating regions and of how such algorithm takes advantage of an array representation. Figure 3(a) shows that each `INPUT`

is represented as 6-dimensional arrays (recall Section 3); for the purpose of histogram evaluation we do not consider strands, therefore each region within a sample is characterized by chromosome, right end, left end, and x value. Thanks to suitable redimension and apply operations, we build two 2D matrixes, respectively called `LEFT` and `RIGHT`, illustrated in Fig. 3(b), where one dimension is the chromosome and the other dimension is a projection of the region respectively on the left and right end; the tables contain +M at every region's left end and -N at every region's right end, where M and N respectively denote the number of regions starting and ending at each base.

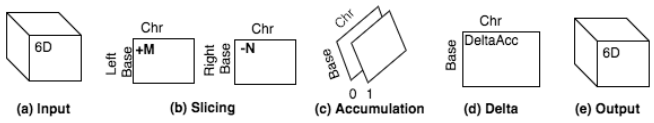


Fig. 3. Schematic representation of the steps for histogram computation in SciDB

Then, we merge the two matrixes into a 3D matrix, called `ACCUMULATION`, with a fictitious dimension valued (0,1), that pairs the two 2D matrixes (see Fig. 3(c)); at this point, the cells of the 3D matrix are collapsed by applying the sum aggregation on colliding cells, followed by a redimension; the output matrix, called `DELTA`, is shown in Fig. 3(d); its cell at a place (c,r) is an integer representing a count of regions starting or ending at position r of chromosome c (it can be a positive or negative value).

At this point, the actual histogram can be extracted for each chromosome and cell by applying the `cumulate` operation of SciDB to the `DELTA` matrix (it adds to each value within an array the sum of all its predecessors, e.g. `cumulate([4,-2,1,-2,2]=[4,2,3,1,3])`; results are positive numbers reflecting the organization of regions in the genome; a final application of apply and redimension returns the result as 6-dimensional array (Fig. 3(e)), where regions are characterized by a new attribute `ACCUMULATION` storing their accumulation index.

4.4 Region Mapping

A region mapping operation applies to two datasets, called `Reference` and `Experiment` respectively. This operation performs the intersection of `Experiment` samples over each `Reference` and then computes an aggregate over such intersection. This behavior is explained in Figure 4, where we show a simple case consisting of one sample of `Reference` and one sample of `Experiment` with overlapping regions, where we count the number of experiment regions which intersect with each reference region (e.g., the third region of the `Reference` intersects with 2 regions of `Experiment` and therefore its count is 2). Region mapping requires the computation of a particular kind of join between regions, which is satisfied

when two regions intersect. Joins with arbitrary distal conditions are discussed in [12].

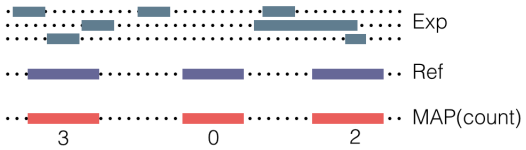


Fig. 4. Mapping experiments to references in genomics.

A. Spark The Spark implementation consists of two main steps: (a) binning and (b) checking for intersection. In the binning phase, the genome is divided into bins and every region of both the `Reference` and the `Experiment` datasets is assigned to all the bins it overlaps. Then, the datasets are left-joined on the key: `(id,bin,chromosome)`. The cross product of the regions within the same bin is then computed and the intersection condition is checked, consisting of testing for overlap (by considering the start and the stop of both regions) and then testing if one of the regions starts on the current bin, thus creating just one result for each matching pair. This condition generalizes a binning method presented in [8]. Finally, adjacent regions on contiguous bins are aggregated using a reduce phase, producing the final result.

B. SciDB In the SciDB implementation we adopt the above binning approach, but with an important difference. In SciDB it is not possible to dynamically split a region and distribute its replicas to an arbitrary number of adjacent bins, as we must apply identical operations to every cell in the array which stores the regions; thus, in order to apply a binning strategy, we must replicate all the regions an identical number of times. Such number is a function of the length `M` of the longest region in the `Reference` and `Experiment` datasets. In general, for given `M` and bin size `S`, each region will span to at most `R` bins, with: $\mathtt{R} = \lceil \mathtt{M}/\mathtt{S} \rceil + 1$. This is a limitation w.r.t. Spark, which can manage variable region replication; region replication in Spark occurs only when the region spans across two or more contiguous bins.

Dataset	Size (MByte)	Regions (Million)	Samples
REF	2.3	0.506	1
DS_1	38	1.012	20
DS_2	375	10.120	200
DS_3	3832	101.2	2000
DS_4	38232	1012	20000

Table 1. Features of the datasets used in the filtering operation.

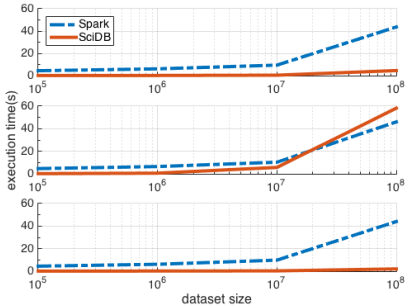
5 Benchmark

We performed our experiments on the Amazon Web Services (AWS) cloud, using a configuration with r3.4xlarge machine, 16 cores, 122 GB of RAM and 320GB of SDD. For the experiments reported in this Section, we use synthetic data, so that we can trace performance scaling with controlled, growing data sizes (In Section 6 we also show experiments over real genomic datasets); synthetic datasets are similar to Encode peak datasets [10].

- The schema includes just a Score attribute. Chromosomes are 22, and each chromosome has 1 million bases.
- Regions in each chromosome are 2300, randomly distributed over the chromosome space; length is randomly distributed between 20 and 500 bases.

We then generate 4 datasets with an increasing number of samples (up to 20K) and regions (up to 1 billion); see Table 1.

5.1 Regions Filtering



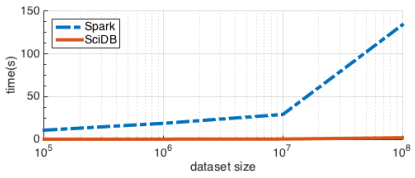
Test	DS_1	DS_2	DS_3	DS_4
Spark *Q1*	4.391	6.063	9.403	43.645
SciDB *Q1*	0.110	0.136	0.385	4.515
Spark *Q2*	4.640	6.447	10.299	46.049
SciDB *Q2*	0.161	0.581	5.673	58.137
Spark *Q3*	4.478	6.145	9.813	44.015
SciDB *Q3*	0.123	0.140	0.284	2.035

Fig. 5. Execution times (in seconds) for the filter operation.

We start comparing how Spark and SciDB execute the filtering operations discussed in Section 3.1. We consider three selection predicates: `Q1: chr='chr1'`, `Q2: score>0.9`, `Q3: (chr='chr1') and (score>0.9)`. Execution times of the operations in Spark and SciDB are reported in Fig. 5. We note that execution times for SciDB on `Q1` and `Q3` are much smaller than on `Q2`; in the former cases SciDB exploits the `between` operator and outperforms Spark. In `Q2`, instead, SciDB must read each single cell in order to apply the filtering operation, and in such case the execution time is similar to Spark, and it actually becomes worse with increasing data sizes.

5.2 Region Aggregation

Execution times of region aggregation `Q4: aggregate(count)` in Spark and SciDB are reported in Fig. 6. In this case we observe a huge difference between the two platforms performance: SciDB exploits the possibility to run in parallel the aggregation function in each chunk, and thus SciDB outperforms Spark.

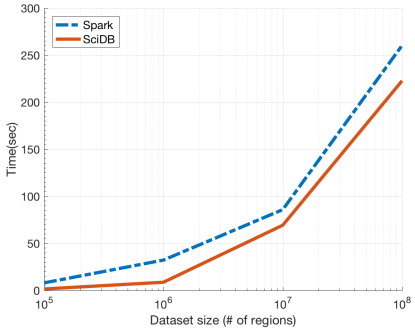


Test	DS_1	DS_2	DS_3	DS_4
Spark *Q4*	10.667	18.730	29.094	133.938
SciDB *Q4*	0.155	0.169	0.307	1.747

Fig. 6. Execution times (in seconds) for the aggregation operation.

5.3 Region Histogram

Execution times of region histogram `Q5` in Spark and SciDB are reported in Fig. 7. In spite of rather different algorithms used in SciDB and Spark for computing the histogram, their overall performance is similar, especially considering the dataset DS_4 and the scale-up. However, for the small datasets DS_1 and DS_2, SciDB has better performance (respectively by factors 6 and 4).



Test	DS_1	DS_2	DS_3	DS_4
Spark *Q5*	8.005	32.234	85.841	260.332
SciDB *Q5*	1.520	8.667	69.275	222.767

Fig. 7. Execution times (in seconds) for the histogram operation.

5.4 Region Mapping

Execution times of region mapping `Q6` in Spark and SciDB are reported in Fig. 8. Region mapping is an operation of **quadratic complexity**, similar to the join;

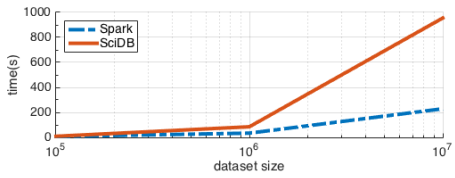
hence, execution times are much higher (expressed in minutes). In this case, we note that Spark outperforms SciDB, whose performance rises to about 1.5 hr when comparing .5 million regions of the reference with 101 million regions of 2000 experiments (note that this is a *big data operation*, as it potentially requires 50 trillion comparisons).

6 Conclusions

Although a large number of benchmarks exist for comparing general purpose cloud-based engines such as Spark and Flink, including academic articles ([5] and [15]) and several posts[2], we are not aware of benchmarks comparing these engines with array-based scientific databases, such as SciDB. Our paper shows that this benchmark has no clear winner; as expected, SciDB performs better when it benefits from the array-based database organization (hence, on region filtering and aggregation), while Spark performs better on massive region mapping operations (similar to joins). The histogram operation, which does not fall in either categories, has very close performance in SciDB and Spark.

The best performances of Spark in massive operations (map) hints to preferring it over SciDB in the management of applications with billions of regions. However, consider that our design matches regions to arrays using a general purpose data design, that serves general data integration requirements; we expect that specific array-based data designs could perform very well in SciDB; among them, cases of variant analysis, gene expression mining and high-throughput screening are described in [1].

Our GMQL architecture, which includes three GMQL implementations to SciDB, Spark and Flink, appears even more strongly motivated after this benchmark; by supporting various implementation engines we will be able to match application requirements to the best target system and to closely follow the evolution of cloud-based platforms during the ERC project timeframe.



Test	DS_1	DS_2	DS_3
Spark *Q6*	0.12	0.57	3.82
SciDB *Q6*	0.28	3.29	95.33

Fig. 8. Execution times (in minutes) for the mapping operation.

[2] See: http://sparkbigdata.com/102-spark-blog-slim-baltagi/14-results-of-a-benchmark-between-apache-flink-and-apache-spark.

Acknowledgment

The authors would like to thank the SciDB support team for help during Simone Cattani's thesis and for comments at his seminar, given at SciDB on July 19, 2016. This work is supported by the ERC Advanced Grant *GeCo (Data-Driven Genomic Computing)*.

References

1. Anonymous paper, Accelerating bioinformatics research with new software for big data to knowledge (BD2K), Paradigm4, 2015.
2. Anonymous paper, SciDB MAC Storage Explained, Paradigm4, 2015.
3. Apache Flink. http://flink.apache.org/ .
4. Apache Spark. http://spark.apache.org/ .
5. M. Bertoni, S. Ceri, A. Kaitoua, P. Pinoli Evaluating cloud frameworks on genomic applications. In *IEEE-Big Data Conference*, 193-202, 2015.
6. P. G. Brown, Overview of SciDB: large scale array storage, processing and analysis. In *Proc. ACM-SIGMOD*, 963-968, 2010.
7. S. Cattani, Genomic Computing with SciDB, a Data Management System for Scientific Computations, *Master Thesis*, Politecnico di Milano, July 2016.
8. B. Chawda et al. Processing Interval Joins On Map-Reduce. In *Proc. EDBT*, 463-474, 2014.
9. S. Edelkamp, D. Sulewski, C. Yucel, Perfect hashing for state sparce exploration on the gpu. In *Proc. ICAPS*, 57-64, 2010.
10. ENCODE Project Consortium. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57-74, 2012.
11. Hadoop 2. http://hadoop.apache.org/docs/stable/
12. A. Kaitoua, S. Ceri, M. Bertoni, P. Pinoli Framework for Supporting Genomic Operations *IEEE-TC*, 2016, DOI 10.1109/TC.2016.2603980.
13. M. Masseroli, et al. GenoMetric Query Language: A novel approach to large-scale genomic data management. *Bioinformatics*, 2015, doi: 10.1093/bioinformatics/btv048.
14. M. Masseroli, A. Kaitoua, P. Pinoli, S. Ceri. Modeling and interoperability of heterogeneous genomic big data for integrative processing and querying. *Methods*, 2016. DOI: 10.1016/j.ymeth.2016.09.002.
15. N. Spangelberg et al. Evaluating New Approaches for Big Data Analytics Frameworks, *BIS 2015*, LNBIP 18, Springer-Verlag, June 25, 2015.
16. M. Stonebraker et al. The architecture of SciDB. in *Proc. Scientific and Statistical Database Management*, 1-16, Springer-Verlag, 2011.
17. M. Stonebraker et al. SciDB: A database management syatem for applications with complex analytics. in *Computing in Science & Engineering,* 15(3), 54-62, 2013.
18. R. Xin et al. Shark: SQL and Rich Analytics at Scale. In *Proc. ACM-SIGMOD*, June 2013.
19. M. S. Weiwiorka et al. SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, 30(18):2652-2653, 2014.
20. M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, 15-28, 2012.
21. M. Zaharia et al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proc. SOSP*, November 2013.