


**AUTHOR QUERY FORM**

	<b>Journal:</b> Computer Physics Communications  <b>Article Number:</b> 5258	<b>Please e-mail or fax your responses and any corrections to:</b>  <b>E-mail:</b> <a href="mailto:corrections.esch@elsevier.river-valley.com">corrections.esch@elsevier.river-valley.com</a>  <b>Fax:</b> +44 1392 285879
---	---	--

Dear Author,

Please check your proof carefully and mark all corrections at the appropriate place in the proof (e.g., by using on-screen annotation in the PDF file) or compile them in a separate list. Note: if you opt to annotate the file with software other than Adobe Reader then please also highlight the appropriate place in the PDF file. To ensure fast publication of your paper please return your corrections within 48 hours.

For correction or revision of any artwork, please consult <http://www.elsevier.com/artworkinstructions>.

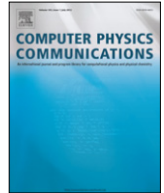
<b>Location in article</b>	<b>Query / Remark <a href="#">click on the Q link to go</a> Please insert your reply or correction at the corresponding line in the proof</b>
<a href="#">Q1</a>	Please confirm that given names and surnames have been identified correctly.
	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> <p style="color: red; margin: 0;">Please check this box or indicate your approval if you have no corrections to make to the PDF file</p> <input style="float: right; margin-left: 20px;" type="checkbox"/> </div>

Thank you for your assistance.



Contents lists available at ScienceDirect

## Computer Physics Communications

journal homepage: [www.elsevier.com/locate/cpc](http://www.elsevier.com/locate/cpc)

## A framework for building hypercubes using MapReduce

α1 D. Tapiador<sup>a,\*</sup>, W. O'Mullane<sup>a</sup>, A.G.A. Brown<sup>b</sup>, X. Luri<sup>c</sup>, E. Huedo<sup>d</sup>, P. Osuna<sup>a</sup>

<sup>a</sup> Science Operations Department, European Space Astronomy Centre, European Space Agency, Madrid, Spain

<sup>b</sup> Sterrewacht Leiden, Leiden University, P.O. Box 9513, 2300 RA Leiden, The Netherlands

<sup>c</sup> Departament d'Astronomia i Meteorologia ICCUB-IEEC, Martí i Franques 1, Barcelona, Spain

<sup>d</sup> Departamento de Arquitectura de Computadores y Automática, Facultad de Informática, Universidad Complutense de Madrid, Spain

## ARTICLE INFO

## Article history:

Received 26 July 2013

Received in revised form

17 November 2013

Accepted 4 February 2014

Available online xxxx

## Keywords:

Hypercube

Histogram

Data mining

MapReduce

Hadoop

Framework

Column-oriented

Gaia mission

## ABSTRACT

The European Space Agency's Gaia mission will create the largest and most precise three dimensional chart of our galaxy (the Milky Way), by providing unprecedented position, parallax, proper motion, and radial velocity measurements for about one billion stars. The resulting catalog will be made available to the scientific community and will be analyzed in many different ways, including the production of a variety of statistics. The latter will often entail the generation of multidimensional histograms and hypercubes as part of the precomputed statistics for each data release, or for scientific analysis involving either the final data products or the raw data coming from the satellite instruments.

In this paper we present and analyze a generic framework that allows the hypercube generation to be easily done within a MapReduce infrastructure, providing all the advantages of the new *Big Data* analysis paradigm but without dealing with any specific interface to the lower level distributed system implementation (Hadoop). Furthermore, we show how executing the framework for different data storage model configurations (i.e. row or column oriented) and compression techniques can considerably improve the response time of this type of workload for the currently available simulated data of the mission.

In addition, we put forward the advantages and shortcomings of the deployment of the framework on a public cloud provider, benchmark against other popular solutions available (that are not always the best for such ad-hoc applications), and describe some user experiences with the framework, which was employed for a number of dedicated astronomical data analysis techniques workshops.

© 2014 Published by Elsevier B.V.

## 1. Introduction

Computer processing capabilities have been growing at a fast pace following Moore's law, i.e. roughly doubling every two years during the last decades. Furthermore, the amount of data managed has been also growing at the same time as disk storage becomes cheaper. Companies like Google, Facebook, Twitter, LinkedIn, etc. nowadays deal with larger and larger data sets which need to be queried on-line by users and also have to answer business related questions for the decision making process. As instrumentation and sensors are basically made of the same technology as computing hardware, this has happened as well in science as we

can discern in projects like the human genome, meteorology information and also in astronomical missions and telescopes like Gaia [1], Euclid [2], the Large Synoptic Survey Telescope – LSST [3] or the Square Kilometer Array – SKA [4], which will produce data sets ranging from a petabyte for the entire mission in the case of Gaia to 10 petabytes of reduced data per day in the SKA.

Furthermore, raw data (re-)analysis is becoming an asset for scientific research as it opens up new possibilities to scientists that may lead to more accurate results, enlarging the scientific return of every mission. In order to cope with the large amount of data, the approach to take has to be different from the traditional one in which the data is requested and afterwards analyzed (even remotely). One option is to move to Cloud environments where one can upload the data analysis work flows so that they run in a low-latency environment and can access every single bit of information.

Quite a lot of research has been going on to address these challenges and new computing paradigms have lately appeared such as NoSQL databases, that relax transaction constraints, or other Massively Parallel Processing (MPP) techniques such as

\* Corresponding author. Tel.: +34 686028931.

E-mail addresses: [dtapiador@gmail.com](mailto:dtapiador@gmail.com) (D. Tapiador), [womullan@sciops.esa.int](mailto:womullan@sciops.esa.int) (W. O'Mullane), [brown@strw.leidenuniv.nl](mailto:brown@strw.leidenuniv.nl) (A.G.A. Brown), [xluri@am.ub.es](mailto:xluri@am.ub.es) (X. Luri), [ehuedo@fdi.ucm.es](mailto:ehuedo@fdi.ucm.es) (E. Huedo), [Pedro.Osuna@sciops.esa.int](mailto:Pedro.Osuna@sciops.esa.int) (P. Osuna).

<http://dx.doi.org/10.1016/j.cpc.2014.02.010>  
0010-4655/© 2014 Published by Elsevier B.V.

MapReduce [5]. This new architecture emphasizes the scalability and availability of the system over the structure of the information and the savings in storage hardware this may produce. In this way the scale-up of problems is kept reasonably close to the theoretical linear case, allowing us to tackle more complex problems by investing more money in hardware instead of making new software developments which are always far more expensive. An interesting feature of this new type of data management system (MapReduce) is that it does not impose a declarative language (i.e. SQL), but it allows users to plug in their algorithms no matter the programming language they are written in and let them run and visit every single record of the data set (always brute force in MapReduce, although this may be worked around if needed by grouping the input data in different paths using certain constraints). This may also be accomplished to some extent in traditional SQL databases through User Defined Functions (UDFs) although code porting is always an issue as it depends a lot on the peculiarities of the database and debugging is not straightforward [6]. However, scientists and many application developers are more experienced at, or may feel more comfortable with, embedding their algorithms in a piece of software (i.e. a framework) that sits on top of the distributed system, while not caring much about what is going on behind the scenes or about the details of the underlying system.

Furthermore, some of the more widely used tools in data mining and statistics are multidimensional hypercubes and histograms, as they can provide summaries of different and complex phenomena (at a coarser or finer granularity) through a graphical representation of the data being analyzed, no matter how large the data set is. These tools are useful for a wide range of disciplines, in particular in science and astronomy, as they allow the study of certain features and their variations depending on other factors, as well as for data classification aggregations, pivot tables confronting two dimensions, etc. They also help scientists validate the generated data sets and check whether they fit within the expected values of the model or the other way around (also applicable to simulations).

As multidimensional histograms can be considered a very simple hypercube which normally contains one, two or three dimensions (often for visualization purposes) and whose measure is the count of objects given certain concrete values (or ranges) of its dimensions, we will generally refer to hypercubes through the paper and will only mention histograms when the above conditions apply (hypercubes with one to three dimensions whose only measure is the object count).

Previous work applying MapReduce to scientific data includes [7], where a High Energy Physics data analysis framework is embedded into the MapReduce system by means of wrappers (in the Map and Reduce phases) and external storage. The wrappers ensure that the analytical algorithms (implemented in a different programming language) can natively read the data in the framework specific format by copying it to the local file system or to other content distribution infrastructures outside the MapReduce platform. Furthermore, [8] and [9] examine some of the current public Cloud computing infrastructures for MapReduce and study the effects and limitations of parallel applications porting to the Cloud respectively, both from a scientific data analysis perspective. In addition, [10] shows that novel storage techniques being currently used in commercial parallel DBMS (i.e. column-oriented) can also be applied to MapReduce work flows, producing significant improvements in the response time of the data processing as well as in the compression ratios achieved for randomly-generated data sets. Last but not least, several general-purpose layers on top of Hadoop (i.e. Fig [11] and Hive [12]) have lately appeared, aiming at processing and querying large data sets without dealing directly with the lower level API of Hadoop, but using a declarative language that gets translated into MapReduce jobs.

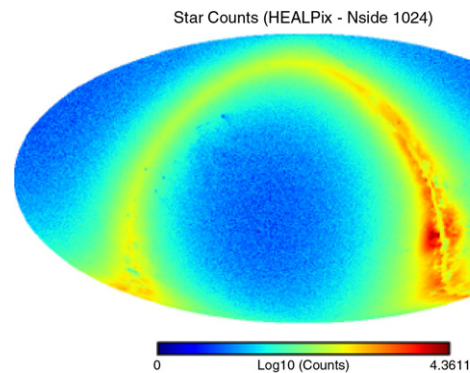


Fig. 1. Star density map using HEALPix.

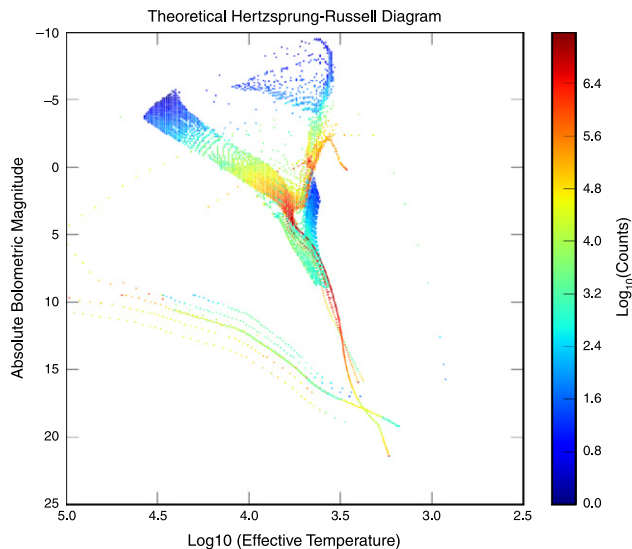
This paper is structured as follows. In Section 2, we present the simulated data set that will be used through the paper and some simple but useful examples that can be built with the framework. Section 3 describes the framework internals. In Section 4, we show the experiments carried out, analyzing the deployment in a public Cloud provider, examining the data storage models (including the column-oriented approach) and compression techniques, and benchmarking against two other well known approaches. Section 5 puts forward some user experiences in some astronomical data analysis techniques workshops. Finally, Sections 6 and 7 refer to the conclusions and future work respectively.

## 2. Data analysis in the Gaia mission

In the case of the Gaia mission, many histograms will be produced for each data release in order to summarize and document the catalogs produced. Furthermore, a lot of density maps will have to be computed, e.g. for visualization purposes, as otherwise it would be impossible to plot such a large amount of objects. All these histograms and plots (see [13] for examples), the so-called *precomputed statistics*, will have to be (re)generated in the shortest period of time and this will imply a load peak in the data center. Therefore, the solution adopted should be able to scale to the Cloud just in case it is needed due to e.g. the absence of a local infrastructure that can execute these work flows (as it would mean a high fixed cost for hardware which is underutilized most of the time).

Figs. 1 and 2 show two simple examples of histograms that have been created with the framework and which we will use throughout the paper for presenting the different results obtained. The GUMS10 data set [13], from which histograms have been created, is a simulated catalog of stars that resembles the one that will be produced by the Gaia mission. It contains a bit more than two billion objects with a size of 343 GB in its original delivery form (binary and compressed with Deflate). Since there is no Gaia data yet, all Gaia data processing software is verified against simulated observations of this universe model [13].

The histogram shown in Fig. 1 is a star density map of the sky. It has been built by using a sphere tessellation (pixelization) framework named HEALPix [14], which among other things provides a set of routines for subdividing a spherical surface into equal area pixels, and for obtaining the pixel number corresponding to a given pair of angular coordinates. HEALPix is widely known not only in astronomy but also in the field of earth observation. HEALPix also allows indexing of geometrical data on the sphere for speeding up queries and retrievals in relational databases. The resolution of the pixels is driven by a parameter called  $N_{\text{side}}$ , which must be a power of two. The higher this parameter is, the more pixel subdivisions the sphere will have. For  $N_{\text{side}} = 1024$  (used in Fig. 1 and in the rest of tests below) there are 12 582 912 pixels.



**Fig. 2.** Theoretical Hertzsprung–Russell diagram. The horizontal axis shows the temperature of the stars on a logarithmic scale and the vertical axis shows a measure of the luminosity (intrinsic brightness) of the stars (also logarithmic, brighter stars are at more negative values).

The example in Fig. 2 is a theoretical Hertzsprung–Russell diagram (two dimensional) which shows the effective temperature of stars vs. their luminosity. This is a widely used diagram in astronomy, which contains information about the age (or mixture of ages) of the plotted set of stars as well as about the physical characteristics and evolutionary status of the individual stars.

For the scenario just sketched the MapReduce approach is the most reasonable one. This is not only due to the fact that it scales up very well (also in the Cloud) or that there are open-source solutions already available like Hadoop<sup>1</sup> (which we will use), but also because the generation of a hypercube fits perfectly into the MapReduce paradigm. This is not necessarily true for other parallel computing paradigms such as Grid computing, where the processing is efficiently distributed but the results are cumbersome to aggregate afterwards, or MPI,<sup>2</sup> where the developer has to take care of the intrinsic problems of a distributed system. In the case of a parallel DBMS, these two simple examples could be easily created either by using UDFs with external HEALPix libraries (something already done for Microsoft SQL Server at the Sloan Digital Sky Survey<sup>3</sup>) or directly with a SQL query for the theoretical Hertzsprung–Russell diagram. However, more complex histograms or hypercubes would be much more difficult to generate. Furthermore, the scalability will be better in Hadoop as the data set grows due to the inherent model of MapReduce. Last but not least, the generation of several histograms and/or hypercubes each one with different constraints in terms of filtering or aggregation (e.g. several star density maps at different  $N_{\text{side}}$  granularities) will be more efficient using our framework on top of Hadoop (provided the amount of data is very large) as they will be computed in one single scan of the data.

### 3. Framework description

The framework (implemented in Java) has been conceived considering the following features:

- Thin layer on top of Hadoop that allows users or external tools to focus only on the definition of the hypercubes to compute.

- Hide all the complexity of this novel computing paradigm and the distributed system on which it runs. Therefore, it provides a way to deal with a cutting-edge distributed system (Hadoop) without any knowledge of *Big Data* internals.
- Possibility to process as many hypercubes as possible in one single scan of data, taking advantage of the brute-force approach used in Hadoop jobs, thus reducing the time for generating the precomputed statistics required for each data release.
- Leverage the capabilities offered by this new computing model so that the solution is scalable.
- *Java generics* have been used throughout the framework in order to ease its integration in any domain and permit a straightforward embedding of any already existing source code.

Hypercubes defined in the framework (see Fig. 3 for an example) may have as many dimensions (categories) as required. They may define intervals (for a continuous function) or be of a discrete type (for discrete functions), depending on the use case. Users may supply their own algorithms in order to specify how the value of each dimension will be computed (by implementing the corresponding *getField* method). This value might be of a custom user-defined type in case of need. The input object being processed at each time is obviously available for performing the relevant calculations.

It is important to highlight that the possibility of defining as many dimensions as needed is what allows us to easily build data mining hypercubes or concrete pivot tables, and do so on-the-fly (at runtime) without losing generality. This is a key feature for scientific data analysis because the data is always exploited in many different ways due to the diversity of research studies that can be done with them. Furthermore, the cubes generated may be further analyzed (i.e. slice, dice, drill-down and pivoting operations) within the framework by defining a new Hadoop input format that reads the output of the cube generation job and delivers it to the next analytical job. The results can also be exported to a database in order to perform the subsequent analysis in there.

We must also set the value that will be returned for each entry being analyzed. This will usually be a value of '1' when performing e.g. counts of objects falling into each combination of categories, but it might also be any other derived (user-implemented) quantity for which we want to know the maximum or minimum value, the average, the standard deviation, or any other linear statistical value. There is only one MapReduce phase for the jobs so more complex statistics cannot currently be calculated, at least not efficiently and in a scalable way (e.g. the median, quartiles and the like). We may however define a custom type such that the cells of the hypercube contain more information than just a determined measure. We can also define a filter which is used to decide which input objects will be analyzed and which ones will be discarded for each hypercube included in the job.

The current implementation offers a lot of helpers that can be plugged in many different places for many different purposes. For instance, if we just want to get a field out of the input object being processed for a certain dimension (or for the value returned), we can just use a helper reader that obtains that field at runtime through *Java reflection*, and avoid the generation of a new class whose only method would just return the field. The user just needs to specify the field name and make sure that the object provides the relevant accessor (*getter* method).

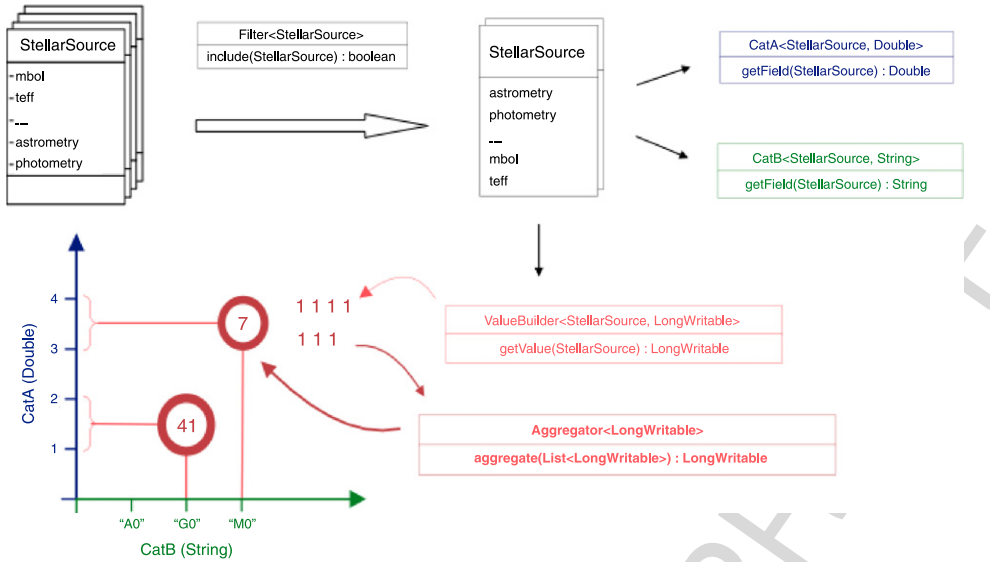
Last but not least, the manner in which the data is aggregated as well as whether the aggregator can be used as the Hadoop combiner for the job (recommended whenever possible [15]) can also be defined by the user (and could also be defined per hypercube with minor changes), although most of the time they will just set one of the currently available helpers (for computing counts, the minimum/maximum value, the average, the standard

<sup>1</sup> <http://hadoop.apache.org>.

<sup>2</sup> [www.mpi-forum.org](http://www.mpi-forum.org).

<sup>3</sup> <http://www.sdss.org/>.





**Fig. 3.** Data workflow through the framework and main interfaces to implement for each hypercube. The sample in the figure shows a hypercube with two dimensions (discrete for the x axis and continuous with intervals for the y axis) that counts the number of elements falling into each combination of the categories.

deviation, etc.). For more complex hypercubes (i.e. several measures aggregated differently on each cell of the hypercube), we could create an aggregator along with a custom type for cell values so that the different measures are aggregated differently (the count for some of them, the average for others, etc.).

Before running the job it is required to set some configuration properties for the definition of the input files and the corresponding input format to use, the path where to leave the results, the class that will define the hypercubes to create (Listing 1 shows the skeleton of this class for the example shown in Fig. 3), and some other parameters like the type of the output value returned for them (mandatory for any Hadoop job). This last constraint forces all entities computed in the same job to have the same return type (the reducer output value, e.g. the count, the maximum value, etc.), although this can be easily worked around if needed by setting more generic types (a *Double* for holding both integers and floating point numbers, a *String* for numbers and text, etc.), or as stated above, developing a custom type (implementing the *Hadoop Writable* interface) that holds them in different fields, along with the corresponding aggregator.

**Listing 1:** Custom class defining the hypercube(s) to compute.

```

public class MyHypercubeBuilder
extends BuilderHelper<StellarSource, LongWritable> {

    @Override
    public List<Hypercube<StellarSource, LongWritable>>
    getHypercubes() {
        // Create list holding the hypercubes
        // Create CatA instance (with ranges)
        // Create CatB instance
        // Create Filter instance
        // Create ValueBuilder (use Helper)
        // Create Aggregator (use Helper)
        // Create Hypercube instance
        // Add to hypercubes list
        // Return hypercubes list
    }
}
    
```

The output files of the job have two columns, the first one for identifying the hypercube name as well as the combination of the concrete values for its dimensions (split by a separator defined by the user), and the second one holding the actual value of that combination of categories (see Listing 2 for a sample of

the output for the Theoretical Hertzsprung–Russell diagram shown in Fig. 2). The types used for discrete categories must provide a method to return a string which unequivocally identifies each of the possible values of the dimension. For categories with intervals, the string in the output file will contain information on the interval itself with square brackets and parentheses as appropriate (closed and open ends respectively), but again they must ensure that the types of the interval ends (bin ends) supply a unequivocal string representation. This unequivocal representation might be the primary key of the dimension's concrete value (for more advanced hypercubes) so that it can later on be joined with the rest of the information of that dimension as it usually happens in data mining *star* schemas.

**Listing 2:** Sample of the output for the Theoretical Hertzsprung–Russell diagram shown in Fig. 2.

```

[... ]
TheoreticalHR/[3.58, 3.5825]/[16.7,16.725]/ 998
TheoreticalHR/[3.58, 3.5825]/[8.0,8.025]/ 883
TheoreticalHR/[3.5875, 3.59]/[-4.875,-4.85]/ 328
TheoreticalHR/[3.5875, 3.59]/[-5.4,-5.375]/ 391
TheoreticalHR/[3.6075, 3.61]/[-0.9,-0.875]/ 87031
TheoreticalHR/[3.6075, 3.61]/[-3.6,-3.575]/ 2780
TheoreticalHR/[3.6075, 3.61]/[-3.925,-3.9]/ 12384
[... ]
    
```

One straightforward but important optimization that has been implemented is the usage of sorted lists for dimensions that define continuous, non-overlapping intervals. This way the number of comparisons to do per input object is considerably lowered, reducing by a factor of 20 the time taken for the execution. Therefore, although non-continuous (and non-ordered) interval categories are allowed in the framework, it is strongly recommended to define continuous (and non-overlapping) ranges even though some of them may be later on discarded.

**4. Experiments**

**4.1. Cloud deployment**

Recently there has been a blossoming of commercial Cloud computing service providers, for example Amazon Web Services (AWS), Google Compute Engine, Rackspace Cloud, Microsoft Azure

and several other companies or products sometimes focused on different needs (Dropbox, Google Drive, etc.). AWS has become one of the main actors in this Cloud market, offering a wide range of services such as the ones that have been used for this work: Amazon Elastic MapReduce (Amazon EMR<sup>4</sup>), Amazon Simple Storage Service (Amazon S3<sup>5</sup>) and Amazon Elastic Compute Cloud (Amazon EC2<sup>6</sup>). The way these three services are used is as follows: EC2 provides the computers that will run the work flows, S3 is the data store where to take the data and leave the results, and EMR is the Hadoop ad-hoc deployment (and configuration) provided for MapReduce jobs. Amazon charges for each service, and not only for the computing resources but also for the storage on S3 and the data transfers in and out of their infrastructure. They also provide different instances (on-demand, reserved and spot instances) which obviously have different prices at different levels of availability and service. The EMR Hadoop configuration is based on the current operational version of Hadoop with some bug fixes included. Furthermore, the overall experience with Amazon EMR is very good and it has been quite easy to start submitting jobs to it through command line tools openly available. Debugging is also quite easy to do as *ssh* access is provided for the whole cluster of nodes.

The deployment used for testing and benchmarking consists of eight worker nodes each one having the Hadoop data and task tracker nodes running on them. There is also one master node which runs the name node and the job tracker. The AWS instance chosen is *m1.xlarge*, which has the following features:

- 4 virtual cores (64-bit platform).
- 15 GB of memory.
- High I/O performance profile (1 Gbps).
- 1690 GB of local Direct Attached Storage (DAS), which sums up to a bit more than 13 TB of raw storage which may be cut down by half or more depending on the Hadoop Distributed File System (HDFS) [16] replication factor chosen.

With this layout, EMR Hadoop deployment launches a maximum number of 8 mappers and 3 reducers per worker node (64 and 24 for the entire cluster respectively). It is important to remark that the time taken for starting the tasks of a job in Hadoop is not negligible (more than one minute for the tests carried out) and certainly affects the performance of short jobs [6], as it imposes a minimum amount of time that a job will always last (sequential workload) no matter the amount of data to process. This is one of the reasons why Hadoop is mostly advised for very big workloads (*Big Data*), where this effect can just be disregarded.

#### 4.2. Data storage model considerations

Scientific raw data sets are not normally delivered in a uniformly sized set of files as the parameters chosen for placing the data produce a lot of skew due to features inherent to the data collection process (some areas of the sky are more densely populated, a determined event does not occur at regular intervals, etc.). This is also true for the data set being analyzed in this paper (GUMS10) as it comprises a set of files each one holding the sources of the corresponding equal-area sky region (see Fig. 1 for its histogram drawn in a sky projection). This may be a problem for binary (and often compressed) files when stored in HDFS as the records cannot be split into blocks (there is no delimiter as in the text format). The data formats studied in this paper are of

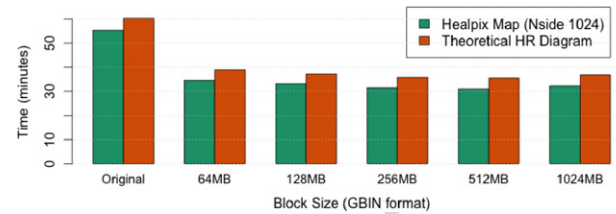


Fig. 4. Performance for different HDFS block and file sizes (files in GBIN format are binary and compressed with Deflate).

this type (binary and compressed with no delimiters), defined by the Gaia mission SOC (Science Operations Centre). Thus we have to read each of them sequentially in one Hadoop mapper and their size must be roughly the same and equal to the defined HDFS block size to maximize performance through data locality. Fig. 4 shows the performance obtained when computing the different histograms shown in Figs. 1 and 2: a HEALPix density map and a theoretical Hertzsprung–Russell diagram. As we can see, once we group the data into equally sized files and set the HDFS block to that size, the time consumed for generating them is approximately 2/3 the time taken when the original highly-skewed delivery is used. The standard format chosen for data deliveries within the Gaia mission is called ‘GBIN’, which contains *Java*-serialized binary objects compressed with Deflate (ZLIB).

In Fig. 4 we can also see that there is a block size which performs slightly better than the others (512 MB) which is a consequence of the concrete configuration used for the testbed, as more files mean more tasks (Hadoop mappers) being started which is known to be slow in Hadoop as already remarked above. Furthermore, less but bigger files may produce a slowdown in the data shuffling period (each Hadoop mapper outputs more data which then has to be combined and shuffled). Therefore, we will use the best configuration (data files and block size of 512 MB) for the comparison with other data storage techniques and for benchmarking.

To analyze the effects of the different compression techniques available and study how they perform for an astronomical data set, a generic data input format has been developed. This way, different compression algorithms and techniques may be plugged into Hadoop, again without dealing with any Hadoop internals (input formats and record readers). This is more or less the same idea as the generic input format interface provided by Hadoop but more focused on binary (non-splittable) and compressed data. The data reader to use for the job must be configured through a property and its implementation must provide operations for setting up and closing the input stream to use for reading, and for iterating through the data objects. The readers developed so far store *Java* serialized binary objects with different compression techniques which are indicated below as: GBIN for Deflate (ZLIB), Snappy for Google Snappy<sup>7</sup> compression and Plain for no compression.

Fig. 5 shows the results obtained when these compression techniques are used with the GUMS10 data set for creating the same histograms as before. It is important to remark that no attention has been paid to other popular serialization formats currently available like Thrift,<sup>8</sup> Avro<sup>9</sup> etc., as the time to (de)serialize is always negligible compared to the (de)compression one. Furthermore, as stated above, the data is always stored in binary format as the textual counterpart would lead to much worse results (a proof of this is the battery of tests presented in [6]).

<sup>4</sup> <http://aws.amazon.com/elasticmapreduce/>.

<sup>5</sup> <http://aws.amazon.com/s3/>.

<sup>6</sup> <http://aws.amazon.com/ec2/>.

<sup>7</sup> <http://code.google.com/p/snappy/>.

<sup>8</sup> <http://thrift.apache.org/>.

<sup>9</sup> <http://avro.apache.org/>.

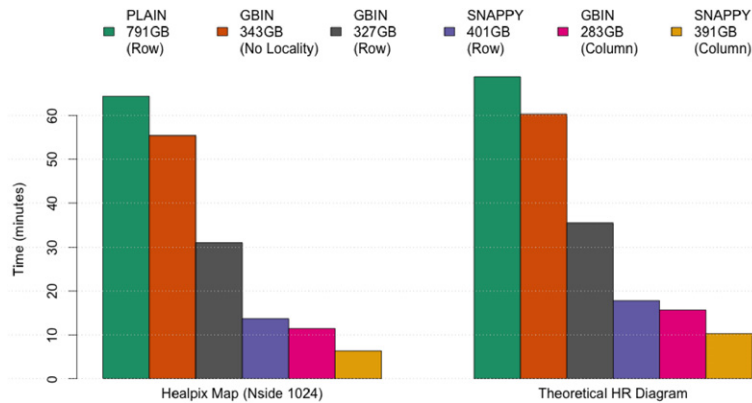


Fig. 5. Data storage model approaches performance comparison.

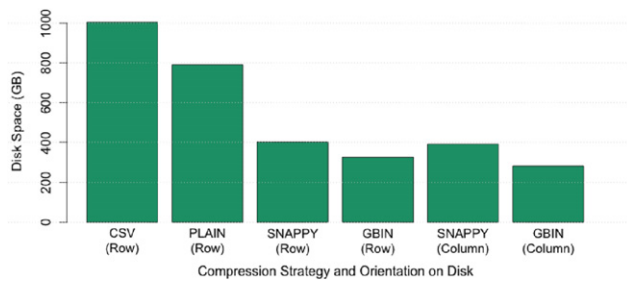


Fig. 6. Data set size for different compression and format approaches.

Google Snappy codec gives a much better result as the decompression is faster than Deflate (GBIN). It takes half of the time to process the histograms (50%) and the extra size occupied on disk is only around 23% (see Fig. 6). This confirms the suitability of this codec for data to be stored in HDFS and later on analyzed by Hadoop MapReduce work flows.

Fig. 5 also shows the performance obtained with another data storage model developed ad-hoc using a new Hadoop input format, column-oriented (see the rightmost two columns in both histograms), which resembles the one presented in [10], although it integrates better in the client code as the objects returned are of the relevant type (the information that we want to populate from disk still has to be statically specified though as in [10]). Furthermore, we obviously expect that the improvements made by the column input format are more significant as the data set grows larger (both in number of rows and columns), although we can also state that performance will decrease when most of the data set columns are required in a job, due to overheads incurred in the column-oriented store mechanism (several readers used at the same time, etc.). These issues have to be carefully considered for each particular use case before any of the formats is chosen.

The Hadoop operational version at the time of writing does not yet provide a way to modify the block data placement policy when importing data into HDFS (newer alpha/beta versions do support this to some extent although these could not be used in Amazon EMR). The current algorithm for deciding what data node is used (whenever an input stream is opened for a certain file), chooses the local node if there is a replica in there, then another random node in the same rack (containing a replica) if it exists, and if there is no one serving that block in the same rack it randomly chooses another data node in an external rack (containing a replica of course). Considering this algorithm, if we set the replication policy to the number of cluster worker nodes, we ensure that there will always be a local replica of everything on every node and thus we can simulate that the column files corresponding to the same data objects have been placed in the same data node (and replicas) for data locality of input data readers. This is not to be used in an

operational deployment of course, but it has served its purpose in our study. Meanwhile, new techniques that overcome these issues are being put in place (i.e. embed data for all columns in the same file, and split by row ranges).

These new techniques, whose main implementations are Parquet,<sup>10</sup> ORC<sup>11</sup> (Optimized Row Columnar) and Trevni<sup>12</sup> (already discontinued in favor of Parquet) should always be chosen instead of ad-hoc developments like the one presented here, as even though some of them may not yet be ready for operations, they are rapidly evolving and will become very soon the default input and output formats for many use cases, overall for those found in science and engineering. The main features of this new technique are enumerated below:

- Data are stored contiguously on disk by columns rather than rows. Then, each block in HDFS contains a range of rows of the data set and there is some metadata that can be used to seek to the start or the end of any column data, so if we are reading just two columns, we do not have to scan the whole block, but just the two columns data. This way, it is not necessary to create many different files which might incur in extra overhead for the HDFS name node.
- Compression ratio for each column data will be higher than the row oriented counterpart due to the fact that values for the same column are usually more similar, overall for scientific data sets involving time series, because new values representing certain phenomena are likely to be similar than those just measured. These implementations go beyond a simple compression for the column data by allowing different compression algorithms for different types of data, or even do so on the fly as we create the data set by trying several alternatives. For instance, for a column representing a measure (floating point number) of a determined sensor or instrument, it would be reasonable to use a delta compression algorithm where we store the differences between values which will probably require less bits for their representation. It is important to remark that I/O takes more time than the associated CPU time (de)compressing the same data. For other columns with enumerated values, the values themselves will be stored in the metadata section along with a shorter (minimum) set of bits which will be the ones being used in the column data values. This of course requires (de)serializing the whole block (range of rows) for building back the original values of each column, but this technique usually performs well and takes less time.

<sup>10</sup> <http://parquet.io/>.

<sup>11</sup> [http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds\\_Hive/orcfile.html](http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds_Hive/orcfile.html).

<sup>12</sup> <https://github.com/cutting/trevni>.



- Predicate push-down. Not only we can specify the set of columns that will be read for a determined workflow, but for those that need to be queried or filtered with some constraints, we can also push down the predicates so that the data not needed are not even deserialized in the worker nodes, or even not read from disk (only the metadata available is accessed).
- Complex nested data structures can be represented in the format (not just simple flattening of nested namespaces). The technique for implementing this feature is presented in more detail in [17].
- This data format representation is agnostic to the data processing, data model or even the programming language.
- Further improvements of the column-based approach include the ability to split files without scanning for markers, some kind of indexing for secondary sorting, etc.

Fig. 6 shows that the level of compression achieved by our naive implementation of the column-oriented approach (compared to the row-oriented counterpart) is not as good as it might be expected. This may be caused by the fact that an entire row may much resemble the next row (similar physical properties), so the whole row may be considered a column, but at a higher granularity, leading to a relatively good compression in the row-oriented storage model as well. Another more plausible explanation for this may be that we do not use deltas for adjacent data (in columns) as is usual [18], but the values themselves.

Contrasting the results in Figs. 5 and 6, we see that the column-oriented storage model (using Snappy codec) takes more disk space than the row-oriented one (with Deflate). This extra cost overhead (64 GB) amounts to \$8 per month in Amazon S3 storage (where the data are taken from at cluster initialization time), which is much less than the price incurred in a typical workload where many histograms and hypercubes have to be computed, as e.g. the cluster must be up 24 min more in the case of a HEALPix density map computation (which comes to a bit more than \$3 extra per job) or 25 min more for a single theoretical Hertzsprung–Russell diagram (again a bit more than \$3 extra per job). Therefore, for typical larger workloads of several (and more complex) histograms and/or hypercubes, we can expect larger and larger cost savings with the column-oriented approach as computation is much more expensive than the extra overhead in S3 storage, mainly due to the amount of jobs to be run as well as the non-negligible cost of the cluster nodes.

### 4.3. Benchmarking

Two powerful and well-known products have been chosen for the benchmark, Pig<sup>13</sup> 0.11.0 and Hive<sup>14</sup> 0.10.0. These open source frameworks, which also run on top of Hadoop, offer an abstraction of the MapReduce model, providing users with a general purpose, high-level language that could be used not only for the hypercubes described above, but also for other data processing workflows such as ETL (Extract, Transform and Load). However, they might require some further work to do in case we wanted to build more complex hypercubes (involving several dimensions and values, some of them computed with already existing custom code), or generating several hypercubes in one scan of the input data (something not neatly expressed in a SQL query).

The tests carried out use a row-oriented scheme with compressed (Snappy) binary data and have been run on the infrastructure described in Section 4.1. For the purpose of benchmarking, we will carefully analyze different scenarios:

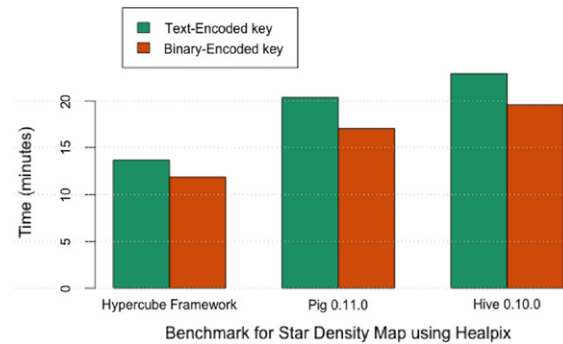


Fig. 7. Comparison among the framework presented and other popular data analysis tools currently available. All tests have been run using the Hadoop standard Merge-Sort algorithm for data aggregation.

- Simple one-dimensional hypercube with the key encoded as text (the default for the framework), and as binary (more efficient but less flexible).
- Two-dimensional hypercube with low key cardinality where the aggregation factor is high.
- Several hypercubes at the same time with different key cardinalities.
- Different aggregation algorithms (default Merge Sort and Hash-based).
- Scalability tests by increasing the data set size, but keeping the same cardinalities for the keys.

Fig. 7 shows the results obtained when generating the hypercube plotted in Fig. 1. Two approaches (with the key encoded as text and as binary) have been considered in order to prove that the proportions in execution time for the different solutions are kept, although the framework is supposed to always work with text in its current version. We can see that the framework performs considerably better (33% in the case of Pig and up to 40% for Hive) and we argue that this may be due to its simplicity in the design yet the efficient core logic built inside, which cannot be achieved by general purpose frameworks that are supposed to span a very wide domain of applications. Therefore, this generality has a high impact in cost when we focus on a particular use case like the one described here.

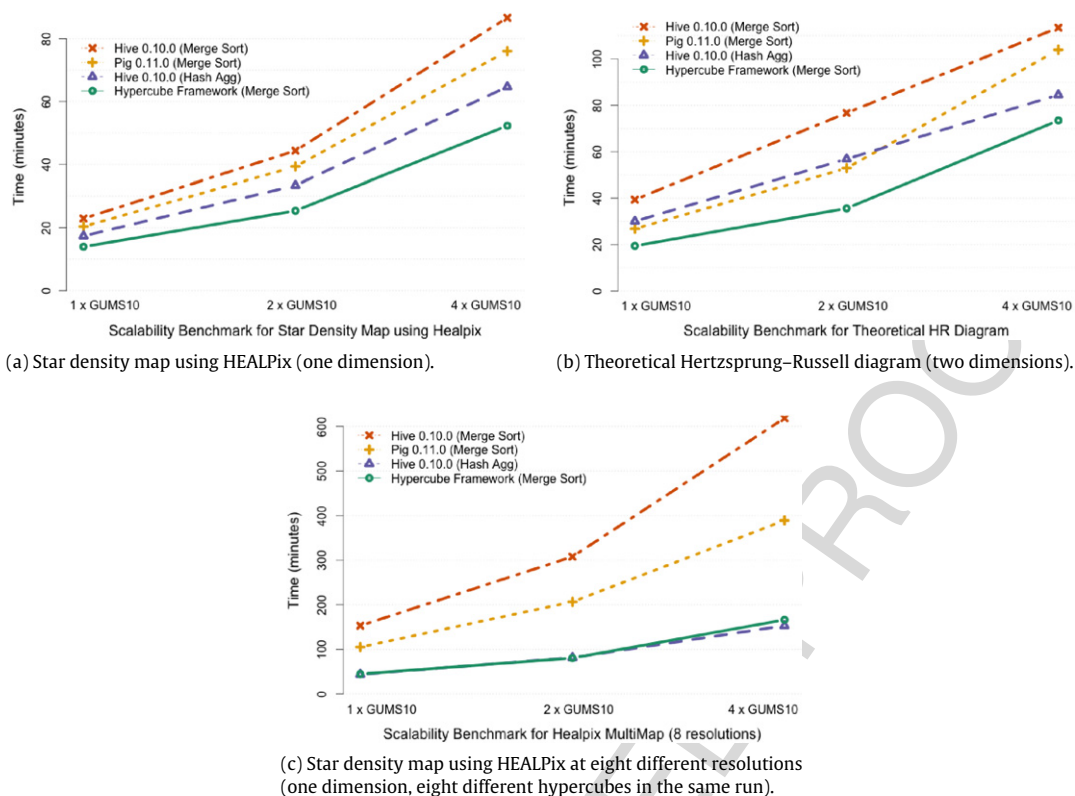
The results shown in Fig. 8 refer to the scalability of the alternatives studied for different computations and configurations. The same data set (GUMS10) is used to enlarge the input size, although it is important to notice that the output size will remain the same as the number of bins will not change as we increase the input data.

We can see that the framework performs significantly better in the use cases studied, which proves that for well-known, operational workloads, it is usually better to use a custom implementation (or an ad-hoc framework) rather than using general purpose tools which are more suited for exploration or situations where performance is not so important. However, we can be certain that these general purpose and higher level implementations are catching up fast enough if we look at the optimizations they are currently releasing, such as hash-based aggregation. This technique (known as *In-Mapper* combiner) tries to avoid data serialization and disk I/O by aggregating data in a hash table in memory. Then, the mappers that run in the same JVM do not emit data until they are all finished (as long as the aggregation ratio is high enough). Then, the steps for serializing data and the associated disk I/O before the combiner is executed are not needed anymore, thus improving performance dramatically. The logic built-in for accomplishing this new functionality is rather complex not only because Hadoop was not designed for this kind of processing in the first place, but also due to the dynamic nature of the implementations, which can switch on-the-fly between hash-based and merge-sort

<sup>13</sup> <http://pig.apache.org/>.

<sup>14</sup> <http://hive.apache.org/>.





**Fig. 8.** Scalability benchmark for (a) star density map, (b) theoretical HR diagram and (c) star density map at eight different resolutions. The approaches shown encompass different alternatives from the Hadoop ecosystem and the two main algorithms used for aggregating data, i.e. Merge Sort (the default for Hadoop) and Hash Aggregation (whose implementation is known in the Hadoop ecosystem as *In-Mapper combiner*). The results for Hash aggregation are only shown for Hive, which is the only one that showed some improvements in the tests run. The **data set** is enlarged one, two and four times with the same data ( $1 \times \text{GUMS10}$ ,  $2 \times \text{GUMS10}$  and  $4 \times \text{GUMS10}$  respectively) and therefore the cardinality of the key space for each hypercube being computed remains unchanged.

aggregations by spilling to disk what is inside the hash table once a certain configured aggregation threshold is not met by the workflow at run time. The implementation of this automatic switching is something that will make these higher level tools much more efficient, but it will also require much more expertise from users for tuning the best configuration for the workflows.

Furthermore, the current implementation for Hive shows a very good performance gain as shown in Fig. 8(c) but is not significantly better than the merge-sort counterpart when using Hadoop directly. This may be caused by the fact that there is not much I/O due to the small size of each pair of keys and values, compared to the savings produced for a better in-memory aggregation. Results for hash-based aggregation in Pig have been omitted due to its very poor performance in the release used, which proves that a more robust implementation must properly handle the memory consumed by the hash table, allowing to switch to Merge Sort dynamically whenever the cardinality goes beyond a predefined threshold. This dynamism in query execution may become an asset for Hadoop-based processing comparing to parallel DBMS, where the query planner picks one alternative at query parsing time and usually sticks to it till the end. In Hadoop, this is more dynamic and gives more flexibility and adaptability at run time.

One of the most common features of data pipelines is that they are often DAG (Directed Acyclic Graph) and not linear pipelines. However, SQL focuses on queries that produce a single **result set**. Thus, SQL handles trees such as joins naturally, but has no built in mechanism for splitting a data processing stream and applying different operators to each sub-stream. It is not uncommon to find use cases that need to read one data set in a pipeline and group it by multiple different grouping keys and store each as separate output. Since disk reads and writes (both scan time and intermediate results) usually dominate processing of large data

sets, reducing the number of times data must be written to and read from disk is crucial to good performance.

The recent inclusion of the *GROUPING SETS* clause in Hive has also contributed to the improvements shown in Fig. 8(c), comparing to those in Fig. 8(a) and (b), as it allows that the aggregation is made with different keys (the ones specified in the clause) yet only one scan of data is needed. This fits perfectly in the scenario posed in the test shown in Fig. 8(c), where we compute several hypercubes at the same time in the same **data set**. However, *GROUPING SETS* clause is complex since the keys specified have to be in separate columns. Therefore, when trying to compute results in the format of a single key plus its corresponding value, we will always get the key which the row refers to, plus the rest of keys with empty values (*null*).

We have found other usability issues in Hive, which we believe will be addressed soon, but which may currently lead to a worse user experience, such as the lack of aliases on columns. This is a minor problem, but most of the users and client applications are used to relying on them everywhere for reducing complexity or increasing flexibility, overall when applying custom UDF or other built-in operators. Furthermore, there is no way to pass parameters to the UDF when initializing, which has made the execution of the multi-hypercube workflow more difficult to run, as several different UDF had to be coded, even though they all share the same functionality and the only difference is the parameter that sets the resolution of the map.

Comparing Pig and Hive, results show that Pig performs significantly better than Hive when the (Hadoop) standard Merge Sort algorithm is chosen. However, the implementation of the hash-based aggregation in Hive seems more mature and gives a better performance than the Merge Sort alternative in Hive, for those cases where the aggregation factor is high enough (see the

tendency of Fig. 8(b) where the aggregation factor is increased as we enlarge the **data set** due to the same data being duplicated).

One of the most important conclusions to take away when looking at these results is that there is no solution that fits all problems. Therefore, special care has to be taken when choosing the product to use as well as when setting the algorithm and tuning its parameters. Results show that a bad decision may even double execution time for certain workloads. In this case, an ad-hoc solution fits better than more generic ones, even though already implemented hash-based algorithms in Hive and Pig may seem more appropriate upfront. In addition, there are other optimizations that could be easily made, such as sort avoidance, because it is normally not needed when processing aggregation workflows.

Another remark worth mentioning is that when we double the input size, the execution time is a bit less than the expected (double) one. This is due to the fact that Hadoop inherent overhead starting jobs is compensated by the larger workload, which proves that Hadoop is not well suited for small **data sets** as there is a non-negligible (and well-known) latency starting tasks in the worker nodes.

## 5. User experience

The hypercube generation framework is packaged as a JAR (Java Archive) file and has a few **dependencies** on other packages (mainly on those of Hadoop distribution). To make use of the framework, the user has to write some code that sets what hypercubes to compute (see Listing 1), as well as any extra code that will be executed by the framework, e.g. when computing the concrete categories or values for each hypercube and input record in the **data set** being processed. Furthermore, the user is expected to package all classes into a JAR, create a file containing (at least) the properties specified in Section 3 (input and output paths, etc.), and run that JAR on a Hadoop cluster following Hadoop documentation.

The framework has been tested by offering it to a variety of user groups. One of the authors (AB), without any background in computer science (but with experience in *Java* programming), tried out the framework as it was being developed. He had no significant problems in understanding how to write pieces of *Java* code needed to generate hypercubes for specific categories or intervals and was able to quickly write a small set of classes for supporting the production of hypercubes for quantities (e.g. energy and angular momentum of stars) that involve significant manipulation of the basic **catalog** quantities. How to write additional filters based on these quantities was also straightforward to comprehend.

Subsequently the framework (together with the small set of additional classes) was offered to students attending a school on the science and techniques of Gaia. During this school the students were asked to produce a variety of hypercubes (such as the ones in Figs. 1 and 2) based on a subset of the GUMS10 data set. The aim was to give the attendants a feel for working with data sets corresponding to the *Big Data* case. The programming experience of this audience (mostly starting Ph.D. students) ranged from almost non-existent, to experience with procedural and scripting languages, to very proficient in *Java*. Hence, although the conceptual parts of the framework were not difficult to understand for the students (what is a hypercube, what is filtering, etc.), the lack of knowledge in both the *Java* programming language, and in its philosophy and methods proved to be a significant barrier in using the framework.

Widespread opinions were for instance: “Given the *Java* programming language learning curve the framework is a huge amount of work for short term studies but is very useful for long term and more complex studies” and “*Java* needs a complete change of mind with respect to the way we are used to programming”. The framework was also offered at a workshop on simulating the Gaia catalogue data and there the attendants consisted of

a mix of junior and senior astronomers. The reactions to the use of the framework were largely the same.

On balance we believe that once the language barrier is overcome the framework provides a very flexible tool to work with. The way of obtaining the data and the fact that the user may choose the treatment of these data, enables a wide range of possibilities with regard to scientific studies based on the data.

The fact that it operates under a Hadoop system makes it an efficient way of serving data analysis of huge amounts of data with respect to conventional database systems, due to the nature of the requests presented by the users in the seminars: “They must be completely customizable for any statistics or studies a scientist wanted to develop with the source data”.

Another interesting point is the way the data is presented on output. It can be parsed with any data mining software that can represent graphical statistical data due to its simple representation, and it can also be understood by the users themselves without major issues.

## 6. Conclusions

In this paper, we have presented a framework that allows us to easily build data mining hypercubes. This framework fills the gap between the computer science and scientific (e.g. astrophysical) communities, easing the adoption of cutting-edge technologies for accomplishing new scientific research challenges not considered before. In this respect the framework adds a layer on top of the Hadoop MapReduce infrastructure so that scientific software engineers can focus on the algorithms themselves and forget about the underlying distributed system. The latter provides a new way of working with big data sets such as the ones that will be produced by ESA’s Gaia mission (only simulations currently available).

Furthermore, we explored the suitability of current commercial Cloud deployments for these types of work flows, analyzed the application of novel data storage model techniques currently being exploited in parallel DBMS (i.e. column-oriented storage) and benchmark the solution against other popular data analysis techniques on top of Hadoop. On one hand, the column orientation has proven to be very effective for the generation of hypercubes as the number of columns involved in the process is often much less than the ones available in the original data set (the facts table in a data mining *star* schema), especially in scientific contexts where each study often focuses on a very specific area (astrometry or photometry in the astrophysical field for instance). On the other hand, the fact that the framework focuses on a specific field (generation of hypercubes), makes it better in terms of performance than other well-known solutions like Pig and Hive under the same conditions for the data processing. We argue that this is one aspect that must always be considered (tradeoff of the generality vs. performance) as too generic solutions may penalize performance as they need more logic inside to cope with the variety of features they provide.

In addition, we show several results that suggest the architecture to be considered for binary data in Hadoop work flows and conclude that it is always better to compress the data set as the CPU time for decompression is much less than the extra I/O overhead for reading the uncompressed counterpart. We also prove that light compression techniques, such as Snappy, are more suited for analytical work flows than more aggressive compression techniques (which are aimed at increasing data transfer rates).

Last but not least, the framework can also be used in any other discipline or field, as it is very common to use hypercubes (and pivot tables) for summarizing big data sets, or for providing business intelligence capabilities. Using the framework in other disciplines can be done without losing its generality (much needed due to the diversity of studies that can be made with scientific data sets).

## 7. Future work

There are several lines of work that have been opened by this research, including:

- Extensions or internal optimizations of the framework.
- Benchmarking against other possible solutions such as the ones provided in the data mining extensions of commercial (parallel) DBMS, or other solutions being currently developed within the Hadoop ecosystem which aim at providing near-real time responses for queries that return small data sets.
- Use other already existing implementations of the column-based approach and benchmark against the improvements already made with more naive implementations.

Some possible extensions are the ability to automatically use data coming from different sources by joining and filtering them in a first MapReduce phase with the user-supplied field and constraints respectively, and computing the hypercube requested in a subsequent phase. This would be ideal for raw data analysis. Furthermore, another internal optimization might be to use binary types instead of text for the keys when the hypercubes are simple enough (as shown in the benchmark), as the usage of long text strings can cause a slowdown in the sorting, hashing and shuffling stages.

The framework's efficiency should be benchmarked and compared to new technologies that aim at producing near-real time responses by both working in memory as much as possible, and by pulling the intermediate results of the internal computations directly from memory instead of disk. These tests should not only focus on the speedup and scale-up, but also analyze different workloads involving geometry and time series. This comparison should also take into account the different data storage model layouts described in this paper (particularly the column input format), as well as the data aggregation ratio for each particular hypercube (and the best algorithms to apply in each case).

## Acknowledgments

This research was partially supported by Ministerio de Ciencia e Innovación (Spanish Ministry of Science and Innovation), through the research grants TIN2012-31518, AYA2009-14648-C02-01 and CONSOLIDER CSD2007-00050. The GUMS simulations were run on the supercomputer MareNostrum at the Barcelona Supercomputing Center—Centro Nacional de Supercomputación.

## References

- [1] F. Mignard, Overall science goals of the Gaia mission, in: C. Turon, K.S. O'Flaherty, M.A.C. Perryman (Eds.), *ESA SP-576: The Three-Dimensional Universe with Gaia*, 2005, pp. 5–+.
- [2] Euclid. Mapping the geometry of the dark Universe, Definition Study Report, Tech. rep., European Space Agency/SRE, July, 2011.
- [3] Z. Ivezić, J.A. Tyson, LSST: from science drivers to reference design and anticipated data products. *ArXiv e-prints arXiv:0805.2366*.
- [4] P. Dewdney, P. Hall, R. Schilizzi, T. Lazio, The square kilometre array, *Proc. IEEE* 97 (8) (2009) 1482–1496.
- [5] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, *Commun. ACM* 51 (1) (2008) 107–113.
- [6] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, M. Stonebraker, A comparison of approaches to large-scale data analysis, in: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD'09*, ACM, New York, NY, USA, 2009, pp. 165–178.
- [7] J. Ekanayake, S. Pallickara, G. Fox, MapReduce for data intensive scientific analyses, in: *Proceedings of the 2008 Fourth IEEE International Conference on eScience, ESCIENCE'08*, IEEE Computer Society, Washington, DC, USA, 2008, pp. 277–284.
- [8] T. Gunarathe, T.-L. Wu, J. Qiu, G. Fox, MapReduce in the Clouds for science, in: *2010 IEEE Second International Conference on Cloud Computing Technology and Science, CloudCom, 2010*, pp. 565–572.
- [9] S.N. Srirama, O. Batrashev, P. Jakovits, E. Vainikko, Scalability of parallel scientific applications on the Cloud, *Sci. Program.* 19 (2–3) (2011) 91–105.
- [10] A. Floratou, J.M. Patel, E.J. Shekita, S. Tata, Column-oriented storage techniques for MapReduce, *Proc. VLDB Endow.* 4 (7) (2011) 419–429.
- [11] A.F. Gates, O. Natkovich, S. Chopra, P. Kamath, S.M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, U. Srivastava, Building a high-level dataflow system on top of Map-Reduce: the Pig experience, *Proc. VLDB Endow.* 2 (2) (2009) 1414–1425. URL <http://dl.acm.org/citation.cfm?id=1687553.1687568>.
- [12] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive: a warehousing solution over a Map-Reduce framework, *Proc. VLDB Endow.* 2 (2) (2009) 1626–1629. URL <http://dl.acm.org/citation.cfm?id=1687553.1687609>.
- [13] A.C. Robin, X. Luri, C. Reylé, Y. Isasi, E. Grux, S. Blanco-Cuaresma, F. Arenou, C. Babusiaux, M. Belcheva, R. Drimmel, C. Jordi, A. Krone-Martins, E. Masana, J.C. Mauduit, F. Mignard, N. Mowlavi, B. Rocca-Volmerange, P. Sartoretti, E. Slezak, A. Sozzetti, Gaia Universe model snapshot, *A&A* 543 (2012) A100.
- [14] K.M. Grski, E. Hivon, A.J. Banday, B.D. Wandelt, F.K. Hansen, M. Reinecke, M. Bartelmann, HEALPix: a framework for high-resolution discretization and fast analysis of data distributed on the sphere, *Astrophys. J.* 622 (2) (2005) 759.
- [15] Y. Kwon, M. Balazinska, B. Howe, J. Rolia, A study of skew in MapReduce applications, in: *5th Open Cirrus Summit*, 2011.
- [16] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.
- [17] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, Dremel: interactive analysis of web-scale datasets, in: *Proc. of the 36th Int'l Conf on Very Large Data Bases*, 2010, pp. 330–339. URL <http://www.vldb2010.org/accept.htm>.
- [18] J. Krueger, M. Grund, C. Tinnefeld, H. Plattner, A. Zeier, F. Faerber, Optimizing write performance for read optimized databases, in: *Proceedings of the 15th International Conference on Database Systems for Advanced Applications, Volume Part II, DASFAA'10*, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 291–305.