SparkArray: An Array-based Scientific Data Management System Built on Apache Spark

Wenjuan Wang*, Taoying Liu*, Dixin Tang†, Hong Liu*, Wei Li*, Rubao Lee‡,
*Institute of Computing Technology, Chinese Academy of Sciences
{wangwenjuan, lty, liuh, liwei}@ict.ac.cn
†University of Chicago
totemtang@uchicago.edu
‡The Ohio State University
liru@cse.ohio-state.edu

***Abstract*—With the highly demanded requirements for manipulating large scientific datasets, scientists are in need of flexible cluster-level software to execute fast scientific data analysis. In this paper, we discuss whether the Apache Spark framework is suitable for scientific data management. We present our system SparkArray, which extends Spark with a multidimensional array data model and a set of common used array operations (e.g., filter, subarray, smooth and join). We present analysis and performance evaluation results on different implementation methods for executing array operations on clusters. We also compared the performance of SparkArray to SciDB, a recent scientific database management system, using the workloads of the Standard Science DBMS Benchmark (SS-DB). The results show that SparkArray is feasible alternative solution for large-scale scientific data management, especially when scientists require fast data loading or one-time data analysis on large scientific datasets.**

I. Introduction

Scientific research has entered the big data era. Nowadays, scientists from different domains, for example life science, cosmology, and particle physics, often face the similar challenge of how to efficiently store and analyze very large datasets, which are frequently generated during large scientific experiments and fine-grained observations. For example, the Large Synoptic Survey Telescope (LSST) [1][2] will generate over 7TB compressed image data every night, while a process performed on a gene-sequencing machine may produce a terabyte of result data. Therefore, it is a critically important task for computer scientists and engineers to provide efficient computing infrastructures for the domain scientists in order to alleviate their burdens of big data management.

Unlike conventional data-oriented computer applications, such as transaction systems or Web sites, scientific data management has its own unique requirements.

1) First, scientists prefer easy-to-use and flexible software systems that can hide hardware complexities (e.g., multi-core CPUs and large-scale computer clusters) and provide quick integration of various analysis tools (e.g., MATLAB and R).
2) Second, scientific data management requires both high throughput and low latency of executing numerical analysis on very large scales of scientific data. Especially, considering the nature of exploring the unknown world, quick query response is very desirable for interactive scientific data analysis.
3) Finally, with the increasingly large data sets generated by quick upgrades of refined scientific experimental devices or telescopes, a computing infrastructure should adopt a scale-out model to allow linear scalability with the available computing/storage devices.

Scientists usually rely on file-systems or relational databases as the storage and analysis platform to manage their scientific data. However, these solutions, which are generally developed for general purpose data management, have limitations to satisfy the combination of the unique requirements of scientific data management. A file-system based solution, which means that data owners and users directly access underlying file systems for data read and write, can provide the maximal flexibility for scientists to develop their applications. However, its low-level nature often causes extra burdens and redundant development efforts to solve common issues of data privacy, fault-tolerance, software portability, and performance optimizations.

A database based solution can take advantage of high-level features of a relational database management system (DBMS). For example, a RDBMS can provide simple data accessing interface (often SQL-based) and built-in data indexing structures (e.g., a B+-tree). However, most existing RDBMSs are not specifically designed to handle scientific datasets. Instead, their major functionalities are optimized for business data (i.e., banking/shopping transactions). Although there are already some database research efforts to implement scientific data oriented database systems, such as SciDB [3][4][5], RIOT [6], and RasDaMan [7], such systems (and conventional relational database systems) have two mismatches to fully satisfy the requirements of scientific data management. First, the data processing model of database systems is called *"first-loading, then-query"*, which limits their support to the in-situ scientific data processing or instant raw data analysis without a pre-loading phase. Second, unlike modern big data processing frameworks such as Apache Hadoop [8] and Apache Spark [9], database systems are not open and flexible enough for users to easily integrate multiple software tools, which is highly

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

demanded by scientists to execute comprehensive data analysis jobs.

In this paper, we present our solution (called SparkArray) for scientific data management on large-scale clusters. SparkArray is based on Apache Spark, which is the state-of-the-art cluster computing framework being widely used by many big data owners. As a general purpose system, Spark's main technical merits include in-memory fault-tolerant computing based on the abstraction of Resilient Distributed Dataset (RDD) [10] and easy integration with various tools for multiple purposes (e.g., machine learning, streaming, graph data analysis). A dominating approach of deploying Spark in practice is to run Spark on top of the Hadoop Distributed File System (HDFS), which is the most widely used file system for big data management on computer clusters.

Despite its attractive features for big data processing and quick adoption in many data processing environments, however, Spark does not provide native support for scientific data management. As the representative scientific database SciDB has demonstrated [3], an **array** data model and corresponding data operations are the key to optimize scientific data storage and analysis. Therefore, for the purpose of using Spark for scientific data management, an important problem is how to efficiently implement the array model and operations in Spark and HDFS without losing Spark's existing technical merits for large-scale data management. A solution to this problem is based on both 1) how to efficiently store array data in HDFS and 2) how to parallelize the executions of array operations on clusters. Although the first issue is simple since SparkArray can choose one of existing HDFS data formats (e.g., TextFile, RCFile [11]), the second issue is not trivial considering the complexities of various array operations – that is what this paper is focused on.

In summary, our contributions in this paper are two-fold.

1) First, we describe the design and implementation of SparkArray, an array-oriented scientific data management system running on clusters, focusing on describing how to efficiently implement common array operations based on the Spark programming framework.
2) Second, we present performance comparison results between SparkArray and SciDB (the state-of-the-art scientific database system) using the SS-DB (Standard Science DBMS Benchmark)[12] workloads. Our results show that SparkArray has significant advantages over SciDB considering both data loading times and total query execution times.

The remaining of this paper begins by introducing the array model and operations (Section II). Next, we introduce the implementation of unary operations (Section III) and binary operations (Section IV). Then we present our performance evaluation results (Section V). Finally, we introduce related work (Section VI) and conclude this paper (Section VII).

TABLE I
ARRAY OPERATIONS

Unary Operation		Binary Operation
Independent	Bounded Operation	
$subarray$, $filter$ $slice$, $apply$	$smooth$, $regrid$, $cluster-extraction$	$join$

II. THE ARRAY DATA MODEL

A. Data Structure

SparkArray uses the multidimensional array [13] as its underlying data structure. Like a relational table, a data array is well structured with an associative data schema. The schema includes a list of dimension values and a list of attribute values. The former provides a quick accessing index to the array data, while the latter is used to describe the array data.

For example, we create a 2-dimensional array to represent a simple image where each cell has two attributes: pixel and color.

$$Image < pixel : int, color : string > [i = 0 : 99, j = 0 : 99]$$

We regard i and j as dimension values, which can improve execution performance for different types of queries by speeding up array accesses.

B. Operations

In SparkArray, array operations [14] are classified into two categories: unary array operations and binary array operations. Unary operations are further classified into independent operations and bounded operations. Table I lists all the array operations currently supported by SparkArray.

An $independent$ array operation is one that the processing of one part is independent from other parts in the input array. The representatives of independent operators are subarray, filter, and slice (details in the next section).

A $bounded$ array operation is one that the processing of one part needs attribute values from other adjacent parts in the input array. Smooth, regrid and cluster extraction are typical representatives of this category (details in the next section).

Unlike a unary operation whose input is only one array, a binary array operation needs to process two arrays. Typically, a binary operation implies a rule that two elements are composed into the third element, for example the join operation. Both join on arrays and join on relational tables need to define a join key to correlate the two input elements. However, the rule of determining the join key is different. Join on arrays always sets index as the fixed join key, while join key of relational tables is specified by users, which can be arbitrarily picked up from the common attributes of two elements.

III. UNARY OPERATION IMPLEMENTATIONS

A. Implementation Methods

In general, dealing with a large array on a cluster is implemented by dividing the array into multiple smaller subarrays, which are called chunks [15]. According to the differences in how the chunks are processed, there are three methods to

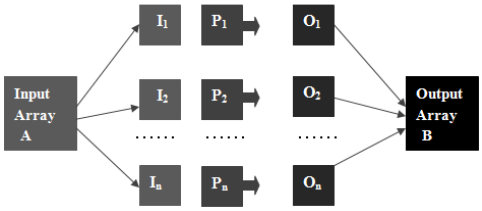

Fig. 1. Input Array A is divided into a set of chunks: I_1 to I_n. Each chunk is independently handled by a processor P_i. And all output chunk O_1 to O_n consist output array B.

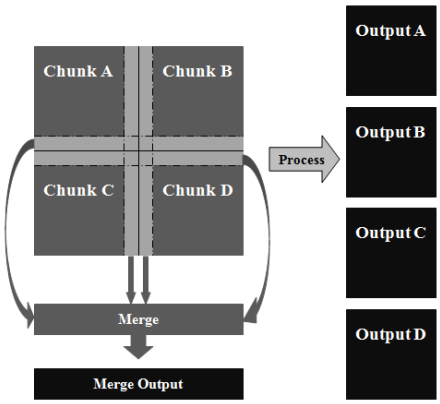

Fig. 2. An example of merging. There are four input chunks A, B, C, D. They are all processed independently at first. Then the intermediate boundary value is pulled to the same node in the phase of merge.

implement the array operations, namely independent, merge and overlap [16].

1) Independent: The independent method reorganizes (often by partitioning) an array into a number of chunks that do not have any dependency among each other, as shown in Figure 1. A cell in result array is computed by the corresponding cell in source array which is in the same position. The independent method offers the best parallelism. Obviously, it is best suitable for the unary independent operations including subarray, filter and slice. Although the independent method can also be used to implement bounded operations, it needs extensive data duplication in chunks to remove possible data dependencies.

2) Merge: To better implement bounded operations, SparkArray provides one method called merge, which is illustrated in Figure 2. Like the independent method, the merge method generates intermediate results in parallel, but an additional phase is used to merge the boundary of chunks to derive final array. A simple implementation of the merge phase is to pull all intermediate data to one node in cluster. However, the amount of the intermediate data may go beyond the capability of one node and cause memory overflow. To solve this problem, an improved implementation in SparkArray is

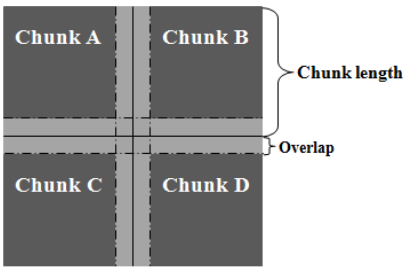

Fig. 3. An example of overlapping. There are four input chunks A, B, C, D. The light shade part is overlap. Each chunk can be processed independently with overlap data. The boundary of chunk is indicated by the dotted line.

to merge adjacent data step by step.

3) Overlap: The overlap method is another technique to implement bounded operations in SparkArray, as shown in Figure 3. Its main idea is to first divide an array into a set of chunks with fixed-size overlapping portions along each array dimension. Then, the data operations can take advantage of the overlaps to process each chunk independently, like the independent method. Compared to the merge method, the overlap method does not need an additional merging phase.

B. Independent Operation

SparkArray has implemented four operations, namely filter, apply, slice and subsample, which are implemented by the independent method.

1) Filter: Filter is applied to the attribute and dimension values of the input array. It returns a result array with the same dimension as source array. Except for the cells that do not satisfy the conditions to be set to empty value, the value of others is the same as input value. For example, we create a 3×3 array by random number within 0 to 100. And then we want to filter the values that are greater than 50. The result is:

$$\begin{pmatrix} 56 & 23 & 12 \\ 32 & 45 & 85 \\ 75 & 93 & 4 \end{pmatrix} \xrightarrow{Filter} \begin{pmatrix} & 23 & 12 \\ 32 & 45 & \\ & & 4 \end{pmatrix}$$

2) apply: Apply produces a result array which includes all attributes present in the source array, adding the newly created attributes. For instance, we create an array called container with an attribute called pound. And we apply the expression 0.45×pound to compute a new attribute named kilogram:

$$\begin{matrix} pound \\ \begin{pmatrix} 0 \\ 100 \\ 200 \\ 300 \\ 400 \\ 500 \\ 600 \\ 700 \end{pmatrix} \end{matrix} \xrightarrow{Apply} \begin{matrix} pound & kilogram \\ \begin{pmatrix} 0 & 0 \\ 100 & 45 \\ 200 & 90 \\ 300 & 135 \\ 400 & 180 \\ 500 & 225 \\ 600 & 270 \\ 700 & 315 \end{pmatrix} \end{matrix}$$

3) Slice: Slice is an operation that is applied to the array structure. Slice refers to project array along a certain dimension.

We create a 4×4 array and fill it with values. This example selects the left column from the array.

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \xrightarrow{Slice} \begin{pmatrix} 0 \\ 4 \\ 8 \\ 12 \end{pmatrix}$$

4) Subsample: Subsample refers to extract a subarray from the source array.

We create an array called Image. In the example, we return cells that are in both right two columns and the last two rows.

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \xrightarrow{Subsample} \begin{pmatrix} 10 & 11 \\ 14 & 15 \end{pmatrix}$$

C. Bounded Operation

A bounded operation requires computations that aggregate multiple neighbors to obtain the result. SparkArray has implemented smooth, regrid and cluster extract operations.

1) Smooth: Smooth needs to calculate the average value over a window around a cell in the input array, and replace center point value with average. Generally, we regard the upper-left corner element as center point. The dimension of result array is as same as the original array.

For example, we create a 4×4 array and a 3×3 window as shown in Figure 4. And the final value in result array is as follows:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \xrightarrow{Smooth} \begin{pmatrix} 5 & 6 & 6.5 & 7 \\ 9 & 10 & 10.5 & 11 \\ 11 & 12 & 12.5 & 13 \\ 13 & 14 & 14.5 & 15 \end{pmatrix}$$

The smooth operation can be implemented using the independent, merge or overlap method. We here describe the independent implementation. For computing result of one cell, we require the neighbor cells of it. By means of aggregating surrounding cells and computing average in parallel, the Spark program outputs the results as $Tuple < key, value >$ format whose key is index and value is the average. The advantages are: 1) Programming on Spark is simple. 2) Compared to other methods, it achieves the highest degree of parallelism. 3) There is no shuffle [17] stage throughout the whole process which provides a contribution to reduce data transfers in network and disk I/O in second phase. However, the typical demerits of this approach are the highest rate of data and the most data transfers in network when generating intermediate results. For instance, if the window size is 5, the data duplication rate staggering reaches up to 25x. In total, by our analysis, this method is advantageous especially in the case where the adjacent cells are located in the same node of cluster.

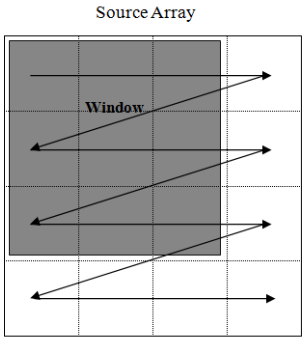

Fig. 4. An example of smooth. This example represents a 4×4 array and a 3×3 window for smooth. The black arrow shows the trajectory of the window. Each movement, we replace the original value with the average of cells in window.

2) Regrid: Regrid relies on compressing or extending the source array so that the result array may contain less or more cells. In this article, our regrid example always mains compression regrid.

Now we present a concrete sample that applies a 2D array to simulate a two-dimensional astronomical image. Astronomers hope for decreasing image resolution by 10:3. To achieve this goal, regrid requires every 10×10 input cells to calculate 3×3 output cells. Like smooth, regrid can also be implemented by all the three methods. We propose an example for the regrid operation. The example divides the source array into 4 partitions and surveys the max value of each one.

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \xrightarrow{Regrid} \begin{pmatrix} 5 & 7 \\ 13 & 15 \end{pmatrix}$$

3) cluster extract: Cluster extraction is an important operation for image data. An 2D astronomy image is simulated by a two-dimensional array. The cluster extraction means extracting the polygon and center coordinate information from cluster which is constituted by a set of bright pixels as shown in Figure 5. Except the center point, most of the result array cells are empty. In order to compute the center of per cluster, system needs all pixel values that consist the cluster. In terms of physical significance, a cluster means a star in the sky. The overlap method is suitable for the implementation of cluster

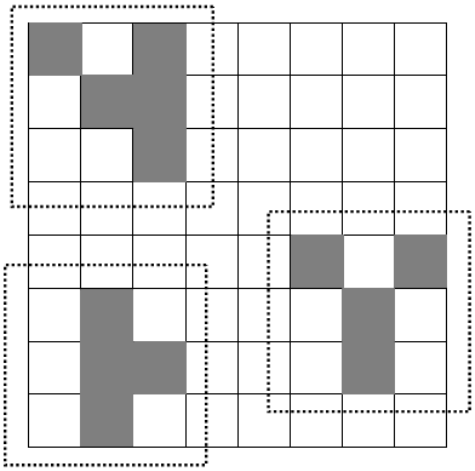

Fig. 5. An example of cluster. This example represents a 8×8 array. The shadow cells illustrate that the points are bright enough and the dashed squares figure out three clusters in this array.

		Cell of Array B	
		Empty	Non-Empty
Cell of Array A	Empty	Empty Cell	Empty Cell
	Non-Empty	Empty Cell	Cell contains attributes of both A and B

Fig. 6. An instance of array join, the gray portion represents cells in output array. Only in the case where values in both A and B are not empty, the result is not empty like lower right part.

extract. We suppose there is a limit about size of cluster polygon and we define the limit size as D. Then D is set to the overlap of per chunk so that each chunk can be processed separately from other blocks. To avoid repetition, each cluster can be represented by the center point.

IV. Binary Operation Implementations

In addition to unary operators above, SparkArray also supports join on arrays, a binary array operation. Join combines the attributes of two input arrays (namely A and B in the rest of this paper) at matching dimension values. If a cell in either A or B is empty, the corresponding cell in the result is also empty as shown in Figure 6.

This paper focuses on the join operation specifically in astronomy applications. In that case, photographs taken by telescope at different times have overlap in position. If we are interested in the overlap portion, we need join two images. Differing from the join above, in our application, the range of index in two input arrays are not from $(0, 0)$ to (n, n) but from $(start_x,start_y)$ to (end_x,end_y). The index means the absolute position of this image in the world coordinate system which represents the whole sky as shown in Figure 7. So the result

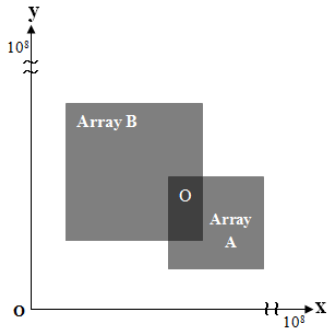

Fig. 7. An example of join in astronomy, both array A and array B are two input array, the result array is the darker part.

array is the overlapped portion. For instance, we suppose that the scope of Array A is from $(700, 200)$ to $(1100, 600)$, while the range of Array B is from $(100, 300)$ to $(800, 1000)$. And the result of join is the overlapped portion in Figure 7 which starts at $(700, 300)$ and ends at $(800, 600)$.

SparkArray has implemented three join algorithms. Two of them are the broadcast join algorithm and the repartition join algorithm. Furthermore, SparkArray provides the third one, the repartition filter join algorithm, which improves the repartition join algorithm.

A. Broadcast join

This method is extensive used in the circumstance where array A is much smaller than array B. At first, array A is broadcasted to the memory of every node in the cluster whose purpose is to guarantee that no matter which node the task runs on, the node can cache array A directly instead of accessing data on other nodes. So that the Spark program can run locally with only map phase [17]. Broadcast join averts shuffling large array B among physical machines, and thus it completely removes the reduce phase. The drawback of broadcast is that it causes computations on driver node and large memory cost which may lead to memory overflow.

Pre-processing: According to the features of array that it always regards index as join key, we optimize the join by computing the overlap portion of two arrays in advance. Then we filter array A to leave the overlap portion which is really required. Finally, we send the filtered array A to each node in order to decrease the amount of data transfers among nodes.

B. Repartition join

Repartition join is the most widely used way to implement join. The key idea of repartition join is to divide two arrays via the same partitioner function using index as key. The disadvantage is that re-partitioning two arrays will produce a shuffle stage and time-consuming inter-node data transfers, while the advantage is that the programming is fairly convenient for programmers.

Pre-processing: The shuffle overhead in the repartition join can be removed if both A and B have already been partitioned on the join key before the join operation. This

can be implemented by pre-partitioning the two arrays on the join key when arrays are generated or loaded in the HDFS. After that, A and B can be used for join directly without an additional repartition step.

However, the applicability of pre-partitioning technique is always limited, since we must use the same function to partition arrays physically, otherwise it still repeats to repartition two arrays. One solution is to apply all arrays with hash partitioner to limit partition function.

Astrophysicists face a challenge of data skew when they use existing join strategies including broadcast join and repartition join. When the collocated data is exactly divided into only a few nodes, it would cause the problem of data skew. Since the procedure does not obtain the range of array coordinate in advance, it seems that it is like a black box. To solve this issue, this paper proposes the following optimization.

C. Repartition filter join

This method is based on the characteristics of an array in astronomy application. Firstly, the program traverses two arrays and records the maximum and minimum coordinate value of two arrays respectively. Then it computes the intersection range of coordinates, e.g., $Range_i(x_i,y_i,x_j,y_j)$ which is the dark portion in Figure 7. So that we just need to filter two arrays to extract the overlapped part which is effective. Finally we run join program on the filtered array (namely A' and B').

Our optimization is only applicable to array because array always regards coordinate values which store position information as join key. For other tuples (e.g. $T < key, value >$) which join key does not relate to position information, this method does not work. However, we discover the fact that in the situation where range $Range_i$ is similar with A or B, the filter phase does not accelerate the time but slows down the processing speed. We summarize the observation in three situations.

1) If the coverage rate is few such as 10%, we directly recommend filtered function no matter how many cells in small array.
2) If the coverage rate is approximately 50% and the small table is greater than **D**6, we choose repartition. Otherwise we still suggest the filtered method.
3) Suppose the coverage rate is around 70% and the small table is larger than **D**5, we choose repartition function; if not, we use filtered technique. The size of D1∼D6 is shown in Table III.

Pre-processing: If we know the coordinate range of two arrays in advance, we can treat them as parameters of the function, calculate the range of intersection directly and save the time spent on traversing two arrays.

D. Discussion on Multiple-array Join

SparkArray does not provide built-in implementation for direct join among three or more arrays. Such an operation can be expressed in a sequence of binary join operations. It is our future work to implement the optimization of join reordering.

TABLE II
SS-DB DATA SCALES.

Configurations	Small	Normal	Large	Very large
Image Width	3750	7500	15000	30000
Total Size	99GB	0.99TB	9.9TB	99TB

V. PERFORMANCE EVALUATION

In this section, we present our performance evaluation results on SparkArray. After introducing our experimental environment and workloads (Section V.A), we will present result of three groups of experiments:

1) In Section V.B, we show the performance of different implementation methods for unary array operations (smooth and cluster extract).
2) In Section V.C, we compare different join implementation algorithms with and without enabling the preprocessing phase.
3) In Section V.D, we present the performance of SparkArray and SciDB for the SS-DB benchmark measuring both data loading times and query execution times.

A. Experimental Environment and Workloads

All our experiments are run on a Spark cluster with 5-nodes. We use one node as master and five nodes as workers. Each node has 24 cores, 96 GB memory and 8 TB disk space. The operating system is CentOS 6.6 with Linux kernel 2.6. We use Spark 1.4 and java 1.7 for our experiments. The HDFS is used to store all data during experiments and the duplicate factor of the block is set as 3. All of the results are calculated by trimmed mean with 10 repetitions.

Our experiments were conducted using the Standard Science DBMS Benchmark (SS-DB). SS-DB is a science data benchmark proposed by the MIT CSAIL lab, which contains two parts: data and queries. The data part refers to a simulation of spatial data which are represented by 2-D arrays. Offered by the Large Synoptic Survey Telescope (LSST), each array simulates an image of a portion of sky.

There are four major scale configurations of SS-DB dataset as shown in Table II. Each of them consists of a variety of sky images taken from telescopes. We focus on the normal dataset. Each 2-D array is in the same local coordinate system, which simply starts at (0,0) and ends at (7499,7499). Hence, each array has 56,250,000 cells and every cell has 11 attribute values which simulate the basic information of the pixel. A normal array is 2.48 GB in size and consists of 400 images which mean the pictures taken on different times. In total, the size of benchmark is 0.99 TB.

Besides the raw data, SS-DB also supports another two types of datasets: cook data and group data. In the aspect of physical meaning, cook data refer to stars, while group data refer to galaxies. A customized $findstar$ function manipulates raw data to generate stars, while a $groupstar$ function derives galaxies from cook data.

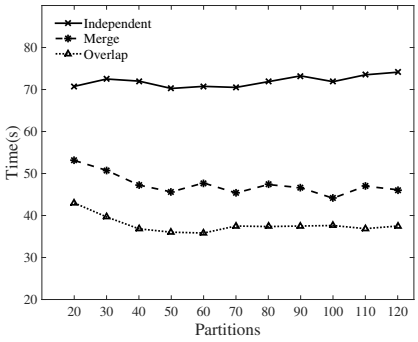

Fig. 8. the Result of Three Algorithms for Smooth.

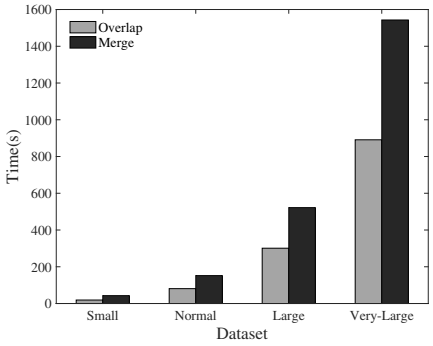

Fig. 9. the Result of Three Algorithms for Cluster.

There are nine queries applied to the datasets of SS-DB. They are briefly introduced as below.

1) Q1: Browse all the raw data and calculate the average of one specific attribute value.
2) Q2: Extract cluster from one image using a different threshold.
3) Q3: Reduce the noise and reduce the raw data for a 10:3 proportion.
4) Q4: Compute the average value of one attribute in each observation which locates in a specific area.
5) Q5: Obtain the information of observations whose polygons locate at a specific range.
6) Q6: Find all tiles which contain more observations than a threshold in a specific slab.
7) Q7: For all groups, find those whose centers fall in a specified rectangle.
8) Q8: Find the groups whose trajectories intersect a specified area, and export raw data which consist those groups. Trajectory is defined as the order of centers of the observations in a group.
9) Q9: Define trajectory as a sequence of polygons that refer to the boundary of the observation group. Find the groups whose trajectories intersect a specified area, and produce the raw data of those groups.

All of the above queries can be categorized into three types according to data formats they focus on.

- Queries on raw data (Q1∼Q3)
- Queries on cook data (Q4∼Q6)
- Queries on group data (Q7∼Q9)

In SparkArray, all of the queries are implemented using the data operations we had introduced in this paper. Therefore, the execution of each query is essentially a set of executions of array operations.

B. Performance of Unary Operation Implementations

1) Smooth: We run experiments on one image of large dataset which is 13GB in size.

In the implementation of Spark program, the amount of partitions during shuffle stage is extremely significant so that we adjust it from 20 to 120 in order to measure the best performance as shown in Figure 8.

TABLE III
SMALL ARRAY SCALES.

Test Set	D1	D2	D3	D4	D5	D6	D7
Width	1000	2000	3000	5000	7000	15000	20000
Size	58.0M	229.0M	514M	1.4G	3.2G	13.0G	23G

We explore the rule that from left to right, with the increase of the amount of partitions the performance of algorithm improves, especially overlap and merge method. The performance is gradually stable while the count of partitions is larger than 60 or 70. Because the magnitude of partitions determines the number of tasks, if the amount of tasks is too few, it cannot make extensive use of CPU resources. Under this condition, if aggregate operators appear unfortunately, tasks may suffer from memory pressure even memory overflow. Therefore, the result recommends to set the number of partitions as 60.

Obviously, compared with other two algorithms, independent always performed worse than them. That is because even if the independent has a high degree of parallelism, the high network traffic also pulls down the whole speed. As is mentioned in Section III, the overlap method is suitable for the smooth operation.

2) Cluster extract: We run experiments to execute the cluster extract operation on four scale configurations shown in Table II.

Figure 9 shows the results. We can explore that for this operation, similar with smooth, the overlap method has better performance than the merge method. This is because the merge phase in the cluster extract operation contains a number of computation tasks. Since the overlap method does not need an extra merge phase, it can finish the operation in a short time. However, the overlap method has its overhead because it needs extra storage space to place data in the overlapping portion.

C. Join Performance

Array B is one of largest dataset which is 50 GB in size, while array A is from D1 to D7 in size as shown in Table III. Another parameter is the proportion, which means how much

TABLE IV
DATA TRANSFORMATION AMONG NETWORKS

Dataset	Repartition	Filtered(50%)	Broadcast
D1	50.05G	50.03G	0.28G
D4	51.40G	50.70G	7.00G
D7	73.00G	61.50G	115.00G

the overlap portion takes up of B. We choose the proportion of 10%, 50%, and 70%, respectively.

Figure 10 and Figure 11 show the results of join algorithms with different coverage rate in two process cases. The x-axis shows the dataset size, and the y-axis shows the completed time in seconds. We have four important conclusions.

At first, we analyze the standard repartition join. Moving from left to right, we can see that as array A becomes larger, the time for join execution is increased. When array A is small, its size is not a major factor. Instead, the overhead brought by tasks starting and scheduling during process dominates the actual speed. The same pattern can be seen in Figure 10 and Figure 11.

Secondly, we focus on the filtered repartition function which we call filtered in brief. Taken the without preprocessing case for instance, in Figure 10 (1), filtered algorithm always runs prior to standard repartition algorithm. This is because our filtered repartition algorithm computes the range of two arrays firstly, then filters the cells that do not locate in the overlap portion. Filtered method can avert the issue of data skew appeared in standard algorithm effectively. However, in Figure 10 (2) and Figure 10 (3), as A becomes larger, the time of improved overruns that of standard. This is because the time cost of computing range is more than the time saved by joining two smaller arrays. And in Figure 10 (2) and Figure 10 (3), two methods intersect at a point. It means that once the size of array A surpasses the point, the overlap range computation provides no contributes to speedup join and filtered method does not take the leading position. We record the point that if the amount of A does not exceed the point or the proportion is less than that rate, we suggest filtered function and vice versa.

Thirdly, we consider the broadcast method. When A becomes smaller, its performance is getting better. If A contains 25 million records or less, broadcast approach is always superior to filtered repartition. However, broadcast joins performance degrades rapidly with A gets larger. This is because spreading A throughout the network has become a bottleneck. Table IV lists the amount of data transferred among physical nodes taken 50% for example.

In the end, we show the comparison between the two cases: without preprocessing and with preprocessing, and study how much improvement can be obtained by preprocessing. Figure 11 illustrates that the rule of preprocessing is as same to without preprocessing. In general, pre-processing improved 40% approximately. The time cost of pre-processing is different, and the average is several minutes.

D. Comparison with SciDB

In this section we show the performance comparison results between SparkArray and SciDB for executing the SS-DB benchmark. The experiment has two parts: 1) The load part is to load the required data (including raw data, cook data and group data) into SparkArray and SciDB; 2) The query part is to execute the nine queries in the two systems. Table V shows the execution times of data loading and query executions for the normal data set (see Table II). We have several significant observations.

1) Data Loading: Both SciDB and Spark parallelize loading raw data on each node. When comparing the data loading times, SparkArray is much faster than SciDB. For the raw data, cook data, and group data, SparkArray's speedups over SciDB are 19x, 17x and 24x.

When SciDB loads data, it needs data re-formatting and compression. However, data loading in Spark only involves data transfers from the local file system to HDFS, thus it is much faster than SciDB.

2) Data Queries: The detailed analyses about all queries are as follows:

- Q1 aggregates one property of all records of the array.
- Q2 needs cluster extraction (called findstar in SS-DB).
- Q3 needs subsample, filter and smooth.
- Q4 is similar to Q1, but runs on observation data.
- Q5 needs filter running on observation data.
- Q6 needs filter and smooth.
- Q7 needs filter on group data.
- Both Q8 and Q9 need join, filter, sample and average. Q8 joins group data and group data, while Q9 joins observation data and group data.

When comparing the query execution times, the performance gap between SparkArray and SciDB varies greatly with different queries. For certain queries (Q3, Q6, Q8, and Q9), SparkArray has comparable performance with SciDB, and the speedups of SciDB over SparkArray are only at most 1.68x. But for several other queries, SciDB is much faster than SparkArray. The most extreme case is Q7, for which SciDB is 76x faster than SparkArray.

A major reason to explain the performance differences is that SciDB uses a column store while in our tests SparkArray uses a row store. When a query (e.g., Q1, Q4, Q5, and Q7) needs only a single array operation, such as filter or subsample, SciDB is much faster than SparkArray. In addition, SparkArray also has a higher cost for task startup and scheduling than SciDB, because the underlying Spark is designed for general-purpose cluster computing. In our future work, we will investigate how the modern HDFS column stores (e.g., ORCFile [18][19]) can be used to accelerate SparkArray.

3) Discussions: If we further consider the total execution time of a query, which needs adding the data loading time and the query execution time, SparkArray is much faster than SciDB for all the queries. For example, the total times of Q1 (loading raw data needed) by SparkArray and SciDB are 1041s and 74.6s, and SparkArray's speedup over SciDB is 14x.

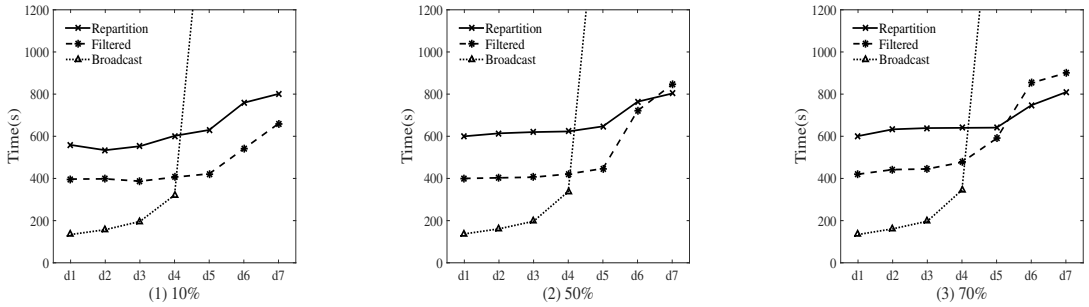

Fig. 10. The Results of three algorithms about join, without pre-processing

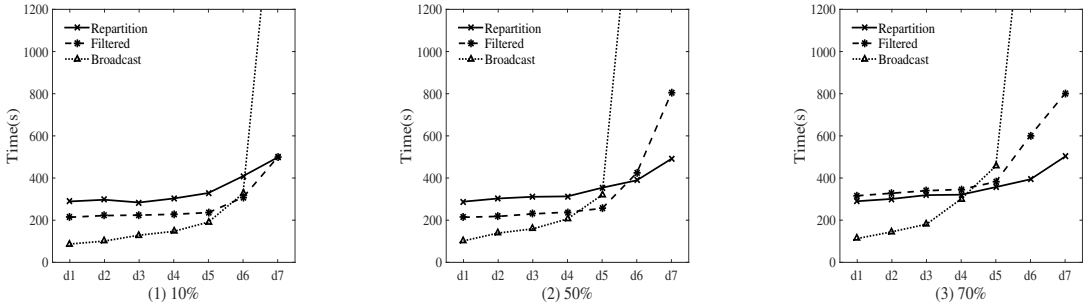

Fig. 11. The Results of three algorithms about join, with pre-processing

TABLE V
QUERY AND LOAD RUNTIMES

System	Load(s)			Query(s)								
	Raw data	Cook data	Group data	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
SciDB	1031	101	71	10	14.2	19.6	4.9	11	31.8	0.4	18.4	16.7
SparkArray	54	6	3	20.6	81.4	27.7	36.7	29.3	45.8	30.6	29.2	28.1

The experimental results confirmed our analysis in the Introduction section. Although SciDB is a database system optimized for array data, it still uses the traditional "first-loading, then-query" model. Under this model, a long data loading phase must be executed before data analysis can be executed. If the user scenario has a style of loading data once and executing queries many times, then the database model is suitable. However, a very desirable feature in the domain of scientific data management is direct data analysis on raw data. For such a scenario, SparkArray is a better solution, which requires only light-weight and fast data loading (in most cases only file copy between local file systems and HDFS) while providing acceptable (at least) query execution performance. As the experimental results shows, considering the total query execution times, SparkArray has a significant advantage over SciDB (with more than 10x speedup) if users need to execute one-time data analysis on raw data,

VI. RELATED WORK

Recently, the emergence of several cluster computing systems (e.g., Apache Hadoop, Apache Spark, Clustera [20] and Dryad [21]) reflects the increasing importance of using large-scale clusters to execute big data analysis. However, none of the systems are specially designed for scientific applications. Scientists, who are not satisfied with traditional database systems, are questioning whether these systems are suitable for scientific workloads.

In the database community, there have been several efforts to build specific database systems for scientific data management. SciDB [4] is a representative scientific database system using the array data model, which is designed to address the mismatches between the relational table structure and scientific datasets. AscotDB [25] is a data analysis system for astronomical surveys, which uses SciDB as its underlying data storage and processing engine while provides extra functionalities for data exploration and visualization. Compared to these specially designed systems, the most significant difference of our solution SparkArray is its nature of using a general-purpose cluster computing framework (Spark) as its underlying executing engine. In this way, SparkArray is more closely related to several recent Spark/Hadoop-based systems, including SciSpark [22], ViSpark [23], and SciHadoop [24].

SciSpark [22] extends Apache Spark for scaling scientific computations. It introduces the Scientific Resilient Distributed Dataset (sRDD), a distributed-computing array structure which supports iterative scientific algorithms for multidimensional

data. ViSpark [23] is an extension of Spark for GPU-accelerated MapReduce processing on array-based scientific computing and image processing tasks. Compared to these two systems, our work focuses more on how to efficiently implement various array operations on Spark.

Before the wide adoption of Apache Spark for large-scale data analysis, there were already research work to study how to extend Hadoop to support array-based data models. SciHadoop [24] is such a representative system, which aims at reducing total data transfers, remote reads, and unnecessary reads when Hadoop processes array data. Another research work [28] compared performance between Pig/Hadoop and relational databases for astrophysical data processing. SparkArray uses Spark because its RDD model has been widely recognized as a replacement for the slow MapReduce model used by Hadoop.

The study on query execution algorithms for parallel and distributed DBMSs has a long history. A recent work [27] compares the performance of MapReduce-based join implementations on large-scale clusters. Although it designs the pre-processing function, the applications it focuses on is log processing but not array processing. Considering the unique characteristics of the array data model, SparkArray introduces an additional join algorithm optimization for astronomical image data.

VII. Conclusion and Future Work

In this paper, we have presented the design, implementation and performance evaluations of SparkArray, a Spark-based scientific data management system running on computer clusters. The core contribution of SparkArray is that it augments Spark with the array data model and provides a set of array operations on top of the Spark computing framework. SparkArray can greatly help users develop scientific data analysis applications, as shown by our implementation of the SS-DB benchmark in this paper. From the performance comparison results between SparkArray and a representative scientific database system (SciDB), we draw the conclusion that SparkArray is a better solution for applications that need fast data loading or one-time data analysis.

Our future work include 1) to further improve SparkArray's performance for certain operations and 2) to optimize complex query execution processes with high-level features including join reordering.

VIII. Acknowledgment

We are grateful to the anonymous reviewers who helped improve the quality of this paper. This work is supported in part by the Hi-Tech Research and Development (863) Program of China (Grant No. 2013AA01A212 and 2013AA01A209) and International Science and Technology Cooperation Program of China (Grant No. 2013DFA10690).

References

[1] http://www.lsst.org/

[2] J. A. Tyson, "Large Synoptic Survey Telescope: Overview," in SPIE, 2002, pp. 10-20.

[3] J. Rogers, R. Simakov, E. Soroush, P. Velikhov, M. Balazinska, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, S. Zdonik, A. Smirnov, K. Knizhnik, and P. G. Brown, "Overview of SciDB: large scale array storage, processing and analysis," In SIGMOD, 2010, pp. 963-968.

[4] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker and S. Zdonik, "A demonstration of SciDB: a science-oriented DBMS," PVLDB, vol. 2, no. 2, pp. 1534-1537, 2009.

[5] M. Stonebraker, J. Becla, D. J. DeWitt, K. Lim, D. Maier, O. Ratzesberger, S. B. Zdonik, "Requirements for Science Data Bases and SciDB," in CIDR, 2009.

[6] Y. Zhang, H. Herodotou and J. Yang, "RIOT: I/O-Efficient Numerical Computing without SQL," in CIDR, 2009.

[7] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "The multidimensional database system RasDaMan," in SIGMOD, 1998, pp. 575-577.

[8] http://hadoop.apache.org/

[9] http://spark.apache.org/

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in NSDI, 2012.

[11] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang and Z. Xu, "RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems," in ICDE, 2011, pp. 1199-1208.

[12] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S. B. Zdonik, and P. G. Brown, "SS-DB: a standard science DBMS benchmark," http://www.xldb.org/science-benchmark/. [Online; August 2012].

[13] https://en.wikipedia.org/wiki/Array_data_structure#Multidimensional_arrays

[14] E. Soroush, M. Balazinska and D. Wang, "ArrayStore: a storage manager for complex parallel array processing," in SIGMOD, 2011, pp. 253-264.

[15] S. Sarawagi and M. Stonebraker, "Efficient Organization of Large Multidimensional Arrays," in ICDE, 1994, pp. 328-336.

[16] E. Soroush and M. Balazinska, "Hybrid merge/overlap execution technique for parallel array processing," in EDBT, 2011, pp. 20-30.

[17] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107-113, 2008.

[18] Y. Huai, S. Ma, R. Lee, O. O'Malley, and X. Zhang, "Understanding insights into the basic structure and essential issues of table placement methods in clusters," in PVLDB 6(14), pp. 1750-1761, 2013.

[19] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major technical advancements in apache hive," in SIGMOD, 2014.

[20] D. J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov, "Clustera: an integrated computation and data management system," PVLDB, vol. 1, no. 1, pp. 28-41, 2008.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in EuroSys, 2007, pp. 59-72.

[22] R. Palamuttam, R. M. Mogrovejo, C. Mattmann, B. Wilson, K. Whitehall, R. Verma, L. McGibbney, and P. Ramirez, "SciSpark: Applying in-memory distributed computing to weather event detection and tracking", in IEEE Big Data, 2015.

[23] Woohyuk Choi and Won-Ki Jeong, "Vispark: GPU-accelerated distributed visual computing using spark," in LDAV, 2015, pp. 125-126.

[24] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt, "SciHadoop: array-based query processing in Hadoop," in SC, 2011, pp. 1-11.

[25] J. VanderPlas, E. Soroush, S. Krughoff and M. Balazinska, "Squeezing a big orange into little boxes: The AscotDB system for parallel processing of data on a sphere," IEEE Data Eng. Bull. Vol. 36, no. 4, pp. 11-20, 2013.

[26] T. Tsuji, A. Hara and K. Higuchi, "An extendible multidimensional array system for MOLAP," in SAC, 2006, pp. 503-510.

[27] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita and Y. Tian, "A comparison of join algorithms for log processing in MapReduce," in SIGMOD, 2010, pp. 975-986.

[28] S. Loebman, D. Nunley, Y. Kwon, B. Howe, B. Balazinska, and J. P. Gardner, "Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help?" in CLUSTER, 2009, pp. 1-10.