Kaleido: Enabling Efficient Scientific Data Processing on Big-Data Systems

Saman Biookaghazadeh
Arizona State University
Tempe, Arizona
Email: sbiookag@asu.edu

Shujia Zhou
Northrup Grumman
Baltimore, Maryland
Email: szhou@umbc.edu

Ming Zhao
Arizona State University
Tempe, Arizona
Email: mingzhao@asu.edu

***Abstract*—Big-Data systems are increasingly important for solving the data-driven problems in many science domains. However, existing big-data systems cannot support the efficient processing of self-describing data formats such as NetCDF which are commonly used by scientific communities for data distribution and sharing. This limitation presents a serious hurdle to the further adoption of big-data systems by science domains. This paper presents Kaleido, a solution to this problem by enabling big-data systems to efficiently store and process scientific data. Specifically, it enables Hadoop to directly store NetCDF data on HDFS, and process them in MapReduce using convenient APIs. It also enables Hive to support queries on NetCDF data, transparent to the users. Moreover, it employs optimizations tailored to scientific data, particularly dimension-aware layouts which allow efficient execution of subset queries targeting any dimension of a multi-dimensional dataset. The paper presents a comprehensive evaluation of Kaleido using representative queries on a typical geoscience dataset. The results show that Kaleido achieves substantial speedup and space saving compared to existing solutions for storing and processing NetCDF data on Hadoop, and it also substantially outperforms the state-of-the-art solutions for supporting subset queries on scientific data.**

I. Introduction

Big data is an important computing paradigm increasingly used by many disciplines for knowledge discovery, decision making, and other data-driven tasks. Big-Data systems are typically built upon programming frameworks that can effectively express data parallelism and exploit data locality (e.g., MapReduce [1]) and storage systems that can provide high scalability and availability (e.g., Google File System [2], Hadoop HDFS [3]). A variety of high-level data services (e.g., BigTabale [4], HBase [5], Hive [6]) can be further built upon such frameworks.

These technologies are increasingly important to scientific applications from various disciplines. For example, typical geoscience models have multi-scale physical processes. With current high performance computing power, ultra-high-resolution, long-time simulations are feasible with a few thousands of computer processors. Consequently, huge amounts of data (easily over 100TB) are produced. Big-data technologies are demanded to analyze the simulation outputs. However, there are two important limitations to the use of existing big-data solutions for scientific data.

First, commonly used big-data systems do not natively support scientific data formats. Scientific data is often stored in self-describing data formats (e.g., NetCDF [7], HDF5 [8]), but, as an example, a NetCDF file loaded into HDFS as raw data cannot be processed by MapReduce applications. Consequently, scientific users who wish to use big-data computing for their applications often have to convert their data to a much more primitive data format (e.g., Comma Separated Values (CSV)), which causes substantial time and space overhead.

Second, existing big-data solutions cannot provide efficient support for the subset queries that are commonly used for processing scientific data. A naive solution that loads all data into memory and then processes the desired subset, requires large amounts of unnecessary processing tasks and unnecessary I/Os for loading the data. Solutions that prune the data based on either the structures or indexes of the data, work only for queries that require a subset on one particular dimension [9], [10].

This paper presents *Kaleido*, a solution to address the aforementioned limitations by enabling commonly used big-data systems to support the storage and analysis of scientific data stored using self-describing formats and optimize the processing of subset queries on such data. First, we extend HDFS to store scientific data in self-describing formats and allow big-data applications such as MapReduce jobs to parse and process the data. We also extend commonly used big-data query engines such as Hive to allow users to query the scientific data stored on HDFS.

Second, we optimize the performance of subset queries using a novel *dimension-aware layout* approach. Kaleido stores several layouts of the same scientific dataset, each one sorted by one of the dimensions of the data and optimized for queries that require subsets on that dimension. In this way, given a query requiring a subset on a specific dimension, Kaleido can always find data sorted by that dimension and process it using the minimum number of tasks.

We have developed a prototype of Kaleido based on Hadoop 2.5.2, Hive 1.2.0, and NetCDF3, and evaluated its performance using a 1TB geoscience dataset on a twenty-two nodes compute cluster. The results first show that Kaleido is able to substantially improve both performance and storage utilization. Compared to the traditional CSV approach and the related Parquet solution [11], it achieves a speedup of more than 657% and 31.81%, respectively, and reduces the storage usage by 413% and 29.1%, respectively. Second, the

results also show that, by using dimension-aware layouts, Kaleido is able to improve the performance for subset queries targeting all dimensions, and it outperforms the related works SciHadoop [9] and an indexing-based solution [10] by up to 373% and 423%, respectively.

The rest of this paper is organized as follows. Section 2 introduces the background and related works. Section 3 presents the design and implementation of Kaleido. Section 4 discusses the experimental evaluation results. Section 5 concludes the paper and outlines the future works.

II. Background

A. Need of Big-Data Systems for Processing Scientific Data

Many science domains are increasingly data driven, requiring processing of large amounts of simulation, experimental, and observational data for scientific discoveries. For example, the experimental data from Large Hadron Collider [12] may provide better answers to the fundamental questions in physics; to improve the predictability of hurricane tracking, a large amount of real time sensor data from various types of instruments need to be processed and incorporated into forecasting models. Therefore, big-data systems are also important platforms for these science domains by providing the necessary scalability and reliability for storing and processing big scientific data.

Typical big-data systems are built upon a highly scalable and available distributed storage system. For example, Google File System (GFS) [2] and its open-source version, Hadoop Distributed File System (HDFS) [3], provide fault tolerance while storing massive amounts of data on a large number of datanodes; while MapReduce [1] applications are executed in a data-parallel fashion on the datanodes where their data is stored. High-level data services can also be built upon such a big-data computing framework. For example, Hive [6] allows users to use common SQL queries on data stored in HDFS, which are automatically converted to MapReduce tasks to process the data.

When data is loaded in its native binary format into a big-data file system such as GFS and HDFS, it is split into large data blocks which are distributed across the datanodes in the system. Both the map and reduce phases of a MapReduce application can spawn large numbers of parallel tasks, depending on the size of the input, on the datanodes of a big-data system to process the data in parallel. To take advantage of data locality, which is the key to the performance of MapReduce applications, the map tasks are preferably scheduled onto the datanodes that have the data blocks for them to process locally, thereby in essence shipping computing to the data.

B. Need of Efficient Scientific Data Storage and Processing

Big-data systems are indeed increasingly used by users from different science domains. However, there are several key limitations of existing big-data solutions to provide efficient storage and processing of scientific data.

First, existing big-data systems do not support the data formats commonly used by scientific data. The de facto data

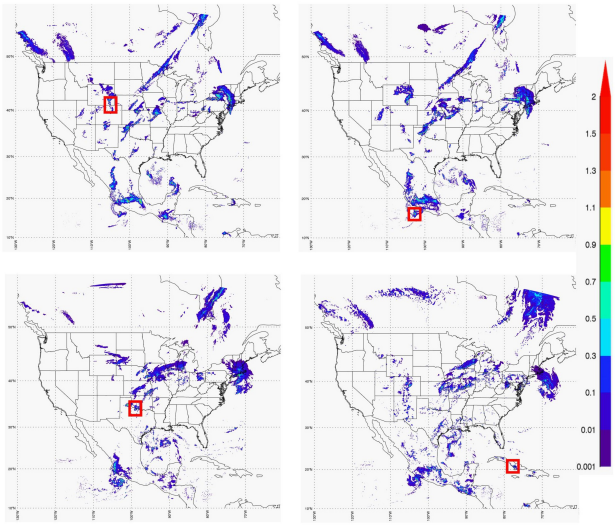

Fig. 1. 2D Illustration of the maximum rainfall on US continent, over different time frames. Each small red box encloses a user-defined region for further analysis of rainfall patterns.

formats used in many science domains are the self-describing formats such as NetCDF [7] and HDF [8]. They provide a concise and efficient way of storing array-oriented scientific data in binary. They are self-describing and machine-independent, which means that the description of data is not only well-defined in machine understandable way but also meaningful to human and conforms to relevant conventions [8]. For scientific applications, a wide variety of named dimensions and variables have been frequently used by broad scientific user communities. Existing conventions enable the cooperation and reuse of both standards and codes to transform, combine, analyze, and display specified fields of the data [7]. For example the settings of grids and physical units for climate and weather simulations vary among different models. Self-describing data formats facilitate the sharing of climate and weather data.

These formats are not natively supported by the widely used big-data systems. For example, one can load NetCDF files into HDFS as binary data, but MapReduce applications cannot interpret these data properly for processing. Consequently, users often resort to cumbersome approaches by using these systems for processing their data. First, they have to convert their data stored in self-describing format, e.g., NetCDF, to plain-text format using tools such as ncdump [13]. After the conversion, the file needs to be further translated to a multi-column table format such as CSV that is supported by a big-data system such as Hadoop. This approach is not only cumbersome to users but also incurs substantial time and space overhead.

Second, existing big-data solutions cannot provide efficient support of the subset queries that are commonly used for processing scientific data. For example, a geoscientist may want to find out the temperature characteristics of a certain geographic area at a certain time, which requires processing

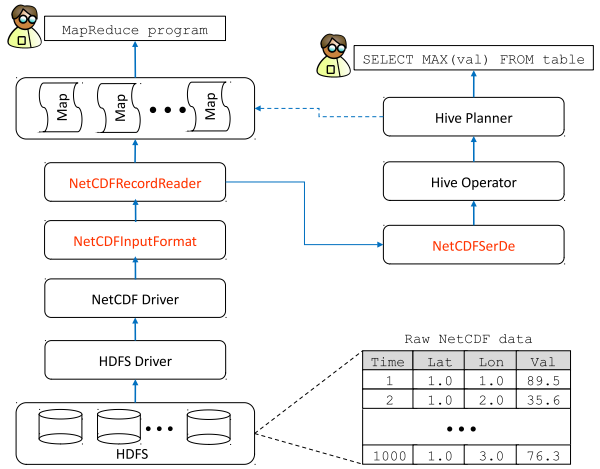

Fig. 2. Architecture of Kaleido for enabling scientific data storage and processing on big-data systems

the temperature values of a subset of the spatiotemporal data specified by the latitude and longitude ranges. For example, in Figure 1, the weather systems, which in this case are the rainfalls in different regions over different timestamps (represented by the colored areas in the maps), generally occupy only small portions of the whole dataset. Users who are interested only in rainfalls use subset queries to process only the relevant portions of data (represented by the red boxes in the maps), which is supposed to be much more efficient that processing the whole dataset [14].

A naive solution is to load all the data into memory and then process only the required data subset, but it requires large amounts of unnecessary map tasks and unnecessary I/Os for loading the data. Other solutions like SciHadoop [9], Hadoop-GIS [15], and SpatialHadoop [16] support loading only the necessary data subset into memory for processing. But a query may require a subset on any dimension of the data and the required data is often not sequentially stored on storage, so such solutions require loading large numbers of small pieces of data and thus do not perform well. Related work has also studied the use of indexing to reduce the data processing overhead for subset queries [10]. But indexes cannot be created for all the dimensions of the data; otherwise, the indexes would be as expensive as the original data. Hence, for queries that require subsets on the non-indexed dimensions, the processing does not get any improvement. For example, for spatial-temporal data, if only the time dimension is indexed, subset queries on latitude and/or longitude are still slow.

III. Design and Implementation

Kaleido is designed to address the aforementioned limitations of existing big-data solutions for scientific data. This section presents how it 1) enables widely used big-data systems to directly support self-describing data formats, and 2) enables these systems to support efficient execution of subset queries on multi-dimensional scientific data.

A. Enabling Hadoop to Support Scientific Data

Hadoop employs parallel map and reduce tasks to complete a large job. Each map task gets a split or multiple splits of the input data and performs the computation preferably on the node where the data locates. There is an *InputFormat* for each type of input file format, which handles the splitting of input data and then reading it through the *RecordReader*.

To support NetCDF data, Kaleido's design is to introduce new *NetCDFInputFormat* and *NetCDFRecordReader* APIs for processing the NetCDF data stored on HDFS. Figure 2 illustrates the overall architecture of our approach. To implement these new APIs, we exploit the standard NetCDF library to implement a *NetCDFDriver*. *NetCDFRecordReader* uses this *NetCDFDriver* to read the records from the NetCDF data, where each record corresponds to a row in the multi-dimensional array data, e.g., the temperatures of different locations (*latitude*, *longitude*) at a specific time.

To ensure good performance of the map tasks that process NetCDF data, *NetCDFInputFormat* needs to split the input data based on the physical distribution of the data so that each map task can get splits that are locally stored for processing. To achieve this, *NetCDFInputFormat* uses the *NetCDFDriver* to find out the offsets of the records and compare them to HDFS block boundaries. Kaleido also supports efficient processing of mutliple small files by packing as many as small NetCDF files into a single split, up to the HDFS block size boundary.

Finally, Kaleido also employs an optimization, which is rather than reading a variable at a time from the multi-dimensional array stored in NetCDF, *NetCDFRecordReader* uses a single read operation to retrieve a number of variables, e.g., an entire row of values from the multi-dimensional array. Experiments discussed in Section IV-D confirm that this optimization can speed up the data processing by up to 2.5 times.

B. Enabling Hive to Support Scientific Data

Hive is a data warehousing solution which is built on top of the Hadoop framework. Users can query data stored in HDFS using SQL-style declarative language. For example, a geoscientist can use a simple query such as `SELECT MAX(temperature) FROM table` to find out the highest temperature of a dataset without writing a single line of code.

As illustrated in Figure 2, every query submitted by users would be transferred into an execution plan by *Hive planner*. *Hive driver* receives this plan and submits corresponding map and reduce tasks to Hadoop, which use *InputFormat* and *SerDe* to read the data and apply the submitted query on it. *InputFormat* is responsible for reading data and passing key-value pairs to the map function. *SerDe* stands for SerializerDeserializer, which transforms the output of *RecordReader* into a column-oriented data format for Hive operators such as join and filter.

We have created a new *NetCDFSerde* to support the NetCDF data format, which converts every variable value from the multi-dimensional array stored in NetCDF into a row for Hive. However, as mentioned earlier, our *NetCDFRecordReader* produces a bulk of values at a time for better

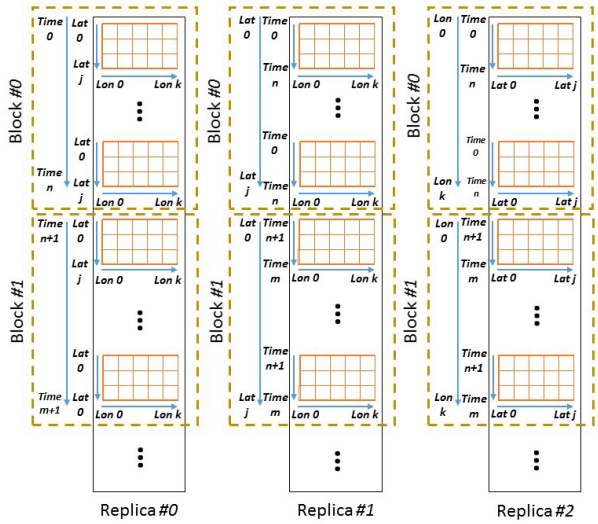

Fig. 3. Replicating single block on multiple nodes, using different layouts

performance. This behavior is not supported by the native *SerDe*, since it only accepts one row at a time. To support our optimization, we change the architecture of the map tasks created by Hive so that each map task is able to use *NetCDFRecordReader* to retrieve a bulk of values in one shot and feed them to *NetCDFSerde* row by row.

C. Enabling Efficient Subset Queries

Based on the aforementioned framework for directly storing and processing scientific data on big-data systems, Kaleido further improves the performance for commonly used subset queries using a novel *dimension-aware layout* technique. Users of scientific data often query only a subset of the data along one or more dimensions. For example, geoscientists often investigate the formation and track of a hurricane or tornado with comprehensive analysis on the areas near the center of the hurricane or tornado. The subset area is along longitudinal as well as latitudinal dimensions. Since the centers of hurricanes or tornadoes move, the subset areas have to be adaptively selected.

To address this problem, we propose a new *dimension-aware layout* scheme in Kaleido which stores several layouts of the same scientific dataset, each one sorted by one of the dimensions of the data. For example, for a spatiotemporal dataset, the original dataset is stored by the time dimension, which is illustrated by the time-oriented layout in Figure 3. Kaleido stores the dataset into three different layouts, each sorted by one of the *time*, *latitude*, and *longitude* dimensions, as illustrated in Figure 3. In this way, given a query requiring subset on a specific dimension, Kaleido can always find the best or near-best layout that is closely sorted by this dimension and use this layout to process the query efficiently.

Specifically, Kaleido's *NetCDFInputFormat* is able to decide which layout to use for processing a query based on its required subset, as illustrated in Figure 4. For example, if the query has a boundary on the latitude dimension such as *"latitude<120"*, *NetCDFInputFormat* will choose to read data from the latitude-based layout of the NetCDF data. It uses the boundaries specified by the subset to determine the necessary data splits to process for the query. Correspondingly, only the necessary map tasks will be launched to load and process these data. If the splits are small, *NetCDFInputFormat* will pack them into a fewer number of larger splits to further reduce the number of required map tasks for processing them and the associated overhead. Moreover, because the required data subset is sequentially stored on storage in this chosen layout, the data loading involves only sequential reads.

If a subset query specifies boundaries on more than one dimension of the data, e.g., *"latitude<60 and longitude>300"*, *NetCDFInputFormat* analyzes the required volumes of data using different layouts and choose the most efficient one to use. This process involves calculating the amount of data to be read from storage from using either one of the two layouts and then choosing the one that leads into less data to be fetched from storage.

To efficiently create the dimension-aware layouts for a dataset, Kaleido leverages the existing replication mechanism of HDFS. When data is loaded into HDFS, instead of creating identical copies of each block, Kaleido creates different layouts of the same data across multiple datanodes. In our modified HDFS data replication pipeline, the primary replica of a block receives the data from the original self-describing file, which is generally sorted by one of the dimensions of the data, e.g., time. It makes sure that the data in the block is aligned to the boundary of this dimension, e.g., it contains all the latitude and longitude values for the last time value in this block. This block of data is then pushed to the other replicas, each of which will apply the appropriate transformation to sort the data in one of the other dimensions and store it persistently on the local storage. Each replica also adds a header to its data, which contains all metadata information, such as the information about all variables available in the block, their sizes, and the ranges of their values. When a map task is assigned to process a block, its header helps the task parse the binary format data.

This layout creation method has several benefits. First, it minimizes the space overhead of our solution by leveraging the existing replicas that big-data systems use for fault tolerance. For example, HDFS employs by default three replicas for each block, which can be well utilized to store the time-, latitude-, and longitude-oriented layouts for spatial-temporal datasets. For datasets that have more dimensions than the default replication factor, users can choose to make additional replicas, if all dimensions are important for servicing subset queries, or skip the dimensions that are less important. Profiling can be performed on the commonly used subset queries to determine which three layouts would produce the most performance improvements. Second, the method does not compromise the reliability of data storage as the replicas of a block still store the same subset of data, although their layouts are different, and any replica can be recovered from the others in case of a replica failure.

In a Hadoop system, the *NameNode* is responsible for keeping track of data blocks and their replicas' locations. It is in contact with all *DataNodes*, in order to keep track of updated information about each single node. In Kaleido, *NameNode* is extended to also keep track of the data layout of each replica, for every single block. The *NetCDFInputFormat* determines which layout it should use to process a subset query by extracting the requested dimensions from the input query and then fetching the metadata of blocks that contain the target dimension from the NameNode. Map tasks are then deployed to process these blocks, which leverage the metadata in the blocks to understand the layout of data and used it properly.

Our dimension-aware layout scheme shares some similarities of the Trojan layout work [17] which uses different column layouts to store the data so that queries that are interested in different attributes of the data can find the attributes stored sequentially and be processed efficiently. In comparison, Kaleido's dimension-aware layouts are sorted by different dimensions of the multidimensional data. More importantly, Kaleido is optimized for scientific data processing. First, it supports the processing of self-describing data formats such as NetCDF commonly used by scientific data. Second, it leverages the metadata in NetCDF format to understand the information about the data's different dimensions. Third, in Kaleido, each replica of a data block is sorted by one of the major dimensions, which enables efficient, fine-grained data pruning and processing for supporting subset queries targeting any of these dimensions.

In native Hadoop, each block of data is replicated on three different nodes, so the map task assigned to process this block can be launched on any one of these nodes with local data as the input. In comparison, Kaleido replicates a data block using three different layouts, and it is often the case that only one of the nodes can provide the best layout to a map task that needs to process the data as required by a subset query. If the node happens to be busy, e.g., when all of its CPU cores are occupied by other tasks from the same or other jobs, the map task that requires the best layout from this node cannot be deployed.

Kaleido supports two different strategies to deal with this situation: (1) *delayed local execution* which delays the scheduling of the map task till there is a CPU core available on the node to execute the task and provide the best layout data to the task; and (2) *remote execution* which schedules the map task to a node that does not have the best layout data and where the task has to fetch the best layout data from a remote node across the network. Delayed scheduling allows the map task to use fast, local I/Os to access the data, whereas remote execution ensures immediate execution of the task with no delay. Our findings in the next section show that both strategies are much more efficient than native Hadoop because of the benefits of dimension-aware layouts.

IV. Evaluation

A. Setup

In this section, we compare Kaleido to:

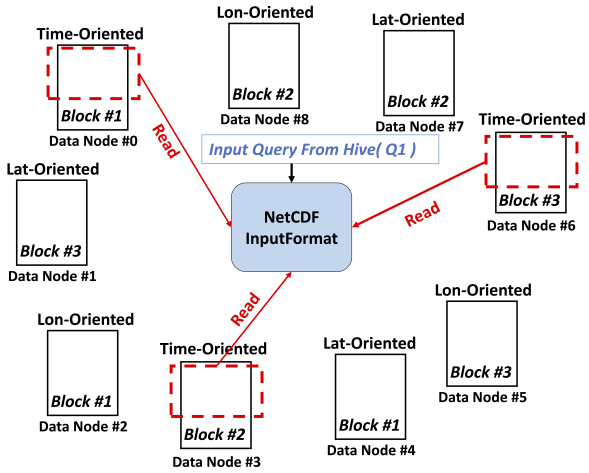

Fig. 4. Dimension-aware layouts of scientific data in big-data systems

TABLE I
Benchmark Queries

Q-Base	SELECT AVG(val) FROM netcdf
Q-Time 1	SELECT AVG(val) FROM netcdf WHERE 598290<time<1794870
Q-Time 2	SELECT AVG(val) FROM netcdf WHERE 1196580<time<1794870
Q-Lat 1	SELECT AVG(val) FROM netcdf WHERE 45<lat<135
Q-Lat 2	SELECT AVG(val) FROM netcdf WHERE 90<lat<135
Q-Lon 1	SELECT AVG(val) FROM netcdf WHERE 90<lon<270
Q-Lon 2	SELECT AVG(val) FROM netcdf WHERE 180<lon<270
Q-LatLon 1	SELECT AVG(val) FROM netcdf WHERE 90<lon<270 and lat<160
Q-LatLon 2	SELECT AVG(val) FROM netcdf WHERE 180<lon<270 and lat<160

- *Native approaches*, including native *Hadoop* and *Spark*, which provide a query interface using Hive and Spark SQL, respectively, and store NetCDF data using CSV and Parquet formats. Parquet is a data format that leverages compression and columnar data representation to reduce data size and avoid reading unnecessary columns or blocks of data [11].
- *SciHadoop* [9], a related project which supports a custom array query language using MapReduce jobs to process NetCDF-based scientific data stored on HDFS. SciHadoop also provides a *NoScan* optimization which prunes the data blocks involved in a query and reads only the necessary data required by the query.
- A related *indexing*-based approach [10] which uses indexing to improve the performance of processing scientific data on Hadoop. For spatiotemporal data, it creates (*variable*, *time*, *altitude*) indexes for 2D spatial grids so that only the necessary grids are loaded and processed for a given data query.

We consider a 1TB NetCDF3 dataset, which represents typical geoscience data containing a set of temperatures of

certain geolocations (*latitude* and *longitude*) at certain times. The range of latitudes in the data is from 0 to 180, and the range of longitude is from 0 to 360. We consider nine different types of queries, listed in Table I and briefly explained below:

- *Base query* does query on the entire dataset.
- *Query on time dimension*: **Q-Time 1** and **Q-Time 2** require time dimension subsets and involve half and quarter of the total data, respectively.
- *Query on latitude dimension*: **Q-Lat 1** and **Q-Lat 2** require latitude dimension subsets and involve half and quarter of the total data.
- *Query on longitude dimension*: **Q-Lon 1** and **Q-Lon 2** require longitude dimension subsets and involve half and quarter of the total data.
- *Query on multiple dimensions*: **Q-LatLon 1** and **Q-LatLon 2** require subsets bounded by two dimensions and involve half and quarter of the longitude dimension, respectively, and 89% of the latitude dimension.

The experiments were done on a cluster of 20 nodes, each with four eight-core 2.4 GHz Intel Xeon E5 CPUs, 64GB of RAM, and 1TB SAS disk, and interconnected by a 10Gbps Ethernet switch. All nodes run Ubuntu 14.04 Linux with the 3.16.0-x86_64 kernel and use EXT4 as the local file system. The evaluation uses Hadoop 2.5.2, Apache Hive 1.2.0, and Apache Spark 1.6.0. One node serves as the NameNode and the others as DataNodes. HDFS block size is defaulted to 128MB, and replication level is three. The NetCDFRecordReader read granularity, which is the total number of variables being read by NetCDFRecordReader at each attempt, is set to 131,072 bytes. Each Hadoop map task's resource usage is set to 1 CPU core and 1GB memory for Kaleido and other approaches except for Parquet which is 1 CPU core and 2GB memory. Parquet requires more memory and using 1GB per task will cause out-of-memory errors. Finally, each Spark worker's resource usage is also set to 1 CPU core and 1GB memory.

B. Query Performance

Figure 5 compares the runtime of the queries using different approaches. Kaleido outperforms the CSV approach by 657% on Hadoop and 420% on Spark, for querying the entire dataset using **Q-Base**. The reason is the difference in the data size—as we will discuss in subsequent sections, the CSV equivalent of the NetCDF data consumes 413% more disk space, which requires more map tasks and more time to process. For the rest of the queries that involve only a subset of the data, the difference is even greater as Kaleido processes only the necessary data, which leads to 3520% better performance.

Kaleido outperforms the Parquet approach by 31.8% on Hadoop and 10.7% on Spark, for querying the entire dataset using **Q-Base**. Kaleido is also faster for all the subset queries since it can leverage the correct layout, except for queries **Q-Time 1** and **Q-Time 2** which show that Parquet on Spark can do up to 22% better than Kaleido. We believe that this exception is because Spark jobs are generally faster than Hadoop jobs, and when ported to Spark, Kaleido should perform similarly to, if not better than, native Spark for time-subset queries as discussed in Section III. Overall, for queries bounded by the latitude dimension, Kaleido is faster than Parquet by up to 207% and 546% while processing half and quarter of data and when the longitude dimension is involved, Kaleido is faster by up to 263% and 672%.

Next, we compare Kaleido with the indexing-based approach [10], which first loads the whole data set into memory and then does the pruning in memory when executing the query. Because this approach has to load the whole dataset from the disk, which issues unnecessary I/Os, it cannot provide any performance improvement over **Q-Base**, except for queries **Q-Time 1** and **Q-Time 2**. As a result, for time-bounded queries Kaleido is 8.4% and 5.8% faster than the indexing-based approach for processing half and quarter of data, respectively; and for subset queries on the other dimensions, Kaleido is faster by up to 423%.

Finally, we compare Kaleido to SciHadoop, which prunes the data before loading it into the memory from disk, by reading only the interesting sections from blocks of the NetCDF data. As the original data is sorted by the time dimension, SciHadoop can benefit from sequential disk I/Os for queries that require subsets on the time dimension, and perform as well as Kaleido for **Q-Time 1** and **Q-Time 2**. But for subset queries on latitude and/or longitude dimensions, because the required data are scattered across the file, SciHadoop has to use many small, non-sequential I/Os to load the data. In order to make the comparison of SciHadoop, and Kaleido fair, we have enhanced SciHadoop to (1) use our solution's optimization for Hive and (2) use Hadoop's *Multi-Split* InputFormat which enables packing multiple small splits into one large split in order to reduce the number of map tasks, for better performance. Nonetheless, because Kaleido can always use sequential data access using its dimension-aware layouts, it still outperforms SciHadoop by up to 126% and 373% for queries that require half and quarter of data, respectively, on the latitude or longitude dimension, and 111% and 334% for queries that require subsetting on both dimensions.

C. Data Conversion Overhead

A significant source of the overhead of CSV and Parquet approaches is the time required to convert the NetCDF data into the CSV and Parquet formats. Without native support for self-describing data formats, scientific users need to first prepare their data in a format that is supported by Hadoop, and CSV and Parquet are two widely used ones.

To convert NetCDF data to CSV, one can use tools such as *ncdump* [13], but it is a single-threaded application and is quite slow. In addition, the amount of data that can be converted is limited by the available resources on the single node. To measure the best case conversion overhead, we leverage Hive and Kaleido's *NetCDFInputFormat* to convert NetCDF data to CSV using all the datanodes in the system directly on HDFS.

There is no existing tool that can convert NetCDF data to Parquet. Users often have to first convert NetCDF to CSV and then use Hive to convert CSV to Parquet. Again,

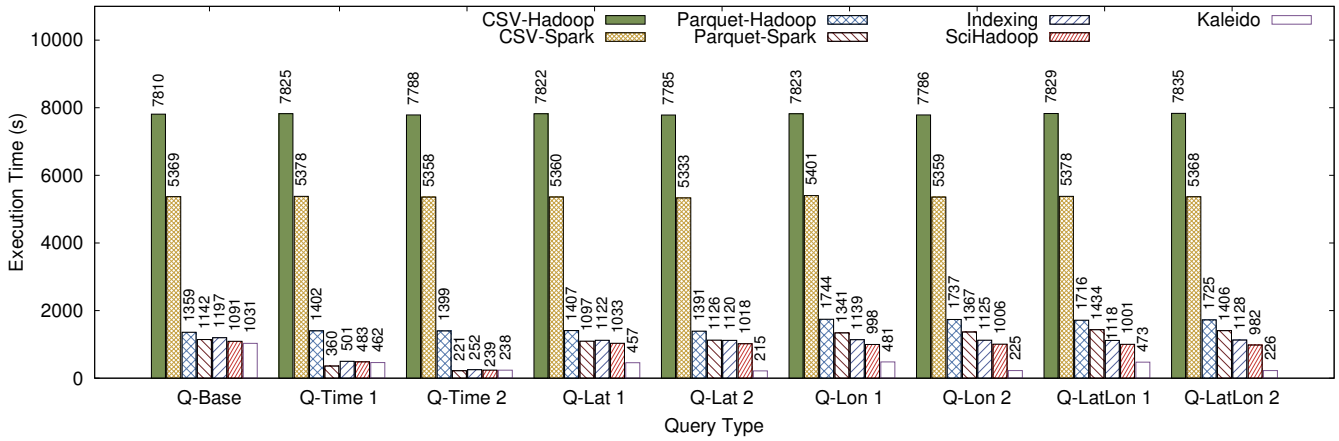

Fig. 5. Query performance

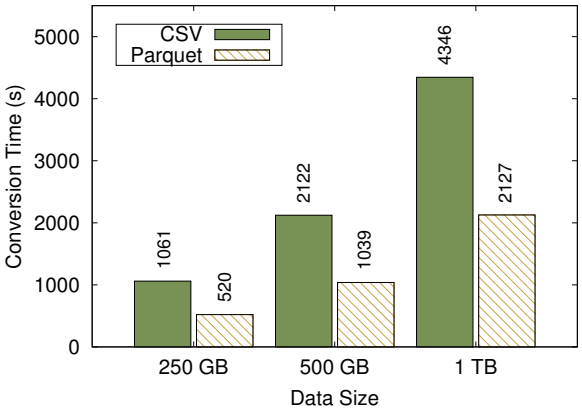

Fig. 6. Overhead for converting NetCDF data to CSV and Parquet

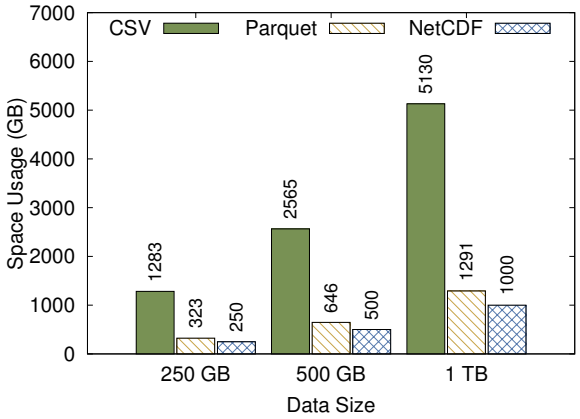

Fig. 7. Space overhead

to estimate the best case overhead, we leverage Hive and Kaleido's *NetCDFInputFormat* to directly convert NetCDF data to Parquet and parallelize the conversion, while in reality the conversion overhead has to include the time that it takes to convert from NetCDF to CSV too.

Figure 6 shows these best-case conversion overhead, which takes 73 and 36 minutes for converting 1TB of data to CSV and Parquet. In comparison, Kaleido avoids this overhead altogether because it can process raw NetCDF data directly from HDFS.

Storage space consumption on the big-data system is another aspect to compare the different approaches. Because Kaleido works directly on the raw NetCDF data, it incurs practically no space overhead—the only overhead is the metadata that it stores in each data block of data, which consumes about 0.0003% of each block' space. As it is shown in Figure 7, the CSV and Parquet approaches require about 413% and 29.1% more space, respectively, after converting the raw NetCDF data. Therefore, in comparison, Kaleido saves space usage substantially.

Although Kaleido works on raw NetCDF data directly, it stores the data in different layouts to support efficient subset

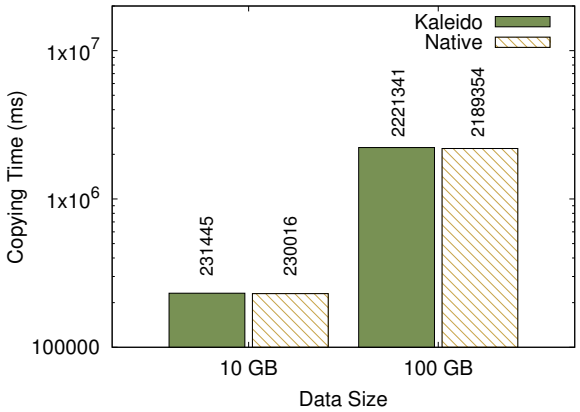

Fig. 8. Time for copying data into HDFS

queries on different dimensions of the data. As discussed in Section III-C, Kaleido modifies the data replication mechanism of HDFS so that when replicating a data block, each datanode that stores a replica of the block transposes the received data according to a different layout, sorted by one of the dimensions of the data. This data transformation incurs some overhead and is measured using an experiment. Figure 8 shows the

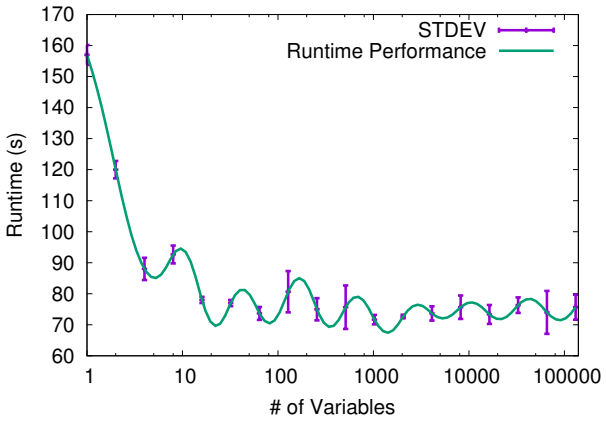

Fig. 9. *NetCDFRecordReader* read granularity

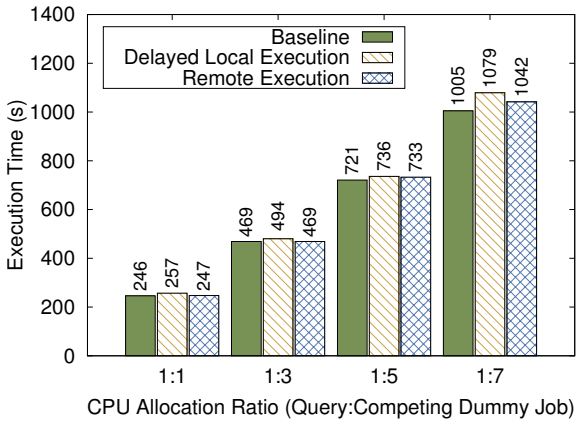

Fig. 10. The impact of data locality using dimension-aware layouts

time required for copying 10GB and 100GB of data from a local file system into the HDFS, using the original HDFS vs. Kaleido's modified version. The results show that Kaleido adds an overhead of less than 1.5% in data copying time compared to native HDFS.

D. NetCDFRecordReader Read Granularity

As discussed in Section III-A, one of the optimizations that Kaleido employs for efficient processing of scientific data is to read data variables from storage in batches, instead of one at a time, in its *NetCDFRecordReader* designed for NetCDF data. Allowing a map task to load data in bulk can exploit the storage's sequential access performance, but loading too much would cause map tasks to run out the datanodes's memory and cause them to crash and stop the MapReduce job from completion.

We used an experiment to quantify the impact of our *NetCDFRecordReader*'s read granularity. We used 6 datanodes loaded with a total of 7.3GB of NetCDF data. Figure 9 shows the execution time of the query **Q-Time 1** when *NetCDFRecordReader* is configured to fetch a different number of variables at a time. The *x*-axis is on log scale and the largest read granularity is 131,072 bytes, which is about 512KB.

When the read granularity is small, the overhead of reading a small number of bytes in each disk seek causes higher query execution time. As we increase the read granularity, the total query runtime drops quickly and it converges to around 73 seconds by reading at least 64 variables (about 256 bytes of data) at a time. These results confirm that the performance benefit of reading variables in batches in the Kaleido approach.

E. Impact of Data Locality

As discussed in Section III-C, with Kaleido's dimension-aware layouts, a map task may have to be delayed till the node that stores the best layout of its required data is available (*Delayed Local Scheduling*) or be executed on a different node and use remote I/Os to fetch the best layout data (*Remote Execution*). To evaluate this overhead, we compare to a baseline Kaleido implementation which does not use dimension-aware layouts and has three identical copies for each block of data. The baseline has all the other optimizations that Kaleido employs and **Q-Time 1** is used for the evaluation, so the results reveal only the overhead caused by delayed scheduling or remote I/Os. To show the worst-case overhead, we ran a dummy MapReduce job side by side with the query to further reduce the query's chance of getting nodes with the best layout data for its map tasks. This competing job contends for CPU cores on each node but does not perform any actual I/O. This experiment used only nine of the cluster nodes.

Figure 10 shows the query runtime with different CPU allocation between the query and the competing dummy job (1:1, 1:3, 1:5, and 1:7). As the CPU allocation to the query decreases from 1:1 to 1:7, the number of remote map tasks increases from 63 to 201 when using the *Remote Execution* strategy, whereas the amount of scheduling delay experienced by the map tasks increases when using the *Delayed Local Scheduling* strategy. The results show that the overhead of *Remote Execution* is less than 3.68% and the overhead of *Delayed Local Scheduling* is less than 7.36%. This overhead is because that compared to the baseline, Kaleido's job has a lower probability of using tasks with locally stored data. But, in both cases, the overhead is far outweighed by the performance improvement achieved from using dimension-aware layouts, as shown in the previous experiment. Moreover, comparing the two strategies, *Remote Execution* is slightly faster, because, with a relative fast network (10GigE in our setup), the overhead of reading data remotely is less than waiting for a local CPU to become available.

V. Discussions

A. Apache Spark Support

Apache Spark [18] is another widely used big-data platform. Many users prefer Spark over Hadoop for its advanced execution engine which enables iterative data flow and in-memory computation. Kaleido could also be well integrated with Spark and support the Spark ecosystem. Apache Spark uses the same *InputFormat* approach as Hadoop to read input data into RDDs, which is an immutable distributed data collection that

can be computed on different nodes of the cluster. As a result, *NetCDFInputFormat* can be readily ported to Spark. During the execution, *NetCDFInputFormat* can automatically iterate over the return values of *NetCDFRecordReader* and feed them into the lambda function of RDD. Note that the same optimization discussed in Section III-A, which is processing multiple rows at a time instead of one, should be applied to Spark too.

SparkSQL [19] is a framework like Hive, which let users query structured data inside Spark programs, using either SQL or the DataFrame API. In case of SparkSQL [19], the same optimization that Kaleido created for Hive could be applied to SparkSQL, in order to treat record reader return values as multiple rows instead of one. Moreover, Spark could also act directly as the execution engine for our enhanced Hive framework. In fact, Hive is able to convert the input query into a Spark execution plan, and let Spark execute the query. This approach allows our optimizations developed for Hive to be immediately avaiable to Spark queries.

B. NetCDF4 and HDF5 support

NetCDF4 and HDF5 are new versions of self-describing data formats. Compared to the NetCDF3 format considered in this paper, these new formats support hierarchical data structures, where data is stored by groups and each group can be a container of additional groups in a hierarchical manner. At the bottom level, each group stores a multi-dimensional array, but across groups in the hierarchy, the array formats can be different, which allows more flexible data storage.

Kaleido can also be readily extended to support NetCDF4 and HDF5 [8] data and achieve a similar performance as reported in this paper for NetCDF3. As the hierarchical format used by NetCDF4/HDF5 [8] allows different groups of data in a file to have different dimensions, Kaleido can store each group as a separate HDFS file (and then replicate it using different layouts). To make this transparent to applications, Kaleido can provide a virtual HDFS file to present the original dataset to the applications, and perform the virtual-to-physical HDFS file mappings internally. HDFS could be modified in a way to store each group inside the original NetCDF data as a separate physical file and perform the layout conversion per group. Following this approach, each group of data in the NetCDF4/HDF5 format could be mapped to a Hive table with a set of column attributes. As a result, a complex user query targeting different parts of distinct groups could be modified into a query targeting multiple tables and serviced by a join operation. Because data processing still happens on the physical HDFS files using layouts optimized for the queries, we can expect the same level of performance improvement when compared to conventional approaches.

VI. Related Work

There are several related efforts on enabling big-data platforms to support scientific data. SciHadoop [9] extended the Hadoop platform for NetCDF data, but it cannot deliver good performance to queries that require subsets along dimensions that are different from how the data is stored. Another related work [10] uses an offline indexing process to create indexes for spatial grids in the data, which is then used at the query execution time to read only the necessary data. But, similarly to SciHadoop, it cannot provide any improvement to queries that require subsets on the non-indexed dimensions. In comparison, Kaleido employs dimension-aware layouts to enable efficient support of subset queries on multiple dimensions, and our evaluation shows that it significantly outperforms these related solutions.

Trojan Data Layouts [17] is a related work that utilizes different layouts of data to optimize query performance and uses offline query analysis for selecting the set of candidate layouts. But it is designed for row-oriented datastore, and cannot handle the storage and processing of scientific data. In comparison, Kaleido embodies several optimizations specifically created for multidimensional data stored in self-describing formats.

Compared to the authors' own short paper [20], this paper has made substantial new contributions, particularly the efficient support for subset queries which are important to many scientific applications. The paper also includes a comprehensive experimental evaluation by comparing to the commonly used solutions, such as Apache Spark platform and Parquet data format, as well as the related works in the literature, such as SciHadoop and indexing-based approaches. The evaluation has provided compelling results that Kaleido outperforms all of these related solution in terms of both performance and resource utilization.

Finally, there are other existing systems designed specifically for scientific data processing, such as SciDB [21] which enables the execution of high-level queries on the scientific data stored on distributed storage. Such systems often have limited scope and use, and cannot be integrated with widely used big-data platforms such as Hadoop and Spark. In comparison, Kaleido enables scientific data to be efficiently processed on the widely used big-data platforms.

VII. Conclusions and Future Work

Kaleido presents an approach for enabling big-data systems to support efficient storage and processing of scientific data. It bridges an important gap between the self-describing data commonly used by scientists for data distribution and sharing and the big-data systems which are increasingly important for scientific productivity. Based on this approach, we have extended two important and widely used big-data platforms, Hadoop and Hive, to support scientific data with optimizations tailored to such data. With Kaleido, users can write MapReduce programs or use Hive queries to conveniently process NetCDF data with substantially better performance and ease of use than the existing methods. Our experiment results obtained from typical queries on a geoscience dataset confirm the improvements made by Kaleido in terms of query performance and storage space usage. As discussed in Section V, in our future works, we will extend Kaleido to support HDF5/NetCDF4 data formats, which are increasingly

used among data scientists, and apply the general approach of Kaleido to other important big-data systems such as Spark.

VIII. Acknowledgment

This research is sponsored by National Science Foundation awards CNS-1562837, CNS-1629888, CMMI-1610282, and IIS-1633381, and CAREER award CNS-1253944.

References

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

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[5] A. S. Foundation. Apache HBase. [Online]. Available: https://hbase.apache.org/

[6] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[7] R. K. Rew and G. P. Davis, "The unidata netcdf: Software for scientific data access," in *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, 1990, pp. 33–40.

[8] hdfgroup. Hdf5. [Online]. Available: https://www.hdfgroup.org/HDF5/doc/H5.intro.html

[9] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "SciHadoop: Array-based query processing in Hadoop," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 66.

[10] Z. Li, F. Hu, J. L. Schnase, D. Q. Duffy, T. Lee, M. K. Bowen, and C. Yang, "A spatiotemporal indexing approach for efficient processing of big array-based climate data with mapreduce," *International Journal of Geographical Information Science*, pp. 1–19, 2016.

[11] A. S. Foundation. Parquet. [Online]. Available: https://parquet.apache.org

[12] S. Dimopoulos and G. Landsberg, "Black holes at the large hadron collider," *Physical Review Letters*, vol. 87, no. 16, p. 161602, 2001.

[13] Unidata. ncdump. [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/docs/netcdf/NetCDF-Utilities.html

[14] X. Yang, S. Liu, K. Feng, S. Zhou, and X.-H. Sun, "Visualization and adaptive subsetting of earth science data in HDFS: A novel data analysis strategy with hadoop and spark," in *Big Data and Cloud Computing (BDCloud), 2016 IEEE International Conferences on*. IEEE, 2016, pp. 89–96.

[15] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop GIS: A high performance spatial data warehousing system over mapreduce," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, 2013.

[16] A. Eldawy and M. F. Mokbel, "A demonstration of SpatialHadoop: An efficient MapReduce framework for spatial data," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, pp. 1230–1233, 2013.

[17] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich, "Trojan data layouts: Right shoes for a running elephant," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 21.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMODthusoo2009hive thusoo2009hiveInternational Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[20] S. Biookaghazadeh, Y. Xu, S. Zhou, and M. Zhao, "Enabling scientific data storage and processing on big-data systems," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 1978–1984.

[21] P. G. Brown, "Overview of SciDB: Large scale array storage, processing and analysis," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 963–968.