SciHadoop: Array-based Query Processing in Hadoop

Joe B. Buck Noah Watkins Jeff LeFevre Kleoni Ioannidou
Carlos Maltzahn Neoklis Polyzotis Scott Brandt
UC Santa Cruz, Dept. of Computer Science
{buck,jayhawk,jlefevre,kleoni,carlosm,alkis,scott}@cs.ucsc.edu

ABSTRACT

Hadoop has become the de facto platform for large-scale data analysis in commercial applications, and increasingly so in scientific applications. However, Hadoop's byte stream data model causes inefficiencies when used to process scientific data that is commonly stored in highly-structured, array-based binary file formats resulting in limited scalability of Hadoop applications in science. We introduce SciHadoop, a Hadoop plugin allowing scientists to specify logical queries over array-based data models. SciHadoop executes queries as map/reduce programs defined over the logical data model. We describe the implementation of a SciHadoop prototype for NetCDF data sets and quantify the performance of five separate optimizations that address the following goals for several representative aggregate queries: reduce total data transfers, reduce remote reads, and reduce unnecessary reads. Two optimizations allow holistic aggregate queries to be evaluated opportunistically during the map phase; two additional optimizations intelligently partition input data to increase read locality, and one optimization avoids block scans by examining the data dependencies of an executing query to prune input partitions. Experiments involving a holistic function show run-time improvements of up to 8x, with drastic reductions of IO, both locally and over the network.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems; H.2.4 [**Systems**]: Query Processing; H.2.8 [**Database Applications**]: Scientific Databases

Keywords

Data intensive, scientific file-formats, map reduce, query optimization

1. INTRODUCTION

The volume of data generated by the scientific community has been increasing rapidly in recent years. Indeed, science today is all about big data, and making sense of all that data requires analysis tools that scale to meet the demands of scientists, and utilize resources efficiently.

One popular approach for tackling data-intensive large-scale analysis is to use the MapReduce framework. This framework is freely accessible, simple to program, provides built-in fault-tolerance, and is specifically designed to analyze large data sets with good scalability. MapReduce excels at tackling problems that are easily sub-divided, and many operations on scientific, array-based data have the property of being *conveniently parallel*. Thus, it would be beneficial if scientists could utilize MapReduce as a scalable, efficient data analysis tool for massive, raw scientific data sets. In this paper we present SciHadoop: a system for enhancing the performance of common analysis tasks (e.g. aggregate queries) over unmodified, array-based scientific data files using MapReduce as the execution substrate.

Many methods of scientific data analysis exist. One popular set of tools, the NetCDF Operators (NCO) [12], are designed to execute common queries over data stored in the NetCDF file format. Unfortunately the scalability of NCO is limited by its design which takes a centralized approach to processing data. With NCO, all computation is performed on a single node, and data is read serially from the file system. The SWAMP project [18] has been successful in parallelizing the execution of NCO queries by shipping subqueries to nodes within the storage system to reduce data transfers, but requires computations to be expressed using procedural scripts. Some steps have been taken to enable processing of NetCDF data with MapReduce, but this solution requires that data first be transformed into a text-based format [20]. While this may seem reasonable, the overhead of format transformation and additional data management costs are too great given the quantity and size of common scientific data sets. In order to provide the most value to practitioners, any approach using MapReduce should be capable of analyzing raw scientific data without any pre-processing.

Unfortunately a straightforward approach is difficult. Typically, the input to a MapReduce computation is a file stored as blocks (contiguous byte extents) in a distributed file system, and MapReduce processes each block of the input file in parallel. This approach works well because typical MapReduce computations are able to independently process arbitrary byte extents by sequentially reading (scanning) each input block from the file system. However, requests for data

expressed at the abstraction level of the logical, scientific data model do not always correspond to contiguous, low-level byte extents at the physical level. Thus, the difficulty of processing scientific data with MapReduce is manifested as a scalability limitation, and arises, as we will explore later in detail, from a disconnect between the logical view of data, and the physical layout of that data within a byte stream.

For example, consider a file format that serializes matrices onto a linear byte stream using row-major ordering (i.e. each row is physically stored one after the other). As the row size of a matrix becomes larger, cells that are near one another in the logical matrix but in separate rows (e.g. elements in a column) will become separated by a greater distance within the byte stream. Thus, a logical partitioning of an input file that ignores the corresponding physical layout may incur many remote reads when processed because the data values referenced by a partition may be located in more than one physical blocks. As the volume and frequency of remote reads increase, contention for network resources can be a limiting factor, and this limitation is a direct result of the unfortunate partitioning of a computation's input.

In general, the logical data view exposes no information about the physical layout and distribution of data. This disconnect is a road block for many types of optimizations that rely on resources defined at the physical level, such as caching and reducing duplicate reads, in addition to the remote read problem described in the previous paragraph. Our system, SciHadoop, addresses these issues by taking into account the physical location and layout of data during the logical partitioning of a computation's input, allowing for a variety of optimizations for common types of data analysis operations.

Our work makes the following technical contributions.

1. We identify performance limitations of a straightforward application of MapReduce to analyzing array-based scientific data. Our analysis employs an algebraic query language that allows us to reason about the deficiencies of such an straightforward approach in a principled fashion (Section 3).

2. To address the shortcomings of the straightforward approach we extend scientific file-format libraries to expose physical locality information which enables a more efficient solution to processing array-based scientific data with MapReduce (Section 4.2).

3. We propose three optimization techniques that take advantage of the semantics of scientific data queries in order to further reduce the cost of analysis: two optimizations for holistic aggregate functions, and one general optimization that eliminates traditional block scans in MapReduce.

4. We conduct and present a thorough experimental study of our solution using representative data and queries (Section 5 and Section 6).

2. MAPREDUCE AND SCIENTIFIC DATA

2.1 MapReduce

Since its introduction in 2004, MapReduce [4] has emerged as a go-to programming model for large-scale, data-intensive processing. The framework is popular because it allows computations to be easily expressed, offers built-in fault-tolerance, and is scalable to thousands of nodes [14].

Computations in MapReduce are expressed by defining two functions: *map* and *reduce*. Conceptually, a set of concurrently executing *map tasks* read, filter and group a set of partitioned input data. Next, the output of each map task is re-partitioned, and each new partition is routed to a single *reduce task* for final processing. Optionally, a "*combiner* function" can be utilized as a type of pre-reduce step, greatly reducing the data output at each map task location before it is transferred to the reducer. A full explanation of MapReduce is beyond the scope of this paper and we refer the reader to [4] for additional details. Next we present the data model assumed by MapReduce and details of the representative system environment that we target.

2.1.1 MapReduce Data Model and Storage

A data model specifies the structure of data and the set of operations available to access that data. MapReduce assumes a *byte stream* data model (i.e. the same format which most common file systems support today) and a set of operations similar to standard POSIX file operations. Generally, MapReduce is deployed on top of a distributed file system, and map and reduce tasks run on the same nodes that also host the file system. Files are composed of fixed-size blocks (byte extents) that are replicated and distributed among the nodes. Formally, a file is composed of a set of m blocks, $B = \{b_0, b_1, \ldots, b_{m-1}\}$, where each block b_i is associated with a set of hosts, H_i, which store a copy of b_i locally. The data contained in a block b_i are accessible indirectly through the file system interface, either remotely via a network connection, or locally on a host $h \in H_i$. Additionally, the MapReduce framework assumes that the underlying file system is capable of exposing the set of hosts, H_i, for any block b_i.

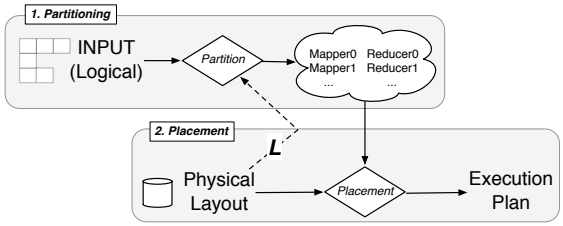


Figure 1: MapReduce processes logical partitions in map tasks and matches each map task with physical locations to form an execution plan. The line labeled **L** is a contribution of SciHadoop which utilizes physical layout knowledge during partitioning (see Section 4).

2.1.2 Partitioning and Placement

MapReduce scales in part because of its ability to intelligently coordinate the execution of map and reduce tasks. At a high-level this coordination consists of two phases, each illustrated in Figure 1. The first phase is concerned with the decomposition of the input into units of parallelization and defines a *partitioning strategy* that dictates how a computation's logical input is decomposed to be read by a set of map tasks. In the second phase a *placement policy* controls where each logical input partition will be processed within

the cluster by scheduling a map task to process a partition on a specific node (ideally where the majority of data represented by the logical split is physically present). The resulting execution plan is a specification used by MapReduce to run a computation and may be based on multiple optimization goals (e.g. minimize run-time and data transfers). These optimizations in turn depend on infrastructure characteristics, including policies related to the interaction with co-existing applications (e.g. load balancing).

2.1.3 An Example

A common optimization goal of MapReduce is to minimize the amount of network transfers which result from map tasks remotely reading blocks out of the distributed file system. Using knowledge of physical block distribution, MapReduce attempts to construct schedules in which a map task processes a block on a host that stores that block locally.

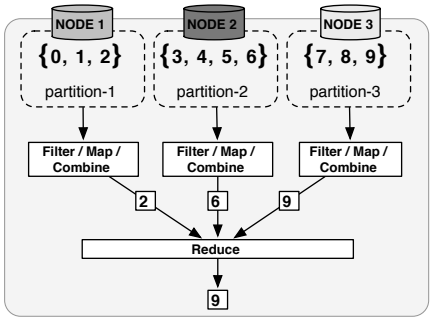


Figure 2: The execution of the *max* aggregate function in MapReduce. The map phase locates the local maximum in each partition and the reduce phase combines the results to produce a global maximum value.

Consider the scenario illustrated in Figure 2 showing the execution flow of a computation searching for the *maximum* value contained in a data set. The data set in this example is a standard byte stream containing the values $0 \ldots 9$, and is stored as three physical blocks, labeled $NODE1$, $NODE2$, and $NODE3$ according to the physical node the blocks are located on. The logical partitioning induced by the physical layout is given by the set notation used in each partition. The placement of each map task within the cluster is determined by the block location. The default placement policy attempts to schedule a map task on a node that stores locally the partition to be processed by the map task. Thus, *partition-1* is processed on the node with block on *NODE1*, and so on.

Algorithm 1 gives a simplified representation of the map function used to process a partition. The input to the map function is a filter, and a partition to process. First the map function extracts relevant array data based on the filter using *ExtractInput()*. A *Group* is assigned to the data representing a single function input. The repartitioned data (*ArrayData*) is then sent to the combine function using *Emit()*.

Algorithm 1: Map()

Input: Filter, Partition
(Group, ArrayData) ← ExtractInput(Filter, Partition)
Emit(Group, ArrayData)

The *combine* function is utilized to calculate per-group partial results from the output of each map function on each node before being sent to the reduce function. This can significantly reduce the size of the map function output sent across the network to reduce functions. The combine function, given by Algorithm 2, first calculates the local maximum and then emits this partial result to the reduce phase using the same *Group*. Finally, the output of all map/combine tasks is repartitioned by *Group*, and each group is sent to the same reduce function which calculates the final, global maximum value. In this particular example the reduce function is identical to the combine function given in Algorithm 2.

Algorithm 2: Combine() / Reduce()

Input: Group, ArrayData
Max ← Maximum(ArrayData)
Emit(Group, Max)

2.2 Scientific Data

Unlike the byte stream data model assumed by MapReduce, scientific data is commonly represented by a multi-dimensional, array-based data model [16]. In this section we present a simple version of such a data model and develop a query language used to express common data analysis tasks. Note that the language is not intended to be a contribution of this paper, but rather serves to expose the semantics of queries necessary to perform certain types of optimizations.

Storage devices today are built for the byte streams data model, thus high-level data models such as arrays must be translated onto low-level byte streams. This translation is performed by scientific file format libraries (e.g. NetCDF and HDF5) which implement high-level logical models on top of byte streams (illustrated in Figure 3b). These libraries serve two primary purposes: first, they present a high-level data model and interface (API) that is semantically aligned with a particular problem domain (e.g. n-dimensional simulation data), and second, they hide the nitty-gritty details of supporting cross-platform portable file formats. In Section 3 we will show how these features work against an efficient use of MapReduce for scientific data analysis.

2.2.1 The Array Data Model

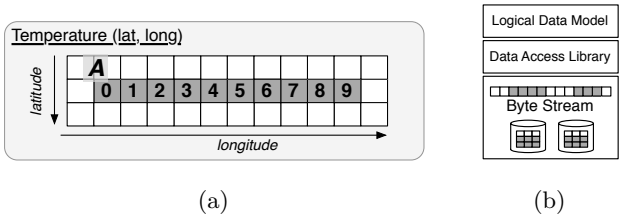
The array-based model used by SciHadoop is defined by two properties. First, the *shape* of an array is given by the length along each of its dimensions. For example, the array illustrated in Figure 3a has the shape 3×12, and the shaded *sub-array* has shape 1×10. Second, the *corner point* of an array defines that array's position within a larger space. In Figure 3a, the shaded sub-array has corner point $(1, 1)$. Arrays also have an associated data type that defines the format of information represented by the array, but in order to simplify presentation we assume a single integer value per cell in an array.[1].

In this paper we use the following notation to describe the shape and corner point of an n-dimensional array, say A:

[1]We have omitted values for the non-shaded region in order to simplify the discussion. The non-shaded regions can be interpreted as *null* values

$$S_A = (s_0, s_1, \ldots, s_{n-1}), s_i > 0$$
$$c_A = (c_0, c_1, \ldots, c_{n-1}), c_i \geq 0$$

where S_A and c_A are the shape and corner point of A, respectively. Thus, the shaded sub-array in Figure 3a is described as the tuple (S_A, c_A) where the $S_A = (1, 10)$, and $c_A = (1, 1)$. Note that throughout this paper we use the terms *array*, *sub-array*, and *slab* interchangeably.



(a) (b)

Figure 3: (a) A 2-dimensional data set. (b) Software stack used with scientific access libraries.

When scientific data is stored in arrays, labels are assigned to each dimension to give the data semantic meaning. For example, the array shown in Figure 3a depicts temperature readings associated with 2-dimensional coordinates specified by *latitude* and *longitude* values. Thus, the shaded sub-array may represent a specific geographic sub-region within the larger region represented by the entire data set A.

One common task when working with this type of data, especially when the content is initially unknown, is to evaluate ad-hoc aggregation queries against the data set. For example, consider the following query **Q1**: *"What is the maximum observed temperature in the shaded sub-array in Figure 3a?"*. We next present a simple query language used to express this type of query.

2.2.2 The Array Query Language

At a high-level, our array-based query language consists of functions that operate on arrays. Specifically, a function in our language takes a set of arrays as input and produces a new set of arrays as output (thus the language is closed under the logical data model). For example, the *max* function used in query **Q1** takes as input the shaded sub-array shown in Figure 3a, and produces as output a 1×1 array where the resulting single cell contains the value 9.

Formally, the language is defined as a 3-tuple (CS, SE, F). First, CS denotes a contiguous sub-array called the *constraining space* that limits the scope of the query. Second, the *slab extraction* function SE generates a set of sub-arrays, IS, referred to as the *input set*, where each sub-array $s \in IS$ is also contained in CS. Finally, a *query function* $f \in F$ is applied to the set IS yielding a *result set* R composed of output arrays $r \in R$. Thus, a function $f \in F$ takes the following form:

$$f : \{s_1, s_2, \ldots\} \rightarrow \{r_1, r_2, \ldots\}$$

Figure 4 illustrates the semantics of the language. The slab extraction function is applied to the constraining space producing the input set *IS*, and then the query function f is applied to *IS*, producing the result set, *RS*.

The process of slab extraction, including a more detailed look at how constraining spaces are used can be found in [3].

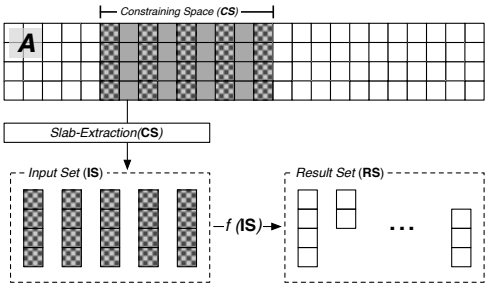


Figure 4: Illustration of the query language semantics. First, a subset of an array A, referred to as the *constraining space* is used to limit the scope of a query. The process of *slab extraction* forms the *input set*. Finally, the function f is applied to the *input set* to produce the *result set*.

2.3 Processing Scientific Data

To process scientific data in the MapReduce framework, mappers are forced to access data via scientific access libraries. This implies that mappers interact with data using the scientific data model and that partitioning occurs on the *logical level* of the scientific data model as opposed to the *physical level* of byte streams. Additionally, access libraries do not expose their data placement algorithms, and thus it is difficult to accurately predict the physical location of data addressed at the logical level. This poses a problem for achieving good locality during the placement phase of MapReduce computation scheduling as the logical-to-physical mapping is completely hidden. Next we present a straw man solution and show how it can perform poorly under common conditions.

3. PARTITIONING REVISITED

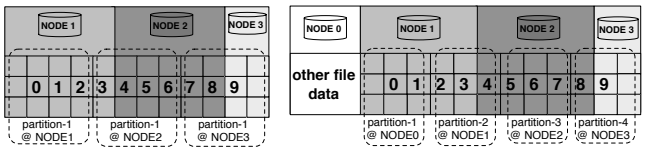
We now develop a representative solution to processing scientific data with MapReduce which will serve as a baseline to which SciHadoop optimizations can be compared. We refer to this solution as the *Baseline partitioning strategy* and it represents a reasonable approach to the problem that does not rely on low-level file format details and file system specifics. Unfortunately, as we will show, the Baseline approach can easily result in inefficient IO, resulting in long execution times.

Scientific access libraries use a black-box design in which file layout knowledge is obscured, thereby frustrating automatic optimizations that rely on such knowledge. Thus, users are left to manual construction of input partitions, making the quality of such partitions dependent on how much file layout information is known by the user. In essence, we model our Baseline partitioning strategy on what we believe to be reasonable assumptions about a scientist's awareness of the physical layout of a data format. Specifically, we assume that the block size of the underlying file system is available, and that high-level information regarding the serialization of the logical space onto the byte stream (e.g. column-major ordering) is known.

The Baseline partitioning strategy is to subdivide the logical input into a set of partitions (i.e. sub-arrays), one for each physical block of the input file. Consider Figure 5a which shows a 3×12 array stored within a file occupying

three physical blocks, located on nodes $NODE1$, $NODE2$, and $NODE3$. Using knowledge that the file format stores data in column-major order, a reasonable partitioning strategy is to form three equally sized partitions with each containing four columns. These partitions are shown in the figure using dashed frames, and are labeled as *partition-1,2,3*.

We assume a simple placement heuristic with round-robin matching of logical partitions to physical locations. The result of a round-robin placement is shown in Figure 5a using the notation *partition-x @ NODEy*. For example, *partition-2* is processed on node *NODE2*. To illustrate the difference in logical and physical partitions consider the example shown in Figure 5a. In the example, *partition-1* references 4 columns all contained in the block on $NODE1$, while *partition-3* references 2 columns from the block on $NODE2$, and 2 columns from the block on $NODE3$. In general, blocks are not stored at the same physical location within a cluster, thus a map task that processes *partition-3* must read at least half its data remotely.



(a) (b)

Figure 5: (a) Baseline partition strategy applied to a file containing a single array. (b) Baseline partition strategy applied to the same array in a different file containing additional arrays. The partitions become smaller because the same logical space is being mapped over a logical physical space.

For infrastructures with network bottlenecks, and when IO is dominated by map task reads, this misalignment of logical partitioning with physical layout can have a significant impact on performance. However, simple data sets with well-defined physical layouts (e.g. column-major ordering) that are stored with typical file block sizes will usually be slightly misaligned. As a result, the Baseline partitioning strategy will generally perform well. Unfortunately, achieving a good partitioning in the general case can be difficult as, in practice, few files contain a single variable.

Consider now the task of partitioning the same logical array shown in Figure 5a, but imagine a context in which the array is stored in a file containing additional data. This additional data could be header information such as data attributes or even other large data sets stored within the same file. A depiction of this new file is shown in Figure 5b. There are two things to notice. First, since there are now four blocks storing the larger file, but the logical space of the array is unchanged, the size of the partitions become smaller with the Baseline partitioning strategy. The second and most important difference, is that a typical round-robin placement of partitions to blocks can result in very poor data placement because of the shifting factor introduced by the additional file data. For instance, the data referenced by *partition-1* will be read entirely over the network when it is processed at node $N0$.

The choice of a baseline is somewhat arbitrary because—as just illustrated—the performance of such a baseline depends on the sophistication of the user and the complexity of the data set.

To illustrate the trade-off between data set complexity and user sophistication we perform two experiments, both of which evaluate a query over an identical logical array using the *Baseline partitioning strategy*. The array used in the first experiment is stored in a file by itself, and in the second experiment the array is stored in a larger file along-side other arrays. We measure the percentage of reads that are satisfied by disks local to map tasks, thereby evaluating the quality of the partitioning and placement strategies. We find that when a single array is stored in a file that the Baseline partitioning and placement achieves 71% read locality, but when multiple arrays are stored in a single file the read locality drops to 5%. This degradation in locality is due to the additional variable(s) that are present in the file, but not in the query, causing misalignment during the placement phase. In contrast, a SciHadoop optimization referred to as *chunking and grouping* (Section 4.2) can achieve 99% read locality for the multi-array data set. To provide a reasonably high bar for the performance of SciHadoop optimizations we chose to use the Baseline partitioning with the simple data set as Baseline for the rest of the paper.

In summary, this section has described why scientific data cannot be efficiently processed using standard scan-based approaches due to access library limitations, and how perturbations in input partitioning at the logical level can lead to poor performance. Thus, any efficient solution needs to overcome at least one of the two restrictions imposed by scientific access libraries. In the next section we describe a set of optimizations that use varying degrees of file-format knowledge to reduce the affects caused by a simple treatment of partitioning and placement.

4. THE DESIGN OF SCIHADOOP

In this section we present the design details of the SciHadoop system. SciHadoop is designed to accept queries expressed in the language described in Section 2.2, and leverages the semantics exposed by the language to implement a variety of query optimizations.

At a high-level SciHadoop modifies the standard task scheduler to function at the level of scientific data models, rather than low-level byte streams. In addition, map and reduce functions that implement query processing in SciHadoop are expressed entirely at the level of scientific data models. The ability of SciHadoop's task manager to act on data at the logical level enables a number of optimization techniques that we will now introduce according to SciHadoop's optimization goals:

Reducing data transfers. As MapReduce executes data is transferred to and from local disks, and HDFS and other tasks over the network. As we show in Section 6 these data transfers introduce overhead, including network contention and memory buffer swaps to disk. A common MapReduce optimization is to utilize a combiner function that performs local data reduction during the map phase. This optimization is easily utilized when using the *Baseline partition strategy*: for example, local maxima can be computed during the map phase and combined during the reduce phase to find a global maximum value. However, generating partial results using combiners is only possible when calculating algebraic aggregate functions, and holistic functions such as *median* must be evaluated over the entire input [11] by sending the

input to a reduce task.

In Section 4.1 we introduce two optimization techniques relevant to holistic function evaluation called *Holistic Combiner* and *Holistic-aware Partitioning*. The first optimization opportunistically evaluates holistic aggregate functions in combiners when it can be determined that an entire input is available in a single map task input, and the second optimization is used to make adjustments to logical partitions at scheduling time that increase the probability of holistic function inputs falling into single partitions.

Reducing remote reads. As described in Section 2.1 remote reads can be reduced by scheduling map tasks to process data blocks on nodes that store those blocks locally. However, Section 3 demonstrates that it is difficult to make scheduling decisions at the logical level of a scientific data model because they physical data layout is hidden. SciHadoop implements two techniques for reducing the volume of remote reads where simple round-robin scheduling heuristics would perform poorly.

The first optimization, *physical-to-logical translation*, generates logical partitions that reference exactly the data contained within a single block. While this optimization minimizes remote reads, the technique may not always be feasible to implement, and thus a more general method is needed. The second optimization, *chunking and grouping*, decomposes a logical space into a set of small chunks and groups the chunks according tho their primary position within the byte stream achieving increased locality of reference over simpler round-robin assignment.

Reducing unnecessary reads. Typical MapReduce computations that process unstructured data, such as log-processing, must scan blocks on disk and filter out unnecessary data in memory. However, the highly structured nature of scientific data enables SciHadoop to avoid block scans by constructing requests at the logical level that contain exactly the data necessary to complete a query. The technique, referred to simply as *NoScan*, prunes input partitions to eliminate unnecessary segments of the logical space, reducing the total amount of data read during query execution.

4.1 Reducing Data Transfers

A holistic aggregate function has the property that it cannot be computed by combining multiple, partial results. For example, consider the task of calculating the *median* value from the range $0 \ldots 4$ using MapReduce. Figure 6a shows the execution flow of this query. Since neither of the two partitions shown contain the entire range of $0 \ldots 4$, each partition must be sent to the reduce function for evaluation. Here we introduce two techniques to help reduce remote data transfer for holistic function evaluation that use semantics exposed by the logical data model and query language.

Holistic Combiner. In SciHadoop partial aggregate values for non-holistic functions are computed, when possible, during the map phase using a combiner. However, holistic functions must be processed entirely by a reducer because the input to a holistic function may be present in multiple partitions. SciHadoop opportunistically evaluates holistic functions during the combine phase when the entire input to the holistic function is present in one or more partitions being evaluated on a single node. In this way the Holistic Combiner optimization extends the data reduction benefits of using a combiner to holistic functions when possible.

Holistic-aware Partitioning. While the holistic combiner can evaluate a holistic function when the entire input is contained in the partitions on a single node, the initial partitioning of the logical input space is entirely unaware of the query being processed, and thus the ability for the combiner to provide a benefit is probabilistic. To account for small misalignments that prevent the combiner from being used for holistic function queries, SciHadoop makes small adjustments to partitions to increase the likelihood that holistic functions can be evaluated using the holistic combiner optimization. Figure 6b shows how the first partition is adjusted to include the entire holistic function input.

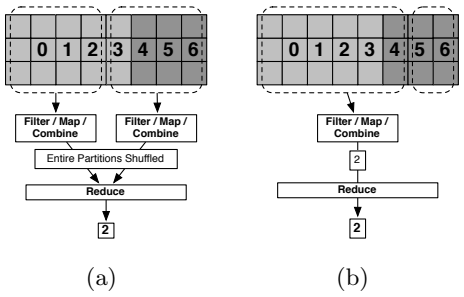


Figure 6: Sub-figure (a) shows two entire partitions being sent to a reduce task to evaluate a holistic function over an input that spans partitions. Sub-figure (b) shows a repartitioning that allows the holistic function to be applied during the map/combine phase.

4.2 Reducing Remote Reads

In this section we introduce two low-level techniques for producing partitions that reduce remote reads. One technique, *physical-to-logical translation*, produces optimal partitions that reference exactly the data contained in a physical block, but may not always be feasible to implement. The second approach, referred to as *chunking and grouping*, is a more general technique that trades-off partition optimality for ease-of-use. Both techniques are query-independent, and focus solely on logical partition creation that exhibits increased physical locality.

Physical-to-Logical Translation. This technique directly translates the extent represented by a physical block into its equivalent logical representation. Figure 7a illustrates how optimal, from a data locality standpoint, partitions are created. At the top of the figure a process is shown that uses file metadata to generate a logical representation of the data contained in a physical block. Since each block is directly converted into its logical representation partitions are precisely aligned with physical block boundaries. Therefore placement is trivial: a partition is matched with the block from which it was generated.

Despite the precision of this technique, it can be difficult to implement for complex file formats. Therefore we introduce a more general purpose technique for constructing partitions.

Chunking and Grouping. The second technique SciHadoop can use to reduce remote reads is referred to as *chunking and grouping*. This technique decomposes the input into many fixed-size units called chunks from which a random sampling of byte stream locations is taken using

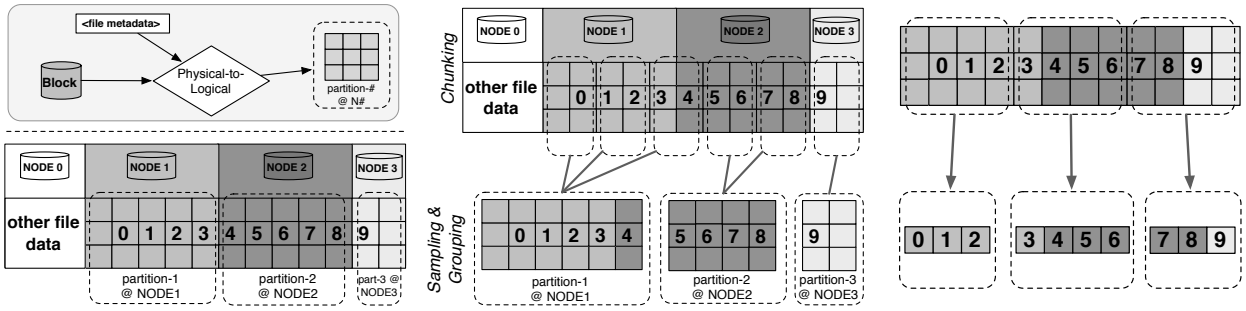


Figure 7: (a) Direct physical-to-logical translation (Section 4.2). (b) Chunking and grouping (Section 4.2). (c) The *NoScan* optimization (Section 4.3.)

extensions to scientific access libraries. The sampling technique allows SciHadoop to then group chunks into flexibly defined partitions with increased locality of reference.

Figure 7b illustrates how chunking and grouping are used to create partitions. At the top of the figure fixed-size chunks are grouped together into partitions such that each partition references primarily data within the same physical block. This optimization requires extensions to file format libraries to allow chunks to be associated with regions of the byte stream, providing the ability to group based on data locality. Next we describe chunking and grouping in more detail.

Chunking. The first step is the decomposition of input at the logical level into a set of chunks (i.e. fixed-size, contiguous, non-overlapping sub-arrays). The set of chunks that cover the entire input space is given by K, where each chunk $k \in K$ is determined by a *chunking strategy*.

There are trade-offs in choosing a chunking strategy. For example, a small chunk size provides a finer granularity at which partitions can be created, but results in more overhead in managing many small chunks. A detailed discussion of chunking strategy is beyond the scope of this paper, but we provide the parameters for our experiments in Section 6.

Grouping. Grouping is the process by which chunks in the set K are combined to form input partitions. The goal of grouping is to form partitions that reference data located in the fewest number of physical blocks. Thus, the problem of creating input partitions is equivalent to grouping chunks $k \in K$ by block, such that each group maximizes the amount of data referenced in the block that the group is associated with. The resulting partitions are what we refer to as the logical-to-physical mapping, defined by the set $LTP = \{(b_0, P_0), (b_1, P_1), \ldots, (b_m, P_m)\}$, which associates with a physical block b_i, a logical partition P_i composed of one or more chunks.

Sampling. The construction of LTP is based on the examination of a randomly sampled set of cells taken from a chunk. First, a set of n cells is selected from the logical space represented by a chunk k using a uniform random distribution. Next, each cell in the sample is translated into its associated physical location on the byte stream using a special function, *getOffset(cell)*, introduced as a SciHadoop extension to scientific libraries. The return value of *getOffset* is the byte stream offset of the cell's logical coordinate.

Finally, a histogram is constructed that gives the frequency of sampled points for a chunk that fall into a given block. The block b_i with the highest frequency is chosen and the chunk being considered is added to the partition P_i associated with that block.

Sampling is the dominate cost of chunking and grouping. For example, in our evaluation section we use a sampling ratio of 0.0001 on a file containing 35 billion logical coordinates resulting in the sampling operation being performed approximately 3.5 million times. Microbenchmarks show that our implementation of sampling for NetCDF-3 files can achieve 600,000 samples per second.

Next we present a concrete example of the functioning of this technique using the query **Q1** from Section 2.2.

4.2.1 Example

First we consider the set of chunks K, consisting of the 6, 3×2 sub-arrays, shown at the top of Figure 7b. We refer to these chunks by their position in the figure (i.e. $1 \ldots 6$).

Next, for each chunk $k \in K$ we perform a random sampling. For chunks *1, 2, 4, 5*, and *6*, it is clear that any random sampling will definitively associate the chunk with a given block because each chunk references data contained within exactly one block. However, a sampling of chunk *3* may result in sample points that fall in either block on $NODE1$ or block on $NODE2$. Thus no matter the location at which chunk *3* is processed, remote data access will be required. Uniform random sampling is an effective way to minimize the amount of remote reads for *3* by choosing a block with a majority of the data. The result of chunking and sampling is shown as the partitions illustrated at the bottom of Figure 7b. The final LTP mapping is given as:

$$LTP(block_1) \rightarrow \{chunk_1, chunk_2, chunk_3\}$$
$$LTP(block_2) \rightarrow \{chunk_4, chunk_5\}$$
$$LTP(block_3) \rightarrow \{chunk_6\}$$

The resulting LTP mapping can now be utilized by MapReduce to schedule the processing of partitions on the nodes associated with each block in order to reduce the amount of remote reads resulting from query execution.

4.3 Reducing Unnecessary Reads

We now present an optimization referred to as *NoScan* that reduces the amount of unnecessary reads to a given partition. The key motivation is that the construction of LTP does not consider a given query's data requirements. That is, LTP represents a partitioning of an entire file, even

when a query may require only a subset of the total input. A placement using LTP alone will thus read entire partitions from the file system, only afterwards considering the data requirements of the query. SciHadoop optimizes the LTP mapping by pruning each partition in LTP to include only the data required by a query. We refer to this new mapping as LTP'. In order to form this new mapping, an optimization process must consider a query's data requirements.

We form LTP' using the *input set* of a query, IS, which represents a query's data requirements, and informally define the intersection $P_i \cap IS$ to be the subset of a partition P_i required by the query. Thus, for any given block in LTP, the associated logical partition P_i can be pruned to reference only data required by the query:

$$LTP' = \{(b_i, P_i') \mid P_i' = P_i \cap IS, \forall (b_i, P_i) \in LTP\}$$

Example (continued from Section 4.2). Using knowledge of the data requirements of query **Q1**, the LTP that was constructed previously in Section 4.2.1 can be trimmed such that it contains exactly the components of the logical model that are required to complete the query. Figure 7c illustrates this trimming process, in which each of the partitions are reduced to only the sub-arrays needed by the query **Q1**.

5. IMPLEMENTATION

SciHadoop is implemented as a plugin to the Hadoop MapReduce framework *v0.21*. It supports the NetCDF-3 file format using the NetCDF-Java library and is currently limited to read-only workloads. The read-only limitation is a consequence of HDFS's append-only file system interface. While writes could be buffered and then appended, a more complete, scalable solution to supporting read/write workloads is left for future work.

Hadoop plugin. SciHadoop array query processing is implemented as a standard MapReduce computation. The initialization phase of SciHadoop is handled by the Hadoop *FileInputFormat* class that partitions the logical input and assigns each partition to a physical host. A partition in SciHadoop is represented by the Hadoop *InputSplit* class that in SciHadoop represents a set of sub-arrays. Each *InputSplit* (i.e. partition) is scheduled transparently by Hadoop using high-level policies such as load balancing. An *InputSplit* is processed at a node by the *RecordReader* class which loads the associated logical partition from the underlying file system using the NetCDF-Java library. Finally, the logical partition is sent from the *RecordReader* to the *map* function at which time the processing resembles the MapReduce computation flow described in Section 2.1.

Integration with NetCDF. In order to support logical-to-physical translation, as described in Section 4.2, we have extended the NetCDF-Java library to expose mapping information. The translation mechanism is exposed via a function that takes as input a set of logical coordinates, and produces a set of physical byte stream offsets corresponding to each coordinate. Internally this is implemented as a simulation of the actual request that NetCDF would perform, but does not complete the read to the underlying file system, and instead responds with the offset of the request.

Additional software layers also had to be created to allow NetCDF-Java to interoperate with Hadoop's underlying distributed file system, HDFS. NetCDF-Java is tightly integrated with the standard POSIX interface to reading files. Unfortunately, HDFS (the distributed file system built for Hadoop), does not expose its interface via standard POSIX system calls. To accommodate this API mismatch we built an HDFS connector which translates POSIX calls issued by NetCDF-Java into calls to the HDFS library.

6. EVALUATION

Experimental Setup. Experiments are performed on a cluster of 31 nodes each with 2x 1.8 GHz Opterons, 8GB RAM, 4x 250 GB Seagate SATA drives, and Gigabit Ethernet running Ubuntu 10.10. Nodes are networked using an Extreme Networks' Summit 400 48-t switch. Hadoop version 0.21 is deployed on 1 master and 30 worker nodes (HDFS and MapReduce). HDFS is configured to use 3 SATA drives for data while the fourth SATA drive was used for the OS and temporary storage. Default HDFS configurations, including 64 MB blocks and 3x replication, are used.

In each experiment a map task is created for each block of the input file, thus the number of map tasks is determined by a the size of the input file. The number of reduce tasks is influenced by common MapReduce balancing heuristics and rules-of-thumb.

6.1 Methodology

NetCDF Dataset. We use a data set modeled after a schema from UCAR[2] in our experiments. The data set contains a single variable measuring air pressure, as 32-bit values, defined as a time series over the dimensions *time*(expressed in days), *latitude* and *longitude* (expressed in half and full degree increments, respectively), and *elevation* with lengths 5475, 360, 360, 50, respectively. The total size of this data set is 132 GB.

Metrics. We evaluate our system using 4 metrics: *total run-time*, *CPU utilization*, *local HDFS reads*, and *temporary data*. Next we discuss the latter two metrics in more detail.

The metric *local HDFS reads* measures the fraction of data read from HDFS which was locally available on the node reading the data (e.g. map task). The metric *temporary data* is a measurement of the volume of data written to temporary storage by the MapReduce framework, expressed in bytes. Temporary storage is used during IO spills by map and combine tasks as buffer space becomes scarce, and for an external merge-sort of reduce task input.

Query 1. "*Apply the median function to the specified time-range, over sets of data contained in two adjacent days, in a given area (defined by a lat x lon box) within an elevation range*" can be written declaratively, as seen in Figure 8a.

```
apply(median, pressure,
   CS = (
      corner = (547, 0, 0 ,0)
      shape = (4380, 360, 360, 50),
   ),
   SE = (2, 36, 36, 10),
)
```

(a)

```
regrid(average, pressure,
   CS = (
      corner = (547, 0, 0 ,0)
      shape = (4380, 360, 360, 50),
   ),
   SE = (7, 2, 1, 1),
)
```

(b)

Figure 8: Function declaration for (a) Query 1 and (b) Query 2

[2]http://www.ucar.edu

Test Name	Local Read (%)	Temp Data (GB)	CPU Util (%)	Run Time (Min)	Time σ (%)
First 4 use no Holistic Combiner					
Baseline	9.3	2,586	34.7	129	7
Baseline +NoScan	9.2	2,588	34.3	118	3
ChkGroup	80	2,608	24.3	145	5
PhysToLog	88	2,588	29.9	138	5
Next 4 use Holistic Combiner with Baseline					
Baseline	9.5	107	79.1	31	2
NoScan	9.5	107	80.7	27	1
+NoScan *+HaPart*	8.8	107	81.3	27	7
+HaPart	8.6	107	79.3	31	0.8
Next 3 use Holistic Combiner with Local-Read Optimizations					
ChkGroup +HaPart +NoScan	70.7	116	84.7	25	0.4
ChkGroup +NoScan	79.3	188	83.1	26	1
PhysToLog +NoScan	88.1	196	82.8	27	6

Table 1: **Overview of Query 1** Runtimes of tests without the Holistic Combiner optimization are dominated by writes to temporary storage. All other runs are bound by CPU (with `iowait` times $< 1\%$). Runtimes are averages from 10 executions. Abbreviations: *ChkGroup*: Chunking & Grouping, *PhysToLog*: Physical-to-Logical, *HaPart*: Holistic-aware Partitioning.

This query can be read as *apply the median function to the pressure variable using constraining space CS and using the slab extraction function with the shape SE*. Since the constraining shape only filters on the time dimension, the effect is that data from dates outside the query are omitted. The slab extraction function operates on all four dimensions: it is grouping data from adjacent time values, within that grouping it is processing data in groups of 36 x 36 (referring to the lat and lon dimensions) and within that slab, it is processing at a range of 10 elements in the elevation dimension.

Query 2. The *regrid* operation is a common operation in scientific data that is used to alter the coordinate system of a data set. For example, the units of a data set may be changed so that it is directly comparable to another data set (e.g. polar to Cartesian). We use the following regrid operation, "*Regrid the pressure variable along time and latitude dimensions using units weeks and full degrees, respectively. Interpolate using average*", Figure 8b.

6.2 Results

An overview of results from running *Query 1* is given in Table 1. We discuss these in terms of our optimization goals.

Reducing Data Volume. The Holistic Combiner and Holistic-aware Partitioning have a significant impact on runtime; roughly an order of magnitude. As Table 1 shows, tests using the Holistic Combiner optimization are largely CPU-bound while others are not, implying that those tests are waiting on IO. We were admittedly surprised by the extent of the performance impact until we realized that data traversing the system is transferred over the network but, more importantly, potentially written to disk repeatedly.

A large volume of data causes a significant increase in temporary storage writes due to buffer spills and external sorting, which can occur at each step of a MapReduce computation. In our setup a single disk is used to store temporary data (HDFS stripes data over 3 SATA drives). This can lead to an IO bottleneck for temporary data. In the case of Query 1, successfully applying the combiner at the map node rather than at the reducer results in a reduction of 25,920 data values, and their associated metadata, to a single value. This dramatic reduction in data results in a commensurate reduction in intermediate data written, shown in the right-most column of Table 1, and translates into significant reductions in query execution time.

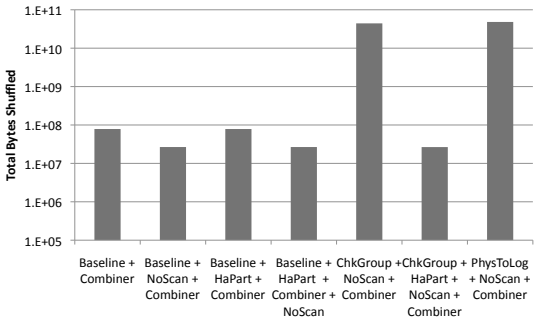


Figure 9: Bytes shuffled in tests with Holistic Combiner (note the y-axis log scale).

Figure 9 shows the number of bytes transferred between map and reduce tasks. Generally, the tests that use Holistic-aware Partitioning reduce data transfer by almost three orders of magnitude compared to Holistic Combiner tests without Holistic-aware partitioning. The exceptions to this are Baseline and Baseline with NoScan tests which show a similar amount of data transferred to the Baseline tests with Holistic-aware partitioning. Log data taken during our experiments reveals that this is the result of an incidental alignment between the shape of the data, the query shape, and the Hadoop configuration used on our cluster.

Reducing Remote Reads. A potential conflict exists between partitioning strategies that reduce remote reads and partitioning strategies that reduce data transfers: the former partitions along physical data location while the latter partitions according to complete input ranges of holistic queries. Consider the last two Chunking & Grouping tests of which one of them uses Holistic-aware partitioning (see Table 1). Given that our experimental setup is never limited by HDFS reads and these two tests are CPU-bound during most of their runtime, sacrificing local reads (8.5% lower local read fraction) in favor of increasing combiner efficacy (62% less temporary storage writes) results in one minute shorter runtime (4% speedup).

As can be seen in Table 1, the Chunking & Grouping partitioning strategy dramatically improves the read locality for map processes, with experiments going from 9% to 80% local reads. The Physical-to-Logical method achieves even better data locality, achieving close to 90% local reads. Recall that all our tests use the default Hadoop scheduler, which will give tasks to non-local nodes rather than let them sit idle, so 100% local reads is infeasible.

Reducing Unnecessary Reads. The NoScan optimization reduces the total amount of data read, based on the query being executed (see Section 4.3). In our tests, we had 80% selectivity and the results show that 80% of the data was being read (105 GB vs 132 GB) when NoScan was

Test Name	Local Read %	Temp Data (GB)	Reduce Output (MB)	Run Time (min)
Baseline +*NoScan* +Comb				
Holistic	9.5	107	0.1	27
Regrid	3	569	7,735	63
ChkGroup +*NoScan* +Comb				
Holistic	79	188	0.1	26
Regrid	84	498	7,735	55
PhysToLog +*NoScan* +Comb				
Holistic	88	196	0.1	27
Regrid	93	498	7,735	56
Serial Program				
Regrid	100	NA	7,735	3,130

Table 2: **Impact of Data Volumes on Runtime** For comparable queries, runtime is dictated by the volume of data generated. The serial regrid program ran on a single node and accessed the data from local storage.

Test Name	Local Read %	Initial Read (GB)	Temp Data (GB)	Run Time (min)	Time σ %
All tests use a Combiner					
Baseline	3	132	569	66	2
Baseline +*NoScan*	3	106	569	63	2
ChkGroup +*NoScan*	84	106	498	55	3
PhysToLog +*NoScan*	93	106	498	56	3

Table 3: **Runtimes for Query 2** All experiments produced 7.7 GB of reducer output. Values are averages over ten runs.

turned on. The impact on run time can be seen in Table 1: Baseline + NoScan ran 17 minutes faster than Baseline without the combiner and a minute faster with it.

Non-Holistic Queries. Query 2 was executed with all the combinations from Table 1 that include a combiner. The results can be seen in Table 3. (HaPart query results were omitted as that optimization makes little sense for non-holistic functions).

Comparing the results from both queries, the interplay between the different salient metrics and their effects on runtime become evident. Table 2 shows select experiments from both queries as well as an execution of a single-threaded program that completes Query 2 serially by reading data locally. As the amount of intermediate data written (Temp Data) and reduce output increases, so does the observed runtime. This is expected, as increases in both translate into more IO and more data that must be processed in the combine and reduce functions.

7. RELATED WORK

Integrating MapReduce systems with scientific data is not a novel idea [19, 20, 5, 9]. In [19] the Kepler+Hadoop project integrates MapReduce processing with the Kepler scientific workflow platform, but the issue of accessing data in scientific formats still remains. In [20] the authors enable NetCDF processing via Hadoop but first require the data be converted into text. This is undesirable as it requires (potentially) significant IO and adds data management issues while also sacrificing the portability of scientific file formats.

There has been work towards extending the NetCDF Operator library to support parallel processing of NetCDF files to reduce data movement [18]. This work analyzes existing NCO scripts and performs automatic parallelization. However, users must still use a scripted, procedural approach to expressing their queries. This approach is format-specific and it doesn't deal with fault-tolerance.

There is existing work in general purpose, array-based query languages [10, 8, 15, 17], but all require data be stored in their respective internal formats. In fact, in [8] the language is prototyped on top of the NetCDF data access library. However, their implementation does not attempt to execute queries in parallel.

HadoopDB [1], SciDB [2] HadoopToSql [7] and HBase [6] are similar to SciHadoop in that they enable the execution of high-level queries on top of distributed storage and partition the query according to where data is locally accessible. However, they all require that the data be ingested into their underlying stores and do not operate over data in-situ, which is one of our primary goals. They also require (potential) data conversions and egressing data is non-trivial.

The Apache Hama Project [13] can be used for efficient matrix operations. SciHadoop is a more general system, but it may benefit from specific optimizations present in Hama.

8. CONCLUSION AND FUTURE WORK

A system for in-situ query execution over scientific data using Hadoop MapReduce has been designed, implemented and evaluated. Holistic functions that are not typically amenable to efficient MapReduce style processing were considered in the context of this system, and a combiner that allowed for opportunistic application of said functions implemented. Among our findings is the discovery that it is possible, and beneficial, to trade map process read locality for enabling more efficient application of the holistic combiner. In future work, we plan on improving IO efficiency of data-intensive processing for scientific data. Specifically, creating a common caching layer shared between map processes to reduce duplicate reads and memory pressure currently caused by isolated library-based caches. We also intend to expand support to other file formats such as HDF5. This may be challenging because the metadata needed for logical/physical conversion can be spread throughout the file in contrast to the NetCDF header that contains all necessary metadata. Also, several performance improvements are planned to reduce storage and computational costs of routing data through the system, thereby reducing runtimes.

9. ACKNOWLEDGMENTS

We would like to thank our shepherd, Shane Canon, for his feedback and guidance. We'd also like to thank our collaborators: John Bent, Meghan Wingate and Gary Grider at Los Alamos National Laboratory; Russ Rew at Unidata; Maya Gokhale at Lawrence Livermore National Laboratory and our colleagues in the Systems Research Lab and Institute for Scalable Scientific Data Management at UC - Santa Cruz. Funding provided by DOE grant DE-SC0005428 and (partially) NSF grant #1018914.

10. REFERENCES

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. In *Proceedings of Very Large Data Bases (VLDB '09)*, Lyon, France, August 24-28 2009.

[2] Paul G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.

[3] Joe Buck, Noah Watkins, Jeff Lefevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott A. Brandt. SciHadoop: Array-based query processing in hadoop. Technical Report UCSC-SOE-11-04, UCSC, 2011.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[5] Thilina Gunarathne, Tak-Lon Wu, Judy Qiu, and Geoffrey Fox. Mapreduce in the clouds for science. *Cloud Computing Technology and Science, IEEE International Conference on*, 0:565–572, 2010.

[6] HBase homepage. `http://http://hbase.apache.org/`.

[7] Ming-Yee Iu and Willy Zwaenepoel. HadoopToSQL: a mapreduce query optimizer. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, pages 251–264, 2010.

[8] Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: design, implementation, and optimization techniques. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 228–239, New York, NY, USA, 1996. ACM.

[9] S. Loebman, D. Nunley, Yong-Chul Kwon, B. Howe, M. Balazinska, and J.P. Gardner. Analyzing massive astrophysical datasets: Can pig/hadoop or a relational dbms help? In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, 31 2009-sept. 4 2009.

[10] Rona Machlin. Index-based multidimensional array queries: safety and equivalence. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 175–184, New York, NY, USA, 2007. ACM.

[11] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI 02*, OSDI '02, pages 131–146, New York, NY, USA, 2002. ACM.

[12] NetCDF operator (NCO) homepage. `http://nco.sourceforge.net/`.

[13] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, IEEE CloudCom 2010. IEEE, December 2010.

[14] Konstantin V. Shvachko. HDFS scalability: the limits to growth. *;login*, 35(2):6–16, April 2010.

[15] Emad Soroush, Magdalena Balazinska, and Daniel Wang. Arraystore: a storage manager for complex parallel array processing. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 253–264, New York, NY, USA, 2011. ACM.

[16] Michael Stonebraker, Jacek Becla, David Dewitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stan Zdonik. Requirements for science data bases and SciDB. In *CIDR 09*, 2009.

[17] Alex van Ballegooij, Roberto Cornacchia, Arjen de Vries, and Martin Kersten. Distribution rules for array database queries. In Kim Andersen, John Debenham, and Roland Wagner, editors, *Database and Expert Systems Applications*, volume 3588 of *Lecture Notes in Computer Science*, pages 55–64. Springer Berlin / Heidelberg, 2005.

[18] Daniel L. Wang, Charles S. Zender, and Stephen F. Jenks. Clustered workflow execution of retargeted data analysis scripts. In *CCGRID 2008*, 2008.

[19] Jianwu Wang, Daniel Crawl, and Ilkay Altintas. Kepler + Hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems. In *SC-WORKS*, 2009.

[20] Hui Zhao, SiYun Ai, ZhenHua Lv, and Bo Li. Parallel accessing massive NetCDF data based on MapReduce. In *Web Information Systems and Mining*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010.