SciHadoop: Array-based Query Processing in Hadoop

Joe B. Buck
UC Santa Cruz
buck@cs.ucsc.edu

Noah Watkins
UC Santa Cruz
jayhawk@cs.ucsc.edu

Kleoni Ioannidou
UC Santa Cruz
kleoni@cs.ucsc.edu

Jeff LeFevre
UC Santa Cruz
jlefevre@soe.ucsc.edu

Neoklis Polyzotis
UC Santa Cruz
alkis@ucsc.edu

Carlos Maltzahn
UC Santa Cruz
carlosm@cs.ucsc.edu

Scott Brandt
UC Santa Cruz
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. This limits the 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 a representative holistic aggregate function query: reduce total data transfers, reduce remote reads, and reduce unnecessary reads. Two optimizations allow holistic functions 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 I/O, 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 large-scale data-intensive 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. The MapReduce framework excels at tackling problems that are easily sub-divided, and many operations on scientific, array-based data have the property of being *embarrassingly 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 have been developed for scientific data analysis. One popular set of tools, the NetCDF Operators (NCO) [8], 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[12] project has been successful in parallelizing the execution of NCO queries by shipping sub-queries 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 the solution requires that data first be transformed into a text-based format[14]. While this may at first appear reasonable, the overhead of format transformation and additional data management costs are too great given the 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 to processing scientific data with MapReduce 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 from an unfortunate partitioning of a computation's input.

In general, the logical layout exposes no information about the physical layout and distribution of data. This disconnect is a road block for many types of optimizations that utilize resources defined at the physical layer, such caching and 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 partitioning of a computation's input, and allows 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, and use this information to create a more performant 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. These include 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 (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 [10].

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 combine function also known as *combiner*, 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. Next we present the data model assumed by MapReduce, and details of a 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 streams* 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 (e.g. POSIX), 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 *locations*, H_i, for any block b_i.

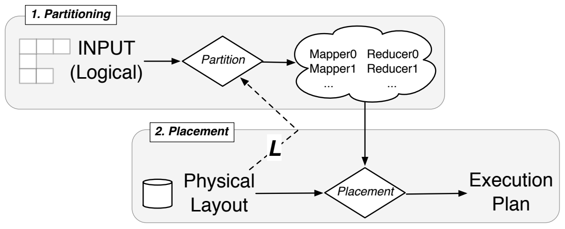


Figure 1: MapReduce processes logical partitions in map tasks, and matches each map task with physical locations to form an execution plan. The line labeled **L** is a contribution of SciHadoop which utilizes physical layout knowledge during partitioning (see Section 4).

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. 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) that 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 frequently used 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 construct schedules in which a map task processes a block on a host that stores that block locally.

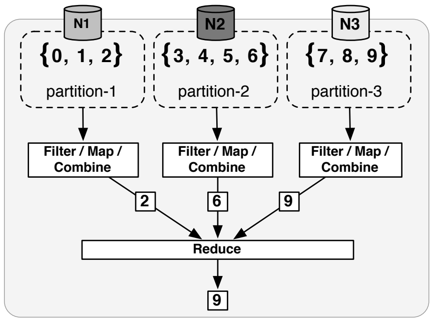


Figure 2: The execution of the *max* aggregate function in MapReduce. The map phase locates the local maximum in each partition and the reduce phase combines the results to produce to global maximum value.

Consider the scenario illustrated in Figure 2. This figure shows 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 $N1$, $N2$, and $N3$. 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 that will be processed by the map task. Thus, *partition-1* is processed on the node with block *N1*, 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 data based on the filter using *ExtractInput()*. A *Group* is assigned to the data representing a single function input. he repartitioned data is then sent to the combine function using *Emit()*.

The *combine* function is utilized to calculate per-group partial results from the output of each map function on a given node before being sent to the reduce function. This

Algorithm 1: Map()

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

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, Partition
Max ← Maximum(Partition)
Emit(Group, Max)
```

2.2 Scientific Data

Unlike the byte streams data model assumed by MapReduce, scientific data is commonly represented by a multi-dimensional, array-based data model [11]. In this section we present a simple version of such a data model, and develop a query language used to express some common types of 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, as illustrated in Figure 4 (b). 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). 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 a performant 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 4 (a) 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 4 (a), 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]

[1]We have omitted values for the non-shaded region in order

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

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

where S_A and c_A are the shape and corner point of A, respectively. Thus, the shaded sub-array at the top of Figure 4 (a) 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.

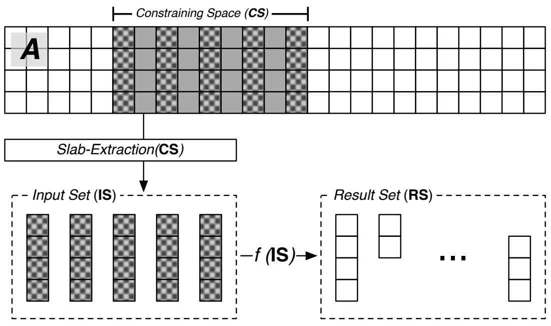


Figure 3: Illustration of the query language semantics. First, a subset of an array A, referred to as the *constraining space* is used to limit the scope of a query. The process of *slab extraction* forms the *input set*. Finally, the function f is applied to the *input set* to produce the *result set*.

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 4 (a) depicts temperature readings associated with 2-dimensional coordinates specified by *latitude* and *longitude* values. Thus, the shaded sub-array may represent a specific geographic subregion 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 4 (a)?"*

In order to express this type of query we have developed a simple query language. The language serves as a formal model that allows us to develop general techniques, and also exposes semantic information used to reason about query optimizations.

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 4 (a), and produces as output a 1×1 array where the output array's single cell contains the value 9.

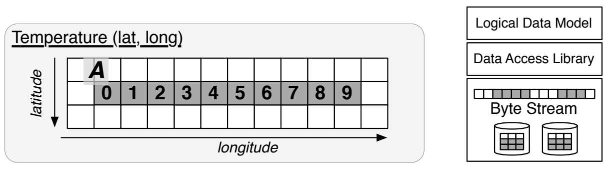
to simplify the discussion. The non-shaded regions can be interpreted as simply *null* values as they are often referred.

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 3 illustrates the semantics of the language. In the figure, 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].



(a) (b)

Figure 4: Sub-figure (a) illustrates a 2-dimensional data set. Sub-figure (b) is a depiction of the software stack used with scientific access libraries.

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.

In summary, the translation between the logical view of data and the byte-stream offset of values are completely hidden within the data access library, and all access to data must be addressed at the logical level, and be served by the library. Next we present a straw man solution and show how it can perform poorly under common conditions.

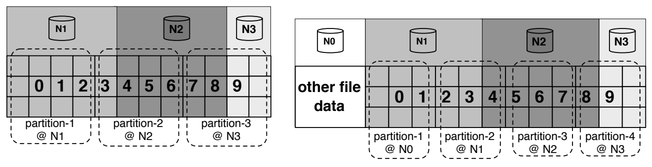
3. PARTITIONING REVISITED

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 performance problems.

Scientific access libraries use a black-box design in which file layout knowledge is obscured, effectively disabling 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 the 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 5 (a) which shows a 3×12 array stored within a file occupying three physical blocks, located on nodes $N1$, $N2$, and $N3$. Using knowledge that the file format stores data in column-major order, a reasonable partitioning is to form three equally sized partitions, each containing four columns. These partitions are shown in the figure using dashed frames, and are labeled as *partition-1,2,3*.

A simple placement heuristic that we assume is a round-robin matching of logical partitions to physical locations. The result of a round-robin placement is shown in Figure 5 (a) using the notation *partition-x* @ *Ny*. For example, *partition-2* is processed on node *N2*. To illustrate the difference in logical and physical partitions consider the example shown in Figure 5 (a). In the example, *partition-1* references 4 columns, all contained in block $N1$, while *partition-3* references 2 columns from block $N2$, and 2 columns from block $N3$. 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: Sub-figure (a) show the baseline partition strategy applied to a file containing a single array. Sub-figure (b) shows the same partitioning strategy applied to the same array in a different file containing additional arrays.

For infrastructures with network bottlenecks, and when I/O 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 exhibit only a small amount misalignment. As a result, the baseline partitioning strategy will generally perform well. Unfortunately, achieving a good partitioning in the general case can be difficult.

Consider now the task of partitioning the same logical array shown in Figure 5 (a), 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 other large data sets stored within the same file. A depiction of this new file is shown in Figure 5 (b). 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 due to the baseline partitioning strategy. The second, 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 $N0$.

In the remainder of this paper, the baseline partitioning strategy will serve as, unsurprisingly, a *baseline* for performance measurements of SciHadoop optimizations introduced in Section 4. 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, each of which evaluates a query over an identical logical array using the *baseline partitioning strategy*. The first array 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 baseline partitioning and placement achieves 71% read locality, but when multiple arrays are stored in a single file the read locality drops to 5% due to the misalignment that occurs during placement. 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 the baseline for the rest of the paper.

In summary, this section has shown that scientific data cannot be processed using standard scan-based approaches due to access library limitations, but that an alternative input partitioning at the logical level can lead to poor performance. Thus, any performant 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 SCI-HADOOP

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. As described in Section 3, a query is converted into an execution plan by decomposing the query into multiple subqueries using *logical* partitioning. These sub-queries in turn are evaluated using map, combine, and reduce tasks, and each are placed on nodes for execution using the default Hadoop scheduler that attempts to minimize remote reads and runtime.

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. Recall from Section 2.1 the general flow of data in MapReduce: on a single node, map tasks read and partition data, which are then grouped together by combiners on the same node, and finally each group is forwarded to reduce tasks. As we will observe in Section 6 these data transfers introduce multiple overheads, including network traffic and memory buffer swaps to disk. One optimization, already available with the *baseline partitioning strategy*, is to evaluate aggregate functions in combiners, rather than reducers. This optimization can significantly reduce the volume of data transferred over the network to reduce tasks. However, this optimization is only available to a class of aggregate functions with the property that multiple, partial results can be combined, such as *max* and *sum*. So called holistic aggregate functions, such as *median*, are not capable of computing intermediate, partial results, and thus it would seem that holistic functions must be applied only during the reduce phase, as a given input may be divided between more than one input partitions.

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 set is available on a single node (potentially in multiple partitions). This is possible because the semantics exposed by the logical data model and query language can be used to identify such opportunities. While the first optimization enables data transfer reductions by extending the use of combiners to holistic function evaluation, the second optimization is used to increase the probability of the technique being applicable in a given query execution. Recall that the first optimization is only available when the function input exists in partitions being processed on a single node. The second optimization is capable of making adjustments to input partitions to increase the likelihood of holistic function inputs being present in single partitions.

Reducing remote reads. In Section 2.1 we described the process of placement in MapReduce that determines the location within a cluster where a partition will be processed, and remote reads can be reduced by scheduling a partition to be processed on a node that stores the data contained in the partition locally. However, Section 3 demonstrates that it is difficult in general to make placement decisions regarding partitions defined at the logical level of a scientific data model because access libraries hide details regarding the physical layout of data expressed at the logical level. SciHadoop implements two techniques for reducing the amount of remote reads where simple round-robin heuristics will perform poorly.

The first optimization employed by SciHadoop, referred to as *physical-to-logical translation*, generates logical partitions that reference *exactly* the data contained within a single physical block. This technique directly translates the extent represented by a physical block into its equivalent logical representation, and in doing so implicitly defines the placement of the partition; the logical partition should be placed according to the physical block from which it was generated. Unfortunately this technique is not always feasible to implement, and a more general method is needed in such cases.

The second technique SciHadoop can use to reduce remote reads is referred to as *chunking and grouping*. This technique decomposes the input into small units called chunks, from which a random sampling of its physical locality is taken using extensions to scientific access libraries. The sampling technique allows SciHadoop to perform a grouping of chunks into flexibly defined partitions with increased locality of reference.

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 provides SciHadoop with the opportunity 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

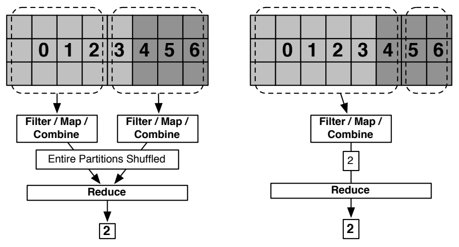
A holistic aggregate function has the property that it cannot be computed by combining multiple, partial results. For example, consider the task of calculating the *median* value from the range $0 \ldots 4$ using MapReduce. Figure 6 (a) shows the execution flow of this query. Since neither of the two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

Holistic Combiner. In SciHadoop partial aggregate values for non-holistic functions are computed, when possible, during the map phase. However in general, 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 combine phase when the entire input to the holistic function is present in one or more partitions being evaluated on a single node.

Holistic-aware Partitioning. While the holistic combiner can evaluate a holistic function when the entire input is contained in the partition(s) on a single node, the initial partitioning of the logical input space is entirely unaware of query being processed, 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 can make small adjustments to partitions to increase the likelihood that holistic functions can be evaluated prior to the reduce phase. Figure 6 (b) shows how the first partition is adjusted to include the entire holistic function input.

4.2 Reducing Remote Reads

In this section we introduce two 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,



(a) (b)

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.

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.

Physical-to-Logical Translation. Figure 7 (a) illustrates how optimal 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, and placement is trivial; a partition is matched with the partition from which it was generated.

Despite the benefits of this technique, it can be difficult to implement for complex file formats, as well as when the amount of metadata required to perform the conversion is large and dispersed. For example, this technique can be performed trivially in memory for NetCDF by reading a file's single, 4KB header. However, in NetCDF-4/HDF5 metadata is spread throughout the byte stream.

Chunking and Grouping. SciHadoop provides an alternative to direct physical-to-logical translation based on random sampling. In this approach SciHadoop constructs partitions by decomposing the input space into many fixed-size chunks, and then grouping chunks together into flexibly defined partitions based on an estimation of each chunk's locality of reference. By choosing a sufficiently small chunk size, the granularity at which the logical space is decomposed allows for efficient groupings, that taken together, reference data in the byte stream with a high degree of locality.

Figure 7 (b) 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 that 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 the 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 range of the 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 the block. Next we present a concrete example of the functioning of this technique using query **Q1** previously discussed in 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 7 (b). We refer to these chunks as by their position in the figure, specifically chunks $1 \ldots 6$.

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 $N1$ or block $N2$. 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 7 (b). The final LTP mapping is given as:

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

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.

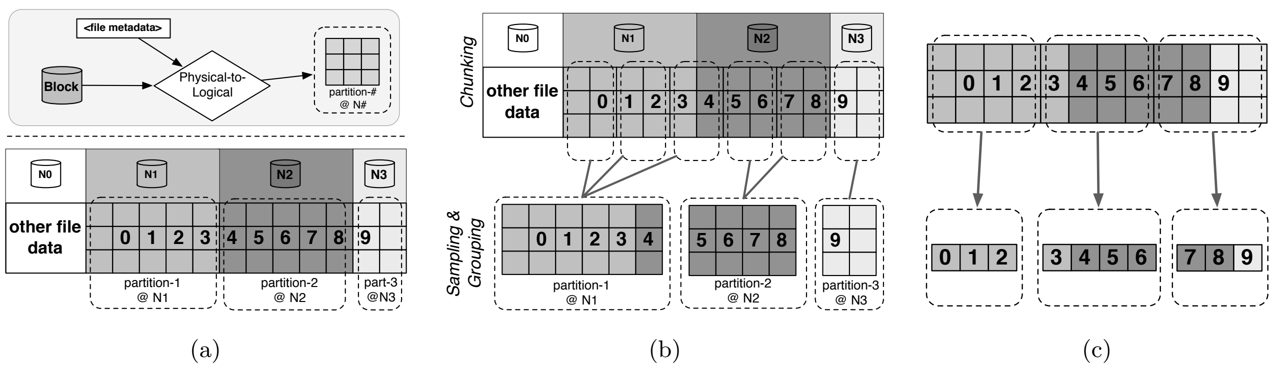


Figure 7: Sub-figure (a) illustrates direct physical-to-logical translation (Section 4.2. Sub-figure (b) depicts the process of chunking and grouping (Section 4.2). Sub-figure (c) illustrates the *NoScan* optimization described in Section 4.3.

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 construction of LTP does not consider the data requirements of a given query. 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', and in order to form this new mapping, an optimization process must consider the data requirements of a query.

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 7 (c) 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 Hadoop MapReduce framework *v0.21*, and in this paper we target the NetCDF-3 scientific file format.

Hadoop plugin. SciHadoop query processing is implemented as a standard MapReduce computation, partitioning and placement are implemented using low-level customizations to Hadoop's default partitioning and placement that targets text-based formats. The initialization phase of SciHadoop, handled by Hadoop's *FileInputFormat* class, performs chunking and grouping which result in a set of partitions, each related to a host on which the partition should be processed. A partition in SciHadoop is represented by the Hadoop *InputSplit* class that in SciHadoop represents a subset of the total logical input space. Each *InputSplit* and associated partition is scheduled transparently by Hadoop using higher level policies such as load balancing. An *InputSplit* is processed at a cluster node by reading the associated logical space from the underlying file system. The reading capability is built into Hadoop's *RecordReader* class that utilizes the NetCDF-Java library to access data specified at the logical level. The *RecordReader* prepares the data to be send to the *map* computation, at which point the processing resembles the description of the *map*, *reduce*, and *combine* functions discussed in Section 2.1. MapReduce is used with default configurations, including those that influence the costly memory spills that result as buffers fill and memory pressure must be relieved.

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 translated POSIX calls issued by NetCDF-Java into calls to the HDFS library. Note that HDFS is itself an append-only file system, thus SciHadoop is currently limited to read-only workloads.

6. EVALUATION

We conducted an experimental study to measure the performance of our SciHadoop prototype on representative queries and data.

Experimental Setup: All experiments described are performed on a 33-node cluster. Each node has a dual-core

Opteron (1.8 GHz) processor, 8GB of RAM, 4 250GB Seagate Barracuda ES SATA drives, and dual GigE Ethernet ports. The nodes are connected using an Extreme Networks' Summit 400 48-t switch. Each node is running a stock installation of Ubuntu 10.10, and the deployed version of Hadoop is 0.21. The Hadoop cluster is configured with 1 master node and 32 MapReduce / HDFS nodes. Each HDFS node uses 3 full SATA drives formatted with Btrfs for a total system capacity of 27 terabytes. The OS is installed on a fourth SATA drive that is also used as *temporary storage* by Hadoop.

6.1 Methodology

NetCDF Dataset: We use a data set modeled after a schema from UCAR as the basis for our experiments. The data set contains a single variable measuring air pressure collected over time, latitude, longitude, and altitude. This variable is defined over the following four dimensions: latitude, longitude, elevation, and time. In order to achieve greater data volumes we have artificially expanded the file to approximately 132 GBs by extending the length of each dimension. The following CDL shows the schema of our data set:

```
netcdf airpressure.nc {
dimensions:
    time = UNLIMITED ; // (5475 currently)
    lat = 360 ;
    lon = 360 ;
    elev = 50 ;
variables:
    int pressure(time, lat, lon, elev) ;
```

Query: The holistic query used to evaluate our system is "Calculate the median air pressure within a lat/lon, between two adjacent days in the dataset".

In terms of the SciHadoop query language, query Q1 looked like this:

```
Total Logical Space: 5475, 360, 360, 50
Constraining Space: 547, 0, 0, 0 -> 4927, 360, 360, 50
Extraction Shape: 2, 36, 36, 10
Result Set Shape: 2190, 10, 10,5
```

The primary metrics used in the evaluation section are *CPU utilization*, *runtime* of a query, *HDFS Local Reads*, *Temporary Storage Writes*, and *Data Transfer Size*. Of these the last three deserve more explanations:

HDFS Local Reads. A measurement of the fraction of the input to the MapReduce function that was read from a local disk managed by HDFS.

Temporary Storage Writes. A measurement of how many bytes were written to temporary storage. Potential causes of locally written bytes include IO spills by map or combine tasks, due to buffer space being exhausted, as well as reducer processes needing to do multiple merge-sort passes to organize their inputs. All bytes in this metric are written to the fourth disk of each node as temporary storage.

Data Transfer Size. The amount of data transferred between the map and reducer tasks.

6.2 Results

Table 1 provides an overview of our results. We will discuss them in terms of our optimization goals.

Test Name	CPU Util (%)	Run Time (Min)	Local Read (%)	Temp Write (GB)
First 4 use no Holistic Combiner				
Baseline	34.7	159	9.3	2,586
Baseline +NoScan	34.3	132	9.2	2,588
ChkGroup	24.3	188	80	2,608
PhysToLog	29.9	201	88	2,588
Next 4 use Holistic Combiner with Baseline				
Baseline	79.1	28	9.5	107
+NoScan	80.7	27	9.5	107
+NoScan *+HaPart*	81.3	26	8.8	107
+HaPart	79.3	26	8.6	107
Next 3 use Holistic Combiner with Local-Read Optimizations				
ChkGroup *+HaPart* +NoScan	84.7	25	70.7	116
ChkGroup +NoScan	83.1	26	79.3	188
PhysToLog +NoScan	82.8	27	88.1	196

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

Reducing Data Transfer. Our optimizations to reduce data transfer (Holistic Combiner and Holistic-aware Partitioning) have a significant impact on runtime: roughly an order of magnitude. As Table 1 shows, tests using Holistic Combiner optimization are largely CPU-bound while others are not. We were admittedly surprised by the extent of the performance impact until we realized that data transfer volume generally causes a significant increase in temporary storage writes and that we only allocated one disk per node for temporary storage in contrast to three disks per node for HDFS which serves map task reads. Temporary storage writes occur when buffers become too full and processes need to write data to disk to free up the space. This occurs when data is passed from map to combiner processes, while combiners are applied, potentially repeatedly, to map output, and at reducers as part of there merge-sort phase (prior to final processing of the data).

Chart 8 shows the amount of bytes transferred between map and reduce tasks. Generally, the tests which do use Holistic-aware Partitioning are able to reduce data transfer by almost three orders of magnitude compared to Holistic Combiner tests without Holistic-aware partitioning. The exceptions to this are the baseline and the baseline with NoScan tests which show a comparable amount of data transferred compared 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.

A potential conflict exists between partitioning strategies to reduce remote reads and partitioning strategies to reduce data transfers: the former partitions along physical data location, 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 lo-

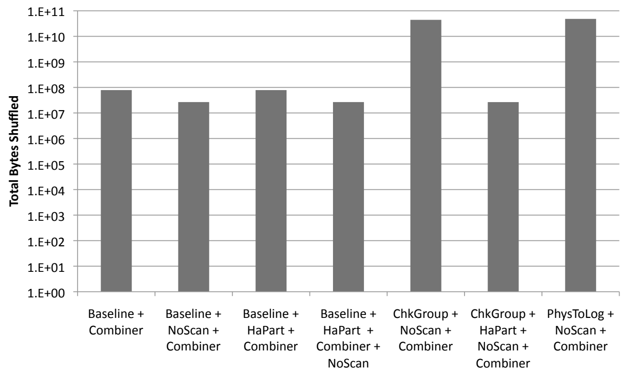


Figure 8: Data Transfer Size in tests with Holistic Combiner optimization enabled (notice the log scale on the y-axis). Generally, Holistic-aware partitioning is reducing the data transfer size by almost three orders of magnitude. Exceptions in this case are the tests using unmodified scientific access libraries ("Baseline ...") where this effect is masked by an incidental alignment.

cal 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).

Reducing Remote Reads. As can be seen in figure 9,

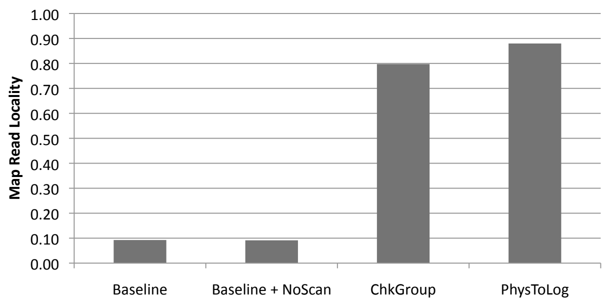


Figure 9: Fraction of HDFS local reads of tests with unmodified (Baseline and Baseline + NoScan) and modified scientific access libraries. Our access library modification as part of our optimizations for reducing remote reads significantly reduce the amount of remote reads.

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

Reducing Unnecessary Reads. The NoScan method reduces the total amount of data read, based on the query being executed. In our tests, we had about 80% selectivity and the experiments showed that 80% as much data was being read (105 GB vs 132 GB) when NoScan was turned on. The impact on run time can be seen in table 1: Baseline + NoScan ran 17 minutes faster than the Baseline experiment without the combiner and a minute faster when the combiner was used.

7. RELATED WORK

Integrating MapReduce systems with scientific data is not a novel idea [13, 14]. In [13] 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 [14] the authors enable NetCDF processing via Hadoop but first require the data be converted into text. This is undesirable as it adds data movement and 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 [12]. 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 [7, 6]. In fact, in [6] the language is prototyped on top of the NetCDF data access library. However, their implementation does not attempt to execute queries in parallel. We do not claim to contribute anything new to the area of query languages, but recognize that as our infrastructure and methods mature, utilizing a richer query language will add value to our work.

HadoopDB [1] is similar to SciHadoop in that it enables the execution of high-level queries on top of Hadoop and partitions the query according to where data is locally accessible. HadoopDB requires that data be ingested into the underlying database, thereby incurring extra data movement and preventing the use of scientific file formats. Similarly, in [5], the Hadoop job is analyzed, transformed into an SQL statement and the input spaced reduced in order to access only necessary data to fulfill the query. Refactored queries can make use of proven database facilities, like indexes, to speed up queries.

The SciDB project [2] also builds on the idea of using database systems to enable large-scale analysis of scientific data. It requires data to be written & read from itself, as opposed to in-situ analysis of existing data, precluding the use of scientific file formats, which is one of this projects primary goals.

The Apache HAMA Project [9] implements efficient matrix multiplication algorithms using the Hadoop MapReduce framework. The HAMA project stores data in HBase, and doesn't attempt to process array-based data in a native format.

8. CONCLUSIONS

A system for in-situ execution of queries over scientific data using Hadoop map/reduce has been designed, implemented and evaluated. Holistic functions which are not typically amenable to efficient MapReduce style processing, were considered in the context of this system. Among our findings is the discovery that it is possible, and beneficial, to trade read locality at map processes for enabling more efficient application of the holistic combiner. In future work, we plan on improving I/O 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.

9. ADDITIONAL AUTHORS

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] 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.

[6] 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.

[7] 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.

[8] netcdf operator (nco) homepage.

[9] 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.

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

[11] 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.

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

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

[14] 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.