# SIDR: Structure-Aware Intelligent Data Routing in Hadoop

Joe Buck[1], Noah Watkins[1], Greg Levin[1], Adam Crume[1], Kleoni Ioannidou[1],
Scott Brandt[1], Carlos Maltzahn[1], Neoklis Polyzotis[1]and Aaron Torres[2]

[1]Department of Computer Science
University of California - Santa Cruz
{buck,jayhawk,glevin,adamcrume,kleoni,
scott,carlosm,alkis}@cs.ucsc.edu

[2]Los Alamos National Laboratory
agtorre@lanl.gov

## ABSTRACT

The *MapReduce* framework is being extended for domains quite different from the web applications for which it was designed, including the processing of big structured data, e.g., scientific and financial data. Previous work using *MapReduce* to process scientific data ignores existing structure when assigning intermediate data and scheduling tasks. In this paper, we present a method for incorporating knowledge of the structure of scientific data and executing query into the *MapReduce* communication model. Built in SciHadoop, a version of the Hadoop *MapReduce* framework for scientific data, SIDR intelligently partitions and routes intermediate data, allowing it to: remove Hadoop's global barrier and execute Reduce tasks prior to all Map tasks completing; minimize intermediate key skew; and produce early, correct results. SIDR executes queries up to 2.5 times faster than Hadoop and 37% faster than SciHadoop; produces initial results with only 6% of the query completed; and produces dense, contiguous output.

## Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed Systems; H.2.4 [**Systems**]: Query Processing; D.4.7 [**Organization and Design**]: Distributed Systems

## General Terms

Hadoop, MapReduce, Scientific Data

## 1. INTRODUCTION

*MapReduce* is a simple framework for parallelizing the processing of large datasets. Its rise in popularity has led to its use in problem domains and with types of data beyond those for which it was originally designed, including scientific computing [9, 15, 39]. For large-scale scientific datasets, *MapReduce* is an attractive framework due to its simple programming model, ability to scale well on commodity hardware, and because many problems in scientific computing translate well to its type of parallelization [7, 34, 9].

Scientific data is often highly structured (modeled as an array, sphere, or some other shape [11, 26, 17]) and stored in file-formats where metadata describing that structure resides alongside the actual data. Previous efforts in applying *MapReduce* to scientific data typically fall into two categories: 1) those that make little use of this additional structure, preferring to process entire files [24, 35], thereby ignoring locality concerns and potentially facing scalability limitations and, 2) those that extract data from its native format and store it in some other layout [19, 8, 2], likely achieving performance gains while forfeiting access to the data in its native format and incurring the resource costs inherent in reformatting. SciHadoop [4] is a recent project that provides the best of both categories by utilizing the structured nature of scientific data, stored in its native file format, to achieve data locality and thereby improve the performance of Hadoop queries over scientific data. SciHadoop's extension of Hadoop was limited to improving the assignment of input to *Map* tasks, later data assignment and communication was unaltered.

While literature indicates that it is common for scientific queries to have some alignment to the structure of the data they are executing on (Section 2.1), the existing *MapReduce* model is agnostic of any order in either its input or intermediate data. This paper presents SIDR (**S**tructure-Aware **I**ntelligent **D**ata **R**outing), an extension of the *MapReduce* model that takes advantage of naturally occurring alignments, and an implementation of that extended model in Hadoop. Building upon SciHadoop's ability to efficiently read scientific data as input to Hadoop, SIDR alters how Hadoop assigns intermediate data, schedules tasks and organizes its output for a large class of queries over scientific data. SIDR is able to provide correct, partial results prior to the completion of all *Map* tasks and guarantee that the number of keys assigned to each *Reduce* task is consistent (key skew is prevented); two topics that have each been the focus of multiple papers, [5, 32] and [27, 21] respectively. Additionally, SIDR allows for the prioritization of portions of the output during scheduling and organizes intermediate data into dense, contiguous chunks, resulting in more efficient query output. These added features, and performance improvements deriving from the new model, significantly advance our goal of providing the broader scientific computing community with convenient access to a scalable platform for in-situ processing of scientific data.

*SC '13,* November 17 - 21 2013, Denver, CO, USA

ACM 978-1-4503-2378-9/13/11 ...$15.00
http://dx.doi.org/10.1145/2503210.2503241.

Table 1: Summary of Common Symbols

| | |
|---|---|
| Sets | |
| $\mathbb{T}$ | *MapReduce* query input set |
| $I$ | Set of all input splits |
| $\mathbb{O}^{\mathbb{T}}_{\mathrm{es}}$ | Output from a *MapReduce* query |
| $v'_{k'}$ | set of all values in $v'$ that are part of a key/value pair where the key is $k'$ |
| $\mathbb{K}^{\mathbb{T}}$ | set of keys that actually exist in $K$ for a *MapReduce* query |
| $\mathbb{K}^{\mathbb{T}}_i$ | set of keys that actually exist in $I_i$ for a *MapReduce* query |
| $\mathbb{K}'^{\mathbb{T}}_\ell$ | set of keys in $K'$ assigned to $\text{KEYBLOCK}_\ell$ for a given query |
| Elements | |
| $I_i$ | the $i^{th}$ input split |
| $k, k'$ | a key in spaces $K$ or $K'$, respectively |
| $\langle k, v \rangle$ | *key/value* pair in $K \times V$ |
| $\langle k', v' \rangle^i$ | *key/value* pair in $K' \times V'$ created by the *Map* task processing $I_i$ |
| $m_i$ | particular *Map* task |

This paper's contributions include:

1. A formal extension to the *MapReduce* communication model that, for structural queries, introduces determinism between input and output keys while preserving correctness, prevents skew in intermediate data, reduces the amount of required communication and enables the production of early results.
2. A working implementation of this new communication model within Hadoop that includes: a new partitioning function that incorporates knowledge about the executing query and input data, a modified scheduler that enables co-scheduling of *Reduce* tasks with the *Map* tasks they depend on and *Reduce* tasks starting as soon as their data dependencies are met
3. A performance evaluation of SIDR relative to both SciHadoop and Hadoop.

## 2. BACKGROUND

### 2.1 Scientific Data

Scientific data is typically stored in binary file formats that provide higher-level abstractions for accessing data. A common abstraction used by these libraries is a coordinate-based system where data is read and written from files via functions that take coordinate arguments in lieu of byte-offsets and then translate those coordinates into accesses in the underlying file. Common scientific file formats include NetCDF [26], HDF5 [17], FITS [11], and GRIB [13] (for the rest of this paper, we use NetCDF notation). Given scientific libraries' requirement that data access occur via coordinates, SciHadoop, which this work builds upon, specifies its units of work via pairs of $n$-dimensional coordinates specifying a corner and a shape in the input data set (e.g., corner: {100,0,0} shape: {20, 50, 50} indicates a 50,000 element cube with its origin at {100,0,0}).

Scientific file formats typically encode structural metadata alongside data in a single file. This metadata is typically exposed by a function that returns the dimensions and data type being stored. Figure 1 is an example of metadata for a file used in our experiments.

```
dimensions:     variables:
   time = 365;     int temperature(time, lat, lon);
   lat = 250;
   lon = 200;
```

Figure 1: Metadata (in NetCDF format) for the dataset shown in Figure 2 whose dimensions are {365, 250, 200} representing 365 daily measurements covering 25°N to 50°N in 1/10° increments and 85°W to 65°W in 1/10° increments.

### 2.2 Scientific Queries

Scientific queries often have a structural component. Specifically, the queries use the semantic meaning of where data points occur within a file to analyze the data. Figure 2 is an example dataset consisting of temperature measurements from the eastern US. Examples of structural queries over this dataset include:

1. Find the weekly averages for every unique location.
2. Find all locations where the 24-hour temperature variations exceed X.
3. Sort the data points for each day by temperature.

These queries are not only representative of climate-oriented research but are functionally equivalent to histogramming in high energy physics [9], identifying stars displaying sudden shifts in brightness (astronomy) [28] and using evolving models to find anomalous behaviors in network packets (machine learning) [14]. We refer to this class of queries as structural queries.

A review of existing work on processing large scientific datasets with *MapReduce* or similar frameworks indicates that structural queries are common in a wide variety of fields. In addition to those already mentioned in this section, we found pairwise sequence alignment [15] (bioinformatics), a variety of selection and correlation queries in astrophysics [24], generating periodograms of stars at different wavelengths [33] and as part of a merge-based system for clustering algorithms [29] (astronomy), and analyzing time series (both microscopic images and measurements) of live cells [38] (genomics).

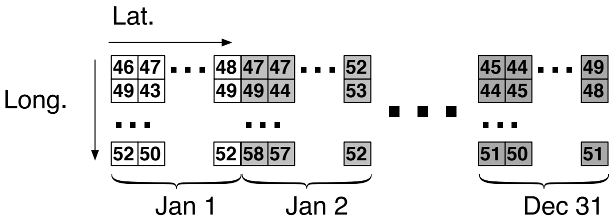


Figure 2: Example data representing daily temperature measurements for the east coast of the US over one year.

Not all queries over scientific data are structural. For example, performing a group-by or join operation on the values in a dataset does not display the structural properties that SIDR capitalizes on; these queries interact with the values in the dataset irrespective of the data's location. We refer to these queries as structure-oblivious. Additionally, queries whose individual computations require global communication are considered structure-oblivious for our purposes as they are not amenable to our structure-informed optimizations. While SIDR is exclusively focused on structural queries, previous projects that used *MapReduce*-like systems for structure-oblivious workloads in scientific computing include $k$-means clustering (data mining) on DryadLINQ [8], and physics simulations requiring a global merge phase on CGL-MapReduce [9].

## 2.3 MapReduce

The decision to use a *MapReduce* system as the basis for our research was motivated by two factors: 1) the design of *MapReduce* aligns perfectly with our ultimate goal of extending a parallel filesystem to support the in-situ analysis of scientific data in its native file format and 2) building upon Hadoop allowed us to proceed directly to conducting research without having to build and support our own distributed, fault-tolerant, parallel processing system. In our discussion of *MapReduce*, we reference Hadoop [16] for implementation details, as it is the predominant publicly-available *MapReduce* framework. A more detailed discussion of our choice to build upon Hadoop is presented here [4].

Figure 3 shows the dataflow for a *MapReduce* query. When a query begins executing, a central coordinator partitions the specified input data, $\mathbb{T}$, into a set $I$ consisting of *InputSplit*s, denoted $I_i$. An *InputSplit* is typically defined as byte-ranges in one or more files (e.g., bytes 1024 - 2048 in file "dataFile1"). Each split is assigned to one *Map* task that employs a file-format specific library, called a RecordReader, to read the assigned $I_i$ and output key/value pairs where the keys are in keyspace $K$. Those key/value pairs are consumed by said *Map* task which then outputs new key/value pairs, referred to as intermediate data, with keys in a different keyspace, $K'$. A partition function then deterministically maps the $k'$ key for each intermediate key/value pair to one KEYBLOCK (a "KEYBLOCK" is a partition of the keyspace $K'$ for the given query).

Each *Reduce* task is assigned a KEYBLOCK and processes all intermediate data assigned to that KEYBLOCK. Prior to the application of the *Reduce* function, *Reduce* tasks merge all their data into a sorted list, combining all key/value pairs with the same $k'$ key into a pair consisting of a single instance of the key and a list containing all the values (denoted $v'_{k'}$). The sorting and merging ensures that all values corresponding to a given key will be processed by the *Reduce* function at the same time. As a *Reduce* task processes its assigned data, it emits a new set of values in the output space $\mathbb{O}$.

*MapReduce* makes two guarantees: 1) all input will eventually be processed by a *Map* task and, 2) for a given $k'$, all values will be processed at the same time, by a single *Reduce* task. The *MapReduce* framework is free to partition data and schedule tasks as it sees fit. In practice, data locality information is often used to partition and assign the input. A lack of ordering guarantees among tasks of a given type (*Map* or *Reduce*) and atomic committal of task output affords *MapReduce* a high degree of scheduling flexibility.

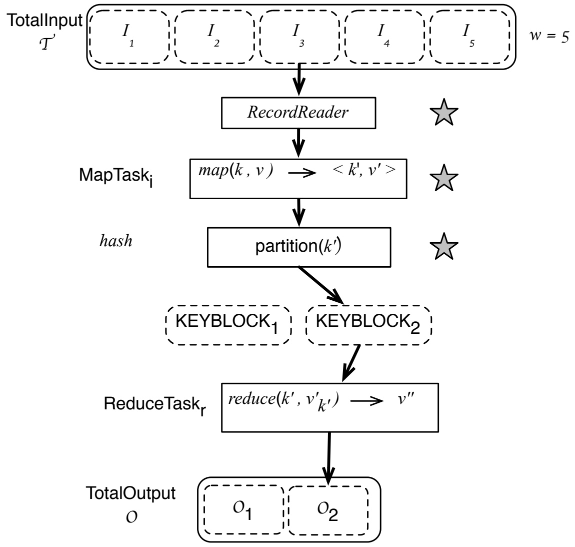


Figure 3: Annotated dataflow for a *MapReduce* query. The arrow follows one possible path of a data point through the entire query with stars indicating areas in the dataflow where keys are opaquely mapped between different keyspaces.

### 2.3.1 *The* MapReduce *Barrier*

*MapReduce*'s single ordering constraint is that a *Reduce* task only processes data for a given $K'$ key when it has all of the key/value pairs with that particular key. Existing *MapReduce* frameworks make no attempt to reason about which keys will be generated by the collective *Map* tasks and how those keys will be assigned to KEYBLOCKs, necessitating a worst-case assumption that any *Map* task may create output for any *Reduce* task. The resultant barrier between the end of the last *Map* task and the beginning of any *Reduce* task, thereby guaranteeing all values for a given key will be processed at the same time. Figure 4a depicts this barrier and it is explored in more detail in Section 3.2.

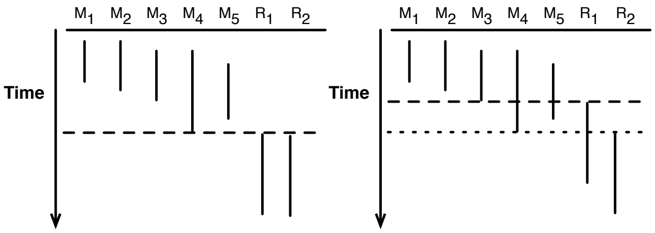

M$_1$ M$_2$ M$_3$ M$_4$ M$_5$ R$_1$ R$_2$
Time
M$_1$ M$_2$ M$_3$ M$_4$ M$_5$ R$_1$ R$_2$
Time

Figure 4: **Left**: In general, *Reduce* tasks wait for all *Map* tasks to complete prior to beginning their execution. **Right**: *Reduce* tasks waiting for their actual data dependencies (e.g., $R_1$ depends on $M_1, M_2, M_3$; $R_2$ on $M_3, M_4, M_5$).

### 2.3.2 MapReduce*'s Communication Model*

While a significant body of work relating to formal definitions of the *MapReduce* query model and its relation to other computing models exists [6, 10, 20, 22], *MapReduce*'s internal communication model has received little attention. Reasoning about the *MapReduce* dataflow is complicated by three areas in the communications model where it is difficult, or impossible, to correlate a function's input with its output.

These areas, marked by stars in Figure 3, prevent any meaningful optimization of communications within *MapReduce* without sacrificing generality (see Section 5 for a discussion of previous work that reduced the *MapReduce* barrier but sacrificed generality to do so).

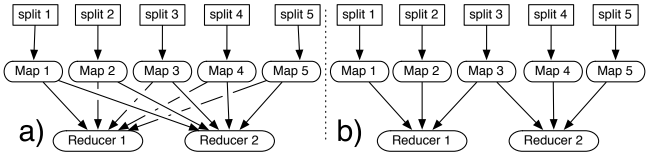


Figure 5: **a**: default communications pattern for the *MapReduce* query in Figure 4. **b**: actual data dependencies for the same query.

**Area 1**. In practice, the input to a RecordReader is frequently expressed as byte-ranges while the output is a set of key/value pairs where the key is in some logical keyspace. This mismatch prevents reasoning about relationships between the input and output without some external knowledge.

**Area 2**. *Map* tasks are user-defined functions. Consequently, there is no general method for predicting how *Map* input keys ($k$) translate into *Map* output keys ($k'$).

**Area 3**. Assigning intermediate key/value pairs to keyblocks represents a partitioning of the intermediate data keyspace ($K'$) where each subset (keyblock) will be assigned to a *Reduce* task. The design of *MapReduce* does not prescribe how this partitioning should be done, so it is regarded as an opaque process.

## 2.4 SciHadoop

SciHadoop [4] integrates structured data into Hadoop by:

1. extending Hadoop's input split generation process to utilize logical coordinates.
2. leveraging scientific metadata to make more informed decisions during input split generation.
3. defining a simple, array-based query language including an extraction shape that explicitly describes the units of data in the input that the specified operator will process together.

### 2.4.1 Expanding the Use of Logical Coordinates

As described in Section 2.3, Hadoop partitions the input into a set of $I_i$ that are each read by a RecordReader that emits key/value pairs for consumption by *Map* tasks. SciHadoop defines its $I_i$ in terms of logical coordinates, as opposed to byte-offsets, creating a situation where both RecordReader input and output are defined at the same level of abstraction and also in the same space (coordinates in the logical space $K$). In other other words, $I_i$ and the set of all keys that a RecordReader will produce when assigned $I_i$, denoted $\mathbb{K}_i^{\mathbb{T}}$, are equivalent in SciHadoop. It is worth noting that defining $I_i$ in terms of logical coordinates, rather than byte-ranges, complicates SciHadoop's attempts to create InputSplits with high rates of data locality. This point is addressed in our previous work [4].

### 2.4.2 Extraction Shape

The extraction shape, described in [4], is a concrete representation of the units of data that the operator, specified as part of the query, will be applied to. The extraction shape is logically tiled, in a given order, over $\mathbb{K}^{\mathbb{T}}$ with each instance representing a unique $k'$ key in $K'$. This process is dependent on the query being structural and is independent of the functions being applied in the *Map* and *Reduce* tasks.

As an example, consider a query over a 2-dimensional dataset that is down-sampled by taking every disjoint 2x2 region of the input data and outputting the average value of the 4 data points. In this example, the extraction shape is $\{2, 2\}$ (indicating every 2x2 input shape translates into a single element in the output). An example of this translation can be seen in Figure 6(b) as well as an extraction shape that represents an up-sampling (Figure 6a). Strided access (reading data at regularly spaced intervals) can be described by adding an additional $n$-dimensional array indicating the stride lengths between extraction shape instances.

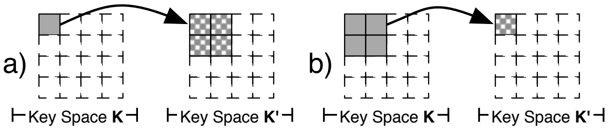


Figure 6: **a**: one value in K translating into four different values in K' **b**: Four different values in K translated into one value in K' due to a {2,2} extraction shape

For some structural queries, such as requesting all of the data for a given range of coordinates where the value exceeds a threshold, a list of zero or more results may be produced rather than a single data point.

## 3. SIDR

The goal of SIDR is to extend the *MapReduce* communications model for structural queries. This work is not specific to SciHadoop or scientific data, but rather is applicable to any combination of query and data where it is possible to reason about the relationship between input and output based on the inputs location in the dataset; a scenario that is common in scientific computing (Section 2.3.2).

SIDR uses structural knowledge of the query/data relationship to make stronger assumptions about data communications than *MapReduce* can and dynamically adjusts data assignment and task scheduling. To accomplish this without impinging on *MapReduce*'s generality, SIDR must overcome the three areas where *MapReduce*'s dataflow is opaque (Section 2.3.2).

**Area 1**. Given that SciHadoop already defines its $I_i$ in logical coordinates, rather than byte-offsets, $I_i$ and $\mathbb{K}_i^{\mathbb{T}}$ are equivalent. SIDR is able to (trivially) map an $I_i$ to the $\mathbb{K}_i^{\mathbb{T}}$ it will produce.

**Area 2**. For structural queries, the extraction shape, used to specify the units of data that an operator will be applied to in SciHadoop, can also be leveraged to map a key in $K$ to the corresponding key(s) in $K'$. In general, this mapping is accomplished by dividing each coordinate in the given key by the corresponding coordinate in the extraction shape. For example, consider the metadata for our temperature example (Figure 1) and assume a query that calculates weekly averages and also down-samples the latitude resolution from 1/10° to 1/2°, resulting in an extraction shape of {7, 5, 1}. Given that extraction shape, an arbitrary key in $K$, say {157, 34, 82}, maps to {22, 6, 82} in the $K'$ keyspace. SIDR uses this approach to translate the set of *Map* task input keys ($\mathbb{K}_i^{\mathbb{T}}$) to the keys that the *Map* task will emit ($\mathbb{K}'^{\mathbb{T}}_i$).

**Area 3**. Given that $\mathbb{K}^{\mathbb{T}}$ is readily available and keys in that space are easily mapped onto $\mathbb{K}'^{\mathbb{T}}$ via the extraction shape, the exact dimensions of the intermediate key space for a query ($\mathbb{K}'^{\mathbb{T}}$) can be computed. For a particular $\mathbb{K}^{\mathbb{T}}$ and extraction shape, $\mathbb{K}'^{\mathbb{T}}$ can be calculated by dividing the length of each dimension in $\mathbb{K}^{\mathbb{T}}$ by the entry in the corresponding dimension of the extraction shape. Since an extraction shape can map keys in $K$ in a one-to-many, one-to-one or many-to-one manner, $\mathbb{K}'^{\mathbb{T}}$ may be smaller, larger or the same size as $\mathbb{K}^{\mathbb{T}}$, respectively.

Again, consider the dataset in Figure 2 whose metadata is shown in Figure 1. Issuing this query over the {365, 250, 200} dataset with an {7, 5, 1} extraction shape (assuming we throw away the data from the 365-th day) results in {52, 50, 200} $\mathbb{K}'^{\mathbb{T}}$, representing 52 weekly measurements covering 25°N to 50°N in 1/2° increments and 85°W to 65°W in 1/10° increments.

## 3.1 partition+

Hadoop's default partition function assigns intermediate key/value pairs to KEYBLOCKs by taking the modulo value of the key's binary representation by the number of *Reduce* tasks, effectively partitioning the entire space that is representable by the data type of the key. Consequently, Hadoop's KEYBLOCK sizes are a function of the set of observed intermediate keys ($\mathbb{K}'^{\mathbb{T}}$) combined with the implementation specifics of the key data type.

By combining the solutions to the three opaque areas in *MapReduce*'s dataflow, SIDR is able to calculate the relationships between the sets $I$, $\mathbb{K}_i^{\mathbb{T}}$ and $\mathbb{K}'^{\mathbb{T}}_i$ based solely on information found in, or derived from, the query specification combined with the input metadata. This newfound knowledge is leveraged to construct an alternative partition function, partition+, that computes the set of intermediate keys that will actually exist ($\mathbb{K}'^{\mathbb{T}}$) and partitions that into $r$ KEYBLOCKs (where $r$ is the number of *Reduce* tasks).

Given that the $\mathbb{K}'^{\mathbb{T}}$ keyspace for a structural query is a fixed, or at least bounded, size, partition+ can create KEYBLOCKs of roughly the same size[1], as described here and shown in Figure 7. First, SIDR selects an upper bound on the permissible amount of skew (either user-defined as part of the query or chosen by the system based on the query), then creates an $n$-dimensional shape whose total size is smaller than that upper bound. The system then determines the total number of instances of the shape that exist in $\mathbb{K}'^{\mathbb{T}}$ and divides that by the number of *Reduce* tasks. The result is the size of each KEYBLOCK and they differ, at most, by one instance of the chosen shape which is itself less than the chosen amount of permissible skew. In practice, we allow the final partition to be smaller than the rest so that the other partitions consist of simpler shapes (making routing logic simpler) while also reducing the load on the last *Reduce* task.

Going a step further, since partition+ knows both the size and the contents of $\mathbb{K}'^{\mathbb{T}}$ for structural queries, SIDR can use the calculated KEYBLOCK shape to create KEYBLOCKs that are contiguous in K'. Since intermediate keys are often used to order output, contiguous blocks of keys in $K'$ often translate in contiguous keys in $\mathbb{O}^{\mathbb{T}}_{\text{es}}$ that should result in efficient writes to the underlying storage (assuming the scientific access library converts logically dense arrays into efficient file accesses). In contrast, SciHadoop uses Hadoop's default partition function, resulting in *Reduce* tasks being assigned K' keys that are spaced throughout $\mathbb{K}'^{\mathbb{T}}$. The performance impact of writing sparse vs dense output is explored in Section 4.4.

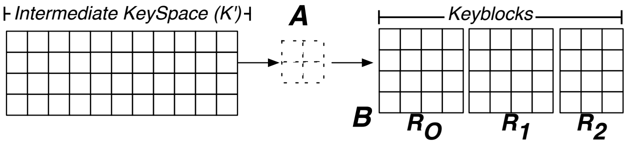


Figure 7: SIDR partitions the intermediate keyspace by (A) choosing a shape less than the permissible amount of skew, determining how many instances of that shape exist (*IntShapes*), and then (B) creating KEYBLOCKs consisting of $|IntShapes/\#Reducers|$ instances of said shape.

## 3.2 Minimizing data dependencies

$I_\ell$ is the set of $I_i$ that, when processed by a RecordReader and associated *Map* task, will produce at least one intermediate key/value pair that will be assigned to KEYBLOCK$_\ell$. SIDR uses the same logic as partition+ to determine *Reduce* task data dependencies ($I_\ell$) and *Reduce* tasks can use their $I_\ell$ as a barrier, rather than the set of all $I_i$ (global barrier), while maintaining correctness and without compromising *MapReduce*'s generality. Figure 4(b) shows *Reduce* task $R_1$ using its actual data dependencies.

The benefit that can be derived from determining $I_\ell$ for a KEYBLOCK is inversely related to the number of $I_i$ the KEYBLOCK depends on. In the extreme case, a KEYBLOCK depending on all $I_i$ must observe the global data dependency that Hadoop presently assumes. While Hadoop's modulo-based partition function typically creates global data dependencies (Section 3.4) while attempting to evenly distribute intermediate keys, partition+ provides a stronger guarantee of balanced key distribution while also maintaining any natural alignment between query and data (Figure 8).

### 3.2.1 Implementation Details

In the current implementation of SIDR, data dependencies are determined when a query begins by calculating which KEYBLOCKs each $I_i$ will generate data for and then inverting those relationships (the end result is a map from KEYBLOCKs to $I_i$). *Reduce* tasks are provided their dependency information when they are scheduled. This approach adds a small IO cost to job submission as the relationships are stored as part of the job specification. Alternatively, each *Reduce* task could calculate the set of $I_i$ that their assigned KEYBLOCK depends on when they start up (a classic "store vs re-compute" decision). A method for calculating $I_\ell$ from a KEYBLOCK is described in an earlier version of this work that was published as a tech report [3].

*Map* tasks often combine key/value pairs sharing the same key in an effort to reduce disk and network IO. The ability

[1] Accepting a small amount of skew to create KEYBLOCKs of simpler shapes can result in more efficient communications and reduced data dependencies between tasks

to calculate $\mathbb{K}_\ell^\mathbb{T}$ is requisite for maintaining correctness when starting *Reduce* tasks early in the case where multiple elements in $K$ map to a single element in $K'$ (Figure 6(b)). The set of keys in $K$ that map to the same point in $K'$ may exist in different $I_i$ (generating multiple $\langle k', v' \rangle$) or the same $I_i$ (generating a single $\langle k', v' \rangle$). Since the *Reduce* task does not know how many $\langle k, v \rangle$ were combined to produce a given $\langle k', v' \rangle$, it cannot begin processing after receiving a particular $\langle k', v' \rangle$ without risking the production of an answer based on insufficient input. This is a significant issue, as a pessimistic solution would require reverting back to using a global barrier.

Resolving the ambiguity of how many $\langle k, v \rangle$ a particular $\langle k', v' \rangle$ represents is accomplished by either:

1. Calculating $I_\ell$, the set of $I_i$ that will produce data destined for KEYBLOCK$_\ell$, and using that as a proxy for $\mathbb{K}_\ell^\mathbb{T}$, since it is a super-set or,
2. Annotating each $\langle k', v' \rangle$ pair to include the number of $\langle k, v \rangle$ pairs it represents. Each *Reduce* task can then keep a running tally of the number of $\langle k, v \rangle$ represented by the set of $\langle k', v' \rangle$ it receives. When the task has accumulated data representing all $\langle k, v \rangle$ in its $\mathbb{K}_\ell^\mathbb{T}$, processing can safely begin.

SIDR uses the former method and also implements the annotations required for the latter method as a means of validating the system's correctness. Approach 1 is simply a matter of generating dependency information in terms of input splits (rather than specific intermediate key ranges). Approach 2 requires the addition of a field to the header for each *Map* output file that indicates how many $\langle k, v \rangle$ are represented by the set of all $\langle k', v' \rangle$ in that file. With this addition, a *Reduce* task can track the count of how many $\langle k, v \rangle$ are represented by the contents of the files containing its intermediate data without having to read and parse those files. This is an efficient means of enabling *Reduce* tasks to partially understand their data at the logical level, as opposed to their current state of ignorance prior to the start of the *Reduce* phase.

## 3.3 Altering task scheduling in Hadoop

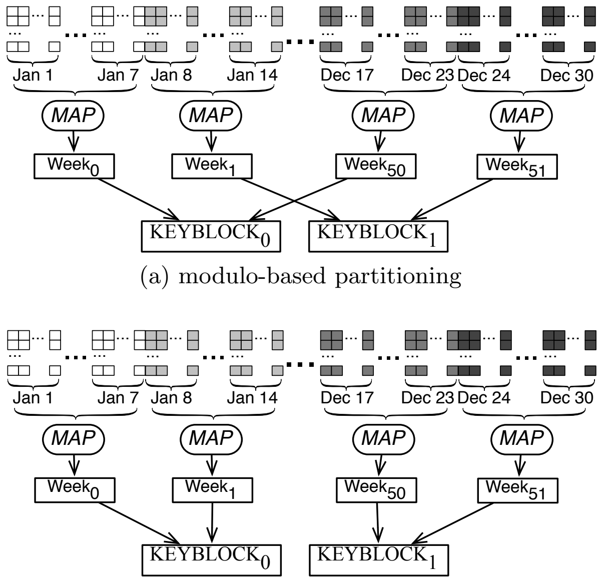
At the onset of a Hadoop query, all *Map* tasks are added to a tree structure representing different levels of data locality. When a server requests a new *Map* task, that tree is traversed, starting at the leaf-node that contains $I_i$ local to that server with nodes at higher levels containing increasingly less local $I_i$, and the first runnable task encountered is returned. In contrast, *Reduce* tasks are scheduled in monotonically increasing order of their IDs. The interaction between the two scheduling polices results in inter-task dependencies being met haphazardly since no attempt is made to co-schedule *Reduce* tasks with the *Map* tasks they depend on. In practice, a *Reduce* task has its data dependencies met probabilistically.

SIDR inverts this process by scheduling *Reduce* tasks first with *Map* tasks only becoming eligible to be scheduled if at least one *Reduce* task that depends on it is already running. Whenever a *Reduce* task is scheduled, the same tree structure is crawled and all *Map* tasks that contribute to the *Reduce* task are marked as schedulable. This adds a constant cost to task scheduling of 2 pointer dereferences per *Map* / *Reduce* dependency, which we view as negligible.

This change in scheduling does introduce an additional delay prior to *Map* tasks starting (specifically the time required to find an available *Reduce* slot and schedule a task). In practice, SIDR's ability to start *Reduce* tasks early (resulting in those tasks ending early) translates into more than one *Reduce* slot typically being available when a query starts. This diminishes the delay introduced by SIDR to the time taken for Hadoop to schedule a task.

## 3.4 Benefits of the Enhanced Model

Structural queries frequently align with ordering properties inherent in data stored in scientific file formats. Returning to the example of down-sampling a temperature dataset from daily measurements to weekly averages, we see how the daily data points that map to the first week are at the beginning of the file while days that map to the last week are at the end of the file. A modulo-based approach (Figure 8(a)) will result in both KEYBLOCKs being dependent on $I_i$ spread throughout the dataset while partition+ assigns logically contiguous ranges of $I_i$ to KEYBLOCKs, exposing any natural alignment between structural queries and the dataset. In Figure 8(b), this results in KEYBLOCK$_0$ only depending on $I_i$ located in the first half of the dataset.



(a) modulo-based partitioning

(b) partition+

Figure 8: Two different approaches to partitioning intermediate keys for a down-sampling; (b) maintains ordering properties while (a) ignores them.

SIDR's ability to calculate $I_\ell$ coupled with the order-preserving properties of partition+ enables the more precise communications model depicted in the bottom portion of Figure 5. This new model's effect on scheduling is shown in Figure 4(b) where *Reduce* task $R_1$ is allowed to start prior to *Map* tasks $M_4$ and $M_5$ completing because the framework has already determined that those *Map* tasks will not produce any data for $R_1$. A *Reduce* task's ability to begin processing data earlier enables additional overlapping of computation with IO beyond that already present in *MapReduce*, which previous research [5, 32] indicates will translate into shorter query run-times.

Since SIDR schedules *Reduce* tasks, with *Map* tasks becoming eligible to be scheduled as a side-effect, it can prioritize portions of the output space by scheduling those KEYBLOCKs first. This functionality is interesting in a few contexts. Firstly, computational steering [31] uses the output of one experiment to choose the parameters for subsequent runs. In this scenario, if the user believes that a certain portion of the output would likely contain the salient result(s), those KEYBLOCKs can be scheduled first, as opposed to waiting for them to be scheduled organically. This prioritization could shorten the analysis cycle and result in more efficient use of computational resources. Secondly, the recently proposed burst buffer architecture [23] presents an opportunity for in-situ processing on SSD-based data staging nodes prior to the data being written to a disk-based storage system. In this scenario, compute resources are not guaranteed and data may be evicted at any point. Given this tenuous access to data on a fast medium, the ability to prioritize the processing of certain portions of the data allows the scientist to better capitalize on their window of opportunity.

# 4. EVALUATION

**Experimental Setup.** Our experiments were conducted on a cluster of 25 nodes, each with two 2.0GHz dual-core Opteron 2212 CPUs, 8GB DDR-2 RAM, four 250GB Seagate 7200-RPM SATA hard drives, running Ubuntu 12.04. SIDR built upon the SciHadoop code [12] that we ported to Hadoop 1.0. Our Hadoop cluster has a single node acting as both the NameNode and JobTracker while the other 24 nodes serve as both DataNodes and TaskTrackers. The 24 DataNode/TaskTracker nodes use one hard drive for the OS, supporting libraries, and temporary storage, while the other 3 hard drives are dedicated to HDFS. All nodes have a single Gigabit network connection to an Extreme Networks' Summit 400 48-t switch. HDFS is configured with 3x replication and 128 MB block size. Each TaskTracker is configured for 4 *Map* and 3 *Reduce* task slots.

## 4.1 Early Results

**Query 1** applies a median function over a 4-dimensional dataset of sizes $\{7200, 360, 720, 50\}$ with an extraction shape of $\{2, 36, 36, 10\}$. The dataset is intended to represent 300 days worth of hourly windspeed measurements at a resolution of 0.5° Longitude by 0.5° Latitude at 50 different elevations with the query representing finding a median value, over 2 consecutive days, for each 18° Longitude by 36° Latitude region, in cross-sections of 10 elevation steps. This function was chosen as a representative structural query.

We first compare Hadoop, SciHadoop and SIDR with 22 *Reduce* tasks (best practices dictate that 90% of the node count as a reasonable number of *Reduce* tasks in systems with low odds of preemption). Figure 9 shows *Map* and *Reduce* task completion over time. SIDR starts producing results around 625 seconds while SciHadoop's first result arrives just after 1,132 seconds and Hadoop's first result coming at 2,797 seconds. The difference in the slopes of both *Map* and *Reduce* tasks between Hadoop and SciHadoop owe to the efficiencies gained by intelligent input split generation and data locality enabled by SciHadoop. The gap between the first result in SIDR and the first results in SciHadoop and Hadoop is a result of SIDR using the actual data dependencies present in the data being processed. The query executing with SIDR completes at 1,264 seconds while SciHadoop completes slightly sooner, at 1,250 seconds.

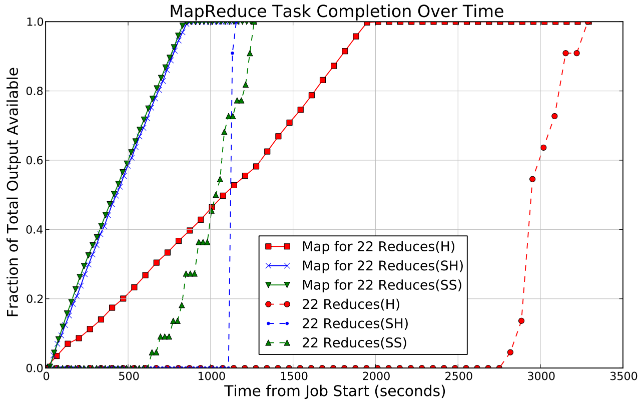


Figure 9: *Map* and *Reduce* task completion for the same query run with Hadoop(H), SciHadoop(SH) and SIDR (SS)

SIDR's slightly longer run-time is a by-product of partition+ creating contiguous KEYBLOCKs. SIDR's last *Reduce* task must copy, merge and process all of the data in the last 4.5% (1/22nd) of the *Map* tasks. In SciHadoop, the output of those *Map* tasks is spread evenly across all *Reduce* tasks.

Next, we present the same query and dataset while varying the number of total *Reduce* tasks (Hadoop results omitted in order to show a finer resolution and *Map* tasks are omitted as they do not vary from Figure 9). Increasing the number of *Reduce* tasks results in each being assigned a smaller amount data, which will reduce its data dependencies. Figure 10 shows results for SIDR and SciHadoop with 22 *Reduce* tasks as well as 66, 176, and 528 *Reduce* tasks for SIDR (given the dataset size (348 GB) and HDFS's block size (128 MB), the SciHadoop partitioning scheme creates 2,781 input splits for this query). As the number of *Reduce* tasks increases, time to first result and total execution time both decrease with SIDR. At 528 *Reduce* tasks, SIDR finishes 29% faster than with SciHadoop and nearly three times faster than Hadoop (Figure 9). These performance improvements are attributable to SIDR's increased ability to overlap computation with IO. Since Hadoop and SciHadoop are beholden to the global *MapReduce* barrier, increasing the number of *Reduce* tasks for either yields no benefit; the same amount of work is distributed to more tasks but none of them can start prior to the last *Map* task completing.

Since a *Reduce* task cannot begin processing its data until all of its data dependencies have been satisfied, the ideal graph for the *Reduce* tasks would be a line that paralleled the graph for *Map* task completion, shifted to the right by the average amount of time it took a *Reduce* task to process its assigned data. In Figure 10, the *Reduce* task completion line approaches the *Map* task completion line as the number of *Reduce* tasks increases, with 528 *Reduce* tasks coming close to optimal. Note that since each additional *Reduce* task adds a small, fixed overhead to the query, increasing the number of *Reduce* tasks past a certain (query-specific) point is detrimental to performance.

**Query 2** uses a dataset of the same size, $\{7200, 360, 720, 50\}$, with normally distributed values and a filter query that returns only values more than three standard deviations greater than the mean (i.e., 0.1% of the total dataset or 93.31 million values of the 93.31 billion in the dataset). Query 2 uses an extraction shape of $\{2, 40, 40, 10\}$ out of

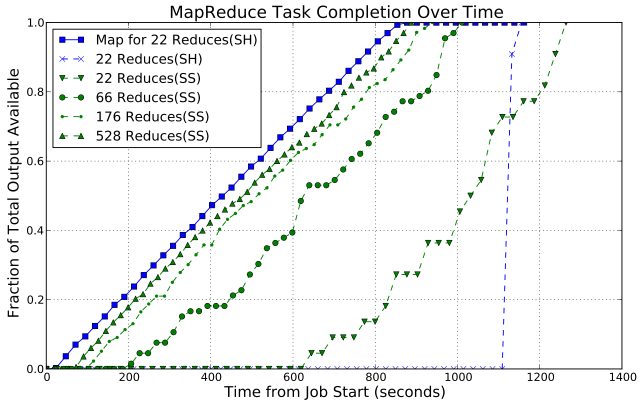


Figure 10: *Reduce* task completion for a fixed query run with Hadoop using 22 *Reduce* tasks, SciHadoop using 22 *Reduce* tasks and SIDR running 22, 66, 176 and 528 *Reduce* tasks.

convenience; results will contain a list of all values greater than the threshold.

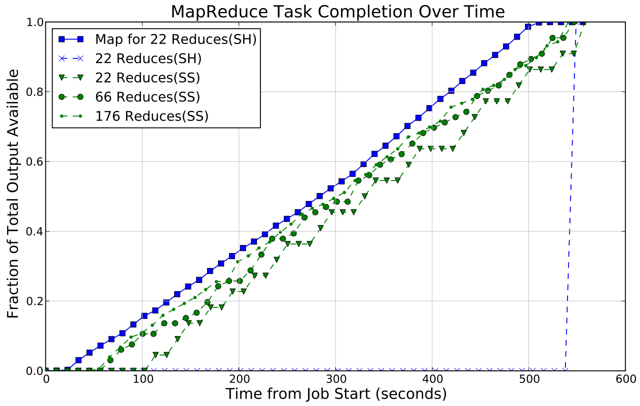


Figure 11: *Reduce* task completion for a filter query run via SciHadoop with 22 *Reduce* tasks and SIDR with 22, 66 and 176 *Reduce* tasks.

Figure 11 shows the results for Query 2 on SciHadoop with 22 *Reduce* tasks and SIDR with 22, 66 and 176 *Reduce* tasks (the *Map* task results for 22 *Reduce* tasks are included for reference). Query 2 processes far less data in each *Reduce* task than Query 1. As a result, each *Reduce* task finishes quicker with that server becoming available to process another *Reduce* task promptly, resulting in the *Reduce* task completion lines approaching optimal with fewer total tasks than Query 1. Also, since the *Reduce* tasks represent such a small fraction of total query execution time to start with (indicated by the slope of the *Reduce* graph for SciHadoop with 22 *Reduce* tasks), there is little room for SIDR to improve the query, and the reduction in total query time is much smaller than it was for Query 1.

By comparing the results of the two queries on data of the same size, it is evident that the nature of the query has a direct effect on the number of *Reduce* tasks required to approach optimal performance as well as the maximum opportunity for improvement by SIDR.

## 4.2 Variance in Reduce Completion Times

Figure 12 shows task variance for *Map* tasks, and both 22 and 88 *Reduce* tasks with SIDR (values are averages from 10 runs with error bars indicating the standard deviation for each data point). With SIDR, data dependencies are small(er) barriers, so *Reduce* tasks display at least as much variance as the set of *Map* tasks they depend on. Increasing the number of *Reduce* tasks diminishes that set (per *Reduce* task) and the probability of a *Reduce* task depending on several abnormally long-running *Map* tasks.

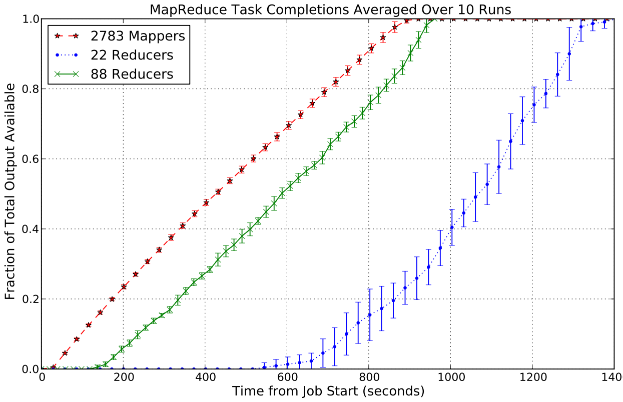


Figure 12: Variance in SIDR task completion times as the *Reduce* task count varies.

## 4.3 Intermediate Key Skew

The intermediate keys for structural queries often display a pattern (i.e., coordinates at fixed intervals). This can create situations where the binary representations of those keys are in turn patterned and therefore do not distribute evenly when assigned to *Reduce* tasks (Section 3.1). For example, we've seen cases where every intermediate key was even, resulting in all odd-numbered *Reduce* tasks being assigned no data to process while their even-numbered counterparts receive twice as much data as expected. Consequently, the lightly loaded *Reduce* tasks finish very quickly while overloaded tasks take much longer. Figure 13 displays the task completion graphs for a query that exhibits this behavior. For the same query, SIDR evenly distributes the work and completes 42% faster.

## 4.4 Contiguous Output

Scientific data is typically stored in dense arrays where the coordinates of individual points are relative to the origin of that dense array and their global position, if part of a larger dataset, is inferred from that origin point. Since the partition+ function creates KEYBLOCKS that are both balanced and contiguous (Section 3.1), it naturally produces data in this manner. Table 2 shows the results of a micro-benchmark that simulates a single *Reduce* task writing out NetCDF data. For the experiment, we fix the amount of data written per task and then scale the total amount of data written (doubling both data and simulated task count at each step). Both the time required and resultant output file size are shown (times are 10 run averages with standard deviations in parenthesis). The bottom entry in Table 2 (SIDR) presents a single *Reduce* task writing a contiguous portion of a larger output. As the number of *Reduce* tasks

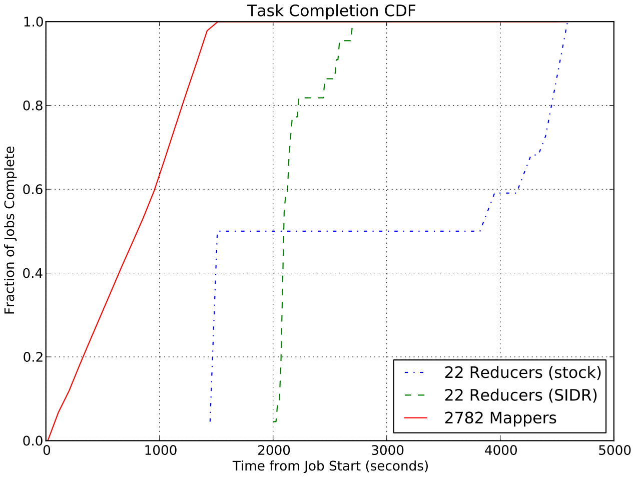


Figure 13: Intermediate data skew caused by Hadoop's partition function results in longer query execution time. SIDR even distributes the data and completes 42% faster.

Table 2: Individual Reduce Write Time and Size Scaling

| Hadoop Reduce Write Scaling | | |
|---|---|---|
| Total Reduces | Avg Time in Seconds (Std Dev) | Output Size (MB) |
| 20 | 6 (.6) | 494 |
| 40 | 11.4 (.9) | 988 |
| 80 | 24.2 (3.2) | 1976 |
| SIDR Reduce Write Scaling | | |
| * | 0.3 (.02) | 24.8 |

and total output are scaled, the representative *Reduce* task writes the same amount of data in the same amount of time for each experiment.

Since Hadoop's modulo-based partition function does not create dense, contiguous KEYBLOCKs, it must take a different approach to writing out scientific data. A common method for writing sparse data is to create a file representing the entire space and using sentinel values for absent data. This approach creates a few issues in the context of *MapReduce*. Firstly, the size of the file written by each *Reduce* task is the size of the total output. Increasing the number of *Reduce* tasks will increase the total number of bytes written for the same amount of useful data. This discourages increasing the number of *Reduce* tasks, which conflicts with our results. Secondly, the time required for a *Reduce* task to write its data will increase along with the number of *Reduce* tasks due to larger seeks between writes. Thirdly, the files are not very useful individually and will likely need to be merged later, requiring extra data movement. The entries in Table 2 for Hadoop show that the sentinel value approach does not scale well.

Another method for storing sparse data is to store coordinate / value pairs. This approach creates storage overhead since both the data and coordinate are explicitly stored, rather than the coordinate being implicit (as is normally the case), but that overhead is a constant scalar relative to the amount of useful data and independent of the number of *Reduce* tasks.

Table 3: Network Connection Scaling

| Hadoop Reduce Write Scaling | | |
|---|---|---|
| Map / Reduce Count | Hadoop (# Connections) | SIDR (# Connections) |
| 2781/22 | 61,182 | 2,820 |
| 2781/66 | 183,546 | 2,905 |
| 2781/132 | 367,092 | 3,031 |
| 2781/264 | 734,184 | 3,267 |
| 2781/528 | 1,468,368 | 3,760 |
| 2781/1024 | 2,936,736 | 5,106 |

### 4.5 partition+ performance

Partitioning of intermediate data occurs in-line with *Map* task execution and therefore cannot be effectively measured within Hadoop. We created a micro-benchmark that measures only the total time required to partition key/value pairs in order to understand the performance impact of our partition+ function. The benchmark loads 6.48M intermediate key/value pairs (meant to represent *Map* task output) into memory and applies a given partitioning function, measuring only the time required to partition the data. Over ten runs, the default partition function took an average of 200 ms ($\sigma$ 18.8 ms) to partition the entire set of key/value pairs while partition+ averaged 223 ms ($\sigma$ 21 ms). While partition+'s slightly (23 ms) slower partitioning has a negligible impact on total *Map* task run-time, given *Map* task execution times range from tens of seconds to tens of minutes, we are looking into optimizing partition+.

### 4.6 Increased Network Efficiency

Table 3 shows how the total number of network connections between *Map* and *Reduce* tasks scales for Query 1 as the number of *Reduce* tasks increases. SIDR's efficiency results from each *Reduce* task only contacting *Map* tasks that produced data assigned to its KEYBLOCK (all other *Map* tasks are ignored). Hadoop requires that every *Reduce* task contact every completed *Map* task. Additionally, Hadoop's limit on the number of concurrent connections (10 per *Reduce* task by default), can create an undesirable serialization of communication in some scenarios. SIDR's efficient communication model diminishes the likelihood of serialization occurring.

## 5. RELATED WORK

The popular MapReduce Online [5] paper explored *online aggregation* in *MapReduce* with their Hadoop Online Prototype (HOP). HOP altered Hadoop's execution model by starting all *Reduce* tasks at the beginning of a query and having *Map* tasks transfer their output directly to *Reduce* tasks where it is immediately incorporated into running estimates of the final output. HOP emits those estimates at fixed intervals (25%, 50%, 75% and 100% of the data processed). This type of early results is useful for gaining a high-level intuition about the final result but is not well-suited to the use cases we identified in Section 3.4. HOP is limited to distributive operators or algorithms that produce approximate results for non-distributive functions, resulting in a solution that is less general than Hadoop. Any subsequent computations that consume HOP's output must be re-run after each estimate is emitted, resulting in increased resource usage. The authors identify several resource requirements and state that HOP devolves to Hadoop if any of those are not met, thereby limiting its applicability. Another project [32] built upon HOP by extending the original in-memory data structures to spill to disk if needed, extending this approach larger datasets than are possible with the original HOP approach. For structural queries, SIDR produces correct, partial results, rather than estimates, meaning that subsequent queries do not need to be re-executed. Also, SIDR does not have HOP's memory-based limitation nor is it limited to distributive functions.

Systems that convert scientific data their own formats [30, 19, 1, 24] do not face the same issues as SIDR because they have complete control, and therefore complete knowledge, of the data being processed. Consequently, efficient data partitioning and communication is easier to achieve. This approach has its drawbacks: data ingest requires time and extra data movement (with the associated resource costs) and access to the data in its native format is lost (along with tool sets built around those formats). Also, the efficiencies typically provided by these systems rely on an ideal data layout being specified ahead of time. Queries that are not aligned to the predefined access patterns do not benefit, resulting in this approach providing limited benefits to ad hoc queries (a common class of *MapReduce* programs).

SciDB [2], an array-based analysis platform designed specifically for scientific data, currently requires data to be ingested and stored in its internal format but they aspire to eventually support in-situ processing of scientific data. It is unclear how their query processing model will interact with non-ingested data.

SIDR is complementary to more expressive computation models, such as Pregel [25], Dryad [18] and Spark [37], as they can leverage the methods presented in this paper within their (structural) sub-computations. While Dryad is capable of supporting a wider range of communication patterns than *MapReduce*, it uses the same fixed pattern for all instances of a vertex whereas SIDR makes task (vertex instance) specific routing decisions. Dryad's support for data-informed communication appears to be limited to sampling [36], which is not possible for structural queries over scientific data because the access libraries obscure the layout of the data [4].

The Sailfish project [27] altered how intermediate data was handled in Hadoop and, by postponing the assignment of KEYBLOCKs until all intermediate keys were produced, eliminated key skew by partitioning that set, rather than using a modulo-based approach. This design strengthens the global *MapReduce* barrier because *Reduce* tasks can no longer overlap the acquisition of their assigned data with the completion of other *Map* tasks. For structural queries, SIDR eliminates key skew without strengthening the global barrier (the barrier is actually weakened). There are other types of skew in *MapReduce*, such as similarly sized datasets consuming vastly different amounts of computational resources. In these cases, an approach like SkewTune [21] can be used to complement SIDR.

## 6. CONCLUSIONS AND FUTURE WORK

This paper presents SIDR, an extension of *MapReduce* for structural queries that: intelligently partitions intermediate data; derives actual data dependencies between *Map* and *Reduce* tasks; and schedules tasks based on that knowledge. SIDR can produce prioritized, correct results for portions of the output space with only a fraction of the input processed. Total query run-time and task run-time variance are reduced, intermediate key skew is prevented and the resultant output is organized into balanced, contiguous datasets. SIDR accomplishes all of this while maintaining the generality of Hadoop and the *MapReduce* model. These developments significantly improve our work's ability to enable scientists to use the Hadoop platform while maintaining access to their data in the desired format.

Building upon SIDR, we plan to investigate altering the *MapReduce* failure recovery model to use the data dependency information to re-execute subsets of *Map* tasks in the event of a *Reduce* task failure in place of persisting all intermediate data to disk. Our hypothesis is that the performance savings in the non-failure case will offset said re-execution cost. Additionally, we will research integrating SIDR's ability to produce early, orderable, correct results for portions of the total output into pipe-lined computations.

## Acknowledgments

We would like to thank our collaborators: Meghan (Wingate) McClelland, Gary Grider, and James Nunez at Los Alamos National Laboratory; Maya Gokhale at Lawrence Livermore National Laboratory; John Bent at EMC; Russ Rew at Unidata; 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.

## References

[1] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. *SIGOD Rec.*, pages 575–577, June 1998.

[2] P. 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.

[3] J. Buck, N. Watkins, G. Levin, A. Crume, K. Ioannidou, S. Brandt, C. Maltzahn, and N. Polyzotis. SIDR: Efficient Structure-aware Intelligent Data Routing in SciHadoop. Technical Report UCSC-SOE-12-08, UCSC, 2012.

[4] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[5] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10. USENIX Assn., 2010.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, Berkeley, CA, USA, 2004.

[7] E. Dede, M. Govindaraju, D. Gunter, and L. Ramakrishnan. Riding the elephant: managing ensembles with hadoop. In *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, MTAGS '11, pages 49–58, New York, NY, USA, 2011. ACM.

[8] J. Ekanayake, T. Gunarathne, G. Fox, A. Balkir, C. Poulain, N. Araujo, and R. Barga. DryadLINQ for Scientific Analyses. In *e-Science, 2009. Fifth IEEE International Conference on*, pages 329 –336, dec. 2009.

[9] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *eScience, 2008. IEEE Fourth International Conference on*, dec. 2008.

[10] M. Felice Pace. BSP vs MapReduce. *ArXiv e-prints*, Mar. 2012.

[11] FITS Homepage. `http://fits.gsfc.nasa.gov/`.

[12] SciHadoop source code on github.com. `https://github.com/four2five/SciHadoop/`.

[13] GRIB Homepage. `http://www.ecmwf.int/products/data/software/grib_api.html`.

[14] R. L. Grossman, M. Sabala, Y. Gu, A. Anand, M. Handley, R. Sulo, and L. Wilkinson. Discovering Emergent Behavior From Network Packet Data: Lessons from the Angle Project.

[15] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. MapReduce in the Clouds for Science. *Cloud Computing Technology and Science, IEEE International Conference on*, 0, 2010.

[16] Hadoop Homepage. `http://hadoop.apache.org/`.

[17] HDF5 Homepage. `http://www.hdfgroup.org/HDF5/`.

[18] M. Isard and Y. Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09. ACM, 2009.

[19] M. Ivanova, N. Nes, R. Goncalves, and M. Kersten. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, SSDBM '07, 2007.

[20] H. Karloff, S. Suri, and S. Vassilvitskii. A Model of Computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, 2010.

[21] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in MapReduce Applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[22] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: A Method for Solving Graph Problems in MapReduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, 2011.

[23] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1 –11, april 2012.

[24] S. Loebman, D. Nunley, Y.-C. Kwon, B. Howe, M. Balazinska, and J. 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*, 31 2009-sept. 4 2009.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, 2010.

[26] NetCDF Homepage. `http://www.unidata.ucar.edu/software/netcdf/`.

[27] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: a framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM.

[28] Appendix: Astronomy in ArrayDB.

[29] E. Soroush and M. Balazinska. Hybrid merge/overlap execution technique for parallel array processing. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 20–30, New York, NY, USA, 2011. ACM.

[30] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, 2011.

[31] R. Van Liere, J. D Mulder, and J. Van Wijk. Computational Steering. *Future Generation Computer Systems*, 12(5):441–450, 1997.

[32] A. Verma, N. Zea, B. Cho, I. Gupta, and R. Campbell. Breaking the MapReduce Stage Barrier. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, sept. 2010.

[33] J.-S. Vöckler, G. Juve, E. Deelman, M. Rynge, and B. Berriman. Experiences using cloud computing for a scientific workflow application. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, pages 15–24, New York, NY, USA, 2011. ACM.

[34] J. Wang, D. Crawl, and I. Altintas. Kepler + Hadoop: A General Architecture Facilitating Data-Intensive Applications in Scientific Workflow Systems. In *SC-WORKS*, 2009.

[35] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats. CCGrid '12. ACM, 2012.

[36] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[38] C. Zhang, H. Sterck, A. Aboulnaga, H. Djambazian, and R. Sladek. Case study of scientific data processing on a cloud using hadoop. In D. Mewhort, N. Cann, G. Slater, and T. Naughton, editors, *High Performance Computing Systems and Applications*, volume 5976 of *Lecture Notes in Computer Science*, pages 400–415. Springer Berlin Heidelberg, 2010.

[39] H. Zhao, S. Ai, Z. Lv, and B. Li. Parallel Accessing Massive NetCDF Data Based on MapReduce. In *Web Information Systems and Mining*, Lecture Notes in Computer Science. 2010.