UNIVERSITY OF CALIFORNIA
SANTA CRUZ

EXTENDING MAPREDUCE FOR SCIENTIFIC DATA

A thesis submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Joe Buck

June 2014

The Thesis of Joe Buck
is approved:

Professor Scott Brandt, Co-Chair

Professor Carlos Maltzahn, Co-Chair

Professor Neoklis Polyzotis

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The volume of data being generated globally is increasing at a dramatic rate. This phenomenon applies to many classes of data, including scientific data. As an example, LLNL's Green Data Oasis hosted a 45 Terabyte(TB) data set of CMIP-3 [26] multi-model data in 2010 and the CMIP-5 model contains 3,000 TB of model data [132].

The dramatic increase in the amount of data that is being stored and analyzed has given rise to the industry of "big data". A diverse ecosystem of frameworks and methodologies now exist for processing large amounts of data via clusters of commodity computers. The various systems currently in existence provide a myriad of options in terms of performance, latency, resiliency and scalability. However, most of these systems do not support a wide variety of types of data. Existing research in scalable processing has mostly focused either on unstructured data (web pages, unstructured text, network traffic logs) [30, 63, 161] or structured data (fixed-size records, or data in column or row-based storage systems (databases)) [133, 23, 57, 53].

Systems built for unstructured data make no assumptions about the data being processed, necessitating pessimistic assumptions when scheduling different portions of the computation. Conversely, systems imposing structure on their data can make assumptions during the processing of the data, since they possess some degree of knowledge about the data content and ordering. However, this later approach requires that the data is re-organized and formatted during ingest into the system so that the data adheres to the system's proscribed structure. The requisite processing represents a cost, both in terms of time and resource usage. Additionally, systems that store data in a structured format typically only support data access via query interfaces, such as SQL — the data is no longer accessible via a file interface.

Scientific data is somewhat of an amalgamation of structured and unstructured; often structured in nature but stored in file formats that obscure this structure. These formats present a binary byte-stream to the underlying storage system while exposing the data to applications via a logical model. Scientific file formats often store fixed-size fields that are aligned to fairly rigid data models (e.g., dense arrays, dense spheres) but some formats do allow for less well-defined data models. Scientific file formats have several aspects in common: they have been developed over time, vary by discipline and are the *linqua franca* of their respective communities. Entire ecosystems of tools, themselves often developed over decades, exist for each file format and represent an investment of time and resources that precludes wholesale abandonment of these formats.

User-defined record-based file formats, such as Avro [7] and protobuf [105], bear a resemblance to the scientific file formats just described. However, scientific formats are better thought of as storing a single, well-defined dataset whose records are consistently sized and ordered where as Avro and protobuf typically store an unordered sequence of well-defined, but possibly irregularly shaped, records

In this thesis, we delve into the unique properties of structured data stored in scientific (binary) file formats and then utilize that knowledge to build an efficient system for processing scientific data. Our research is realized by augmenting an existing parallel computing framework (Hadoop). This approach allows us to leverage existing work on scalable, parallel computing without demanding the abandonment of the existing file formats that scientists use. The resultant system presents scientists with the means to analyze the growing volume of data that they are presented with.

The main accomplishments, with specific contributions noted, are:

1. Identification of the challenges inherent in processing data in scientific file formats

    (a) Identified the difficulties in achieving data locality when processing scientific data in its native file format.

    (b) Demonstrated that the appropriate level of abstraction for partitioning the input to a query is the logical level, rather than the physical level, as is currently done in Hadoop.

    (c) Proposed three approaches to logically partitioning scientific data and presented experimental results showing the degree to which each approach provides data locality.

2. Examination of the assumptions and design decisions in an existing parallel computing framework, and the identification of approaches to processing scientific data that maintain those decisions

   (a) Showed that partitioning the input data logically enables tasks to read only the data that will contribute to the query. This stands in contrast to reading all of the data and filtering out the non-useful elements in the application logic, which must be done if data is partitioned physically.

   (b) Described how Combiners (an optional component in a *MapReduce* program) can integrate structural knowledge to enable their use in situations where they otherwise would not be applicable, thereby leading to reduced data movement and performance improvements.

3. A detailed explanation of how those designs can be altered, given knowledge of the structure of the data being processed, in a manner that improves aspects of the framework without sacrificing its generality

   (a) Formalized the internal data communication logic of Hadoop and identified two points where inputs and outputs of specific steps in that process cannot be correlated.

   (b) Designed and implemented a structure-aware intermediate data partitioning function, called hash+, that replaces the default structure-oblivious partitioning function. This new function enables performance improvements and the ability for Hadoop to better reason about data flow during the execution of a *MapReduce* job.

   (c) Applied the new found ability to better reason about the flow of data throughout a *MapReduce* program to derive the actual data dependencies of *Reduce* tasks. This additional knowledge enables *Reduce* tasks to begin executing once their data dependencies are met, leading to performance gains as well as the production of early results.

   (d) Altered the Hadoop scheduler to temporally co-schedule *Map* tasks with the *Reduce* tasks that depend on them, improving performance and time to first result without negatively impacting data locality or substantially increasing scheduler complexity.

   (e) Modified Hadoop such that intermediate data is transferred directly to the memory of the *Reduce* task(s) it is assigned to rather than persisting it to local, stable storage. Presented associated experiments that quantify the performance benefit achieved in doing so.

   (f) Altered *MapReduce*'s fault-tolerance mechanism to use the actual data dependencies (as opposed to the normally assumed global data dependencies) of a *Reduce* task when scheduling jobs for re-execution in response to a *Reduce* task failure.

   (g) Conducted experiments to determine whether the performance gains realized by foregoing the persisting of intermediate data coupled with the more precise fault-recovery scheduling for *Reduce* tasks presents an attractive alternative to Hadoop's current failure recovery model.

4. Revisit framework-specific optimizations presented in previous research and show their efficacy in the face of our changes to the framework

   (a) Adapted an in-memory implementation of an otherwise disk-based process to our framework and report the measured efficacy.

Chapter 2

Motivation

This section contains a more detailed explanation of the motivation for this work. Several different ideas are touched on here, to set the stage for later sections, and a more thorough treatment of related projects is given in Section 9.

While it is clear that the analytics systems used by scientists will need to scale to larger data sets, the ability of current scientific analytical tools to do so is unclear, bordering on doubtful. Many of the current suites of tools used by scientists are file format specific and run serially on a single computer [88, 21, 44, 84]. Some attempts have been made to parallelize existing serial tools, such as the SWAMP project [144], but these typically execute serial scripts in parallel, with no efforts made in regards to global scheduling, inter-process communication or resiliency. Given the significant fundamental work that would be required to parallelize an existing scientific analysis framework, we instead look towards existing parallel processing frameworks and consider what would be required to extend them for scientific data.

2.1 File Management

The topic of file management must be addressed when considering how best to approach the topic of processing large-scale scientific data. This thesis is predicated on the contention that processing systems should not place limitations on either the number or size of files containing data to be processed. A system that does place limitations (say, by enforcing a maximum file size) implicitly requires that data not conforming to these restrictions must be reorganized to do so. This would require data movement, with the associated costs in both time and resources, which is undesirable as data movement is already projected to be a major cost in running systems as data sizes grow from peta- to exascale [113].

From a file management perspective, it would be the most convenient, and less error-prone, to manage a dataset as a single entity. Interacting with a single data set spread across multiple files places a burden on the meta-data of the storage system and requires extra diligence when interacting with the files themselves (so as to not accidentally interact with a subset, thereby introducing errors). Conversely, writing to a storage system via a single file can cause significant performance issues, due to misalignments between write boundaries and either files locks or file system block boundaries [111].

The authors of PLFS [11] quantified these issues and strove to provide the IO benefits of writing to many files while presenting a single-file to the application. Solving the issue of balancing file management inconvenience against storage system performance is beyond the scope of this thesis. As we will not solve this issue, our system will work with datasets contained in any number of files, deferring decisions about data set construction to the user.

2.2 Databases

Research into using databases, in this case defined as classic database management systems (e.g., MySQL, SQL Server), to solve the issue of processing large scientific data sets is currently underway within other organizations. Databases typically require data be ingested into the system prior to allowing users to conduct analysis on it, which incurs the same data movement and reorganization costs previously mentioned. While restructuring data can yield performance benefits for later queries, since assumptions can then be made about the organization of the data, it also has the side-effect of preventing access via the original format.

As previously mentioned, there are large ecosystems of existing scientific analytical tools that are built around one, or a small set of, scientific file format(s). These tools require access to the data in its original format because each format has its own approach to presenting metadata, its own syntax for accessing data and other file format specific idiosyncrasies. A system that restructures data prior to processing necessitates either abandoning these tools or maintaining two copies of the data (the original and the restructured), the latter approach increasing already onerous data storage requirements while also introducing the burden of keeping the two datasets synchronized.

A common use case for large-scale scientific data, such as simulation output, is "discovery science". In this scenario, a domain expert interacts with datasets (either visually or analytically), issuing a series of ad hoc queries in an effort to find "interesting" aspects within the data. The common use cases for databases is at odds with this approach, as databases are better suited to the repeated execution of a pre-selected set of queries. Furthermore, ad hoc queries often access data along several axes, which is less ideal in terms of generating indexes as a unique index would be required for each axes and that increases the storage overhead for indexes. Ad hoc queries often access data along axes that are not known ahead of time or are run as soon as the data is available, neither waiting for the data to be ingested into a database nor for indexing to be preformed. This class of queries does not observe the same benefit from indexes as queries that are well-aligned to a few axes, and therefore can benefit from a small set of indexes, or are known ahead of time, thereby allowing for indexes to have been specifically created. Jim Gray, a seminal researcher in the database community and primary contributor to the SDSS project, would ask scientific researchers "what 20 queries do you want to work well?" [130], thereby highlighting the requirement that databases be specialized to a given task in order to be efficient.

2.3 MPI

Some scientific communities utilize MPI [86] for simulations and the parallel processing of scientific data. MPI, as a framework, is not resilient to failures, makes no efforts to minimize data movement and requires the application developer to take a much more active role in choosing how inter-process communication occurs. These attributes are unattractive to us when considering the problem space we are addressing (large data on commodity clusters) and we therefore did not elect to build our research on top of MPI.

2.4 Shared Nothing Systems

Both academia and industry are actively working on addressing the need to scale processing frameworks to meet the growing mass of data. Many of these research projects represent "shared nothing" architectures that have been shown to have good scalability properties, both in terms of data size and degree of concurrency. These systems can be viewed as a response to the database community in that many of them do not restructure the data they process (i.e., enforce schemas) but rather interact with the data in whatever format it is presented to the system. Also, they present relatively simple designs that strive to enable as much parallelism as possible, typically via flexible scheduling.

The work presented in this thesis builds upon the "shared nothing" approach as we view the data formatting requirements necessitated by databases as unattractive and we also view MPI as both too low-level and too inflexible for our needs. Also, we conjectured, based on existing literature, that a significant portion of scientific research is done via ad hoc queries along multiple axes. Our theoretical work is based on *MapReduce* [30] due to its role as progenitor for many of the current shared nothing systems and the simplicity of its communication model. All contributions to date are built upon Hadoop, the predominant open-source implementation of *MapReduce* that several other prominent projects build upon [53, 57, 101].

Chapter 3

Background

We now present background information on the constituent aspects of this thesis. This section serves both as context and as a reference point for following discussions on the interactions of scientific data with *MapReduce* and Hadoop.

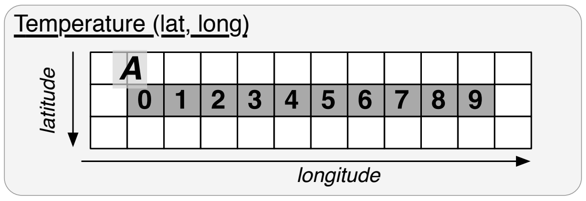
3.1 Scientific Data

As previously mentioned, scientific data is seeing a rampant growth in dataset sizes, with a significant portion of the data being stored in scientific file formats. In addition to the CMIP datasets already mentioned, there is the Sloan Digital Sky Survey (SDSS) [115] whose more recent dataset, DR9, was released in August 2012 and consisted of 60 TB of astronomy data; the Large Synoptic Survey Telescope (LSST) [79] schedule to come online in 2020 and is projected to collect 15 TB / night with a total of 100 PB over its lifetime [67]; and the Million Song Dataset [85], with its 300 GB dataset designed to support research into algorithms for music analysis. These datasets are stored in a variety of formats including NetCDF (CMIP), FITS files stored in an SQL database (SDSS), FITS files stored in a custom array-store (LSST) and HDF (Million Song Dataset). Based on the wide range of file formats used, we contend that research into scalable scientific computing should take this diversity into account by producing as general a solution as possible.

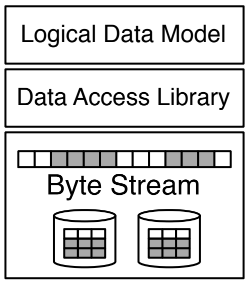
Scientific data is often highly structured, modeled as an array; a sphere; or some other shape [42, 89, 54]; while storage devices are built assuming a byte-stream data model. As a result, the high-level data models must be translated onto the simpler byte stream model. This translation (illustrated in Figure 3.1b) is performed by scientific file format libraries (e.g., NetCDF and HDF5). These libraries offer the dual benefits of (1) presenting a high-level data model and interface (API) that is semantically aligned with a particular problem domain (e.g., n-dimensional simulation data) while (2) hiding the nitty-gritty details of supporting cross-platform portable file formats. In Section 4.1 we show that while these benefits are great for application developers, they work against an efficient use of *MapReduce* for scientific data analysis.

3.1.1 Array Data Model

A data model specifies the structure of data and the set of operations available to access that data. One common representation for scientific data is a multi-dimensional, array-based data model [126] where data is accessed by specifying coordinates in the n-dimensional space, as opposed to offsets in a byte-stream. In this section, we present a simple version of such a data model that will be used throughout this thesis and later we develop a query language used to express common data analysis tasks.



(a) A 2-dimensional dataset



(b) Software stack used with scientific access libraries

Figure 3.1: An example of a multi-dimensional dataset and how the coordinates are translated into accesses of the underlying file's byte-stream interface.

The work presented in this thesis uses an array-based model that 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 3.1a 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 3.1a, the shaded sub-array has the corner point $(1, 1)$. Arrays also have an associated data type that defines the format of information represented by the array, but we assume a single integer value per cell in an array in order to simplify presentation [1]. Some file formats support variable length data types, such as a string. We do not consider those data types in this work as their use does not appear to be common (based on our survey of related work).

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

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

where S_A and c_A are the shape and corner point of A, respectively. Thus, the shaded sub-array in Figure 3.1a is described as the tuple (S_A, c_A) where $S_A = (1, 10)$, and $c_A = (1, 1)$. Note that we use the terms *array*, *sub-array*, and *slab* interchangeably.

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 3.1a depicts temperature readings associated with 2-dimensional coordinates specified by latitude and longitude values. Thus, the shaded sub-array may represent a specific geographic sub-region within the larger region represented by the entire dataset A.

3.2 *MapReduce*

MapReduce is a simple framework for enabling the parallel processing of large, unstructured datasets. The initial motivation for *MapReduce* was the observation that the aggregate bandwidth available within servers (or within server racks) was outstripping the bandwidth of data center networks (cross-rack links). In response, *MapReduce* focuses on moving computation to data (rather than the inverse) with the expectation that the first part of a given computation will likely reduce the dataset size. This reduction in dataset size results in a reduced burden on the network and enables the processing of larger amounts of data than had previously been possible.

One of the appealing aspects of *MapReduce* is that the application developer is not responsible for controlling inter-process communication or task scheduling. Rather, those tasks are handled by the framework. This frees the application developer from low-level concerns and allows them to instead focus on developing the code that is specific to their given work.

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

MapReduce's rise in popularity has led to its use in problem domains and with types of data beyond those for which it was originally designed (unstructured data), including scientific computing [36, 49, 168], and structured data [57, 53, 101]. For large-scale scientific datasets, *MapReduce* is an attractive parallel processing framework due to its simple programming model, ability to scale well on commodity hardware, and the fact that many problems in scientific computing translate well to its type of parallelization (we expand on these natural alignments between scientific computing and *MapReduce* in the following subsections).

3.2.1 *MapReduce* Data Model and Data Flow

As mentioned in Section 1, the implementation of our work is an extension of Hadoop. In this subsection, we dive into some Hadoop-specific implementation details as well as some design decisions originating in the *MapReduce* paper [30] that inform our later discussions.

Several components within Hadoop assume a byte-stream data model (i.e., the same format which most file systems support today) and a POSIX-like set of file operators when reading data from, and writing data to, storage. The user-defined *Map* and *Reduce* functions interact with data as typed key/value pairs, while the same data are treated as collections of bytes by the Hadoop framework during network transfers and local IO. While it is entirely conceivable that a *MapReduce* implementation could interact with data strictly as records, we focus on existing implementations, which interact with data both as byte-streams and records, in this thesis.

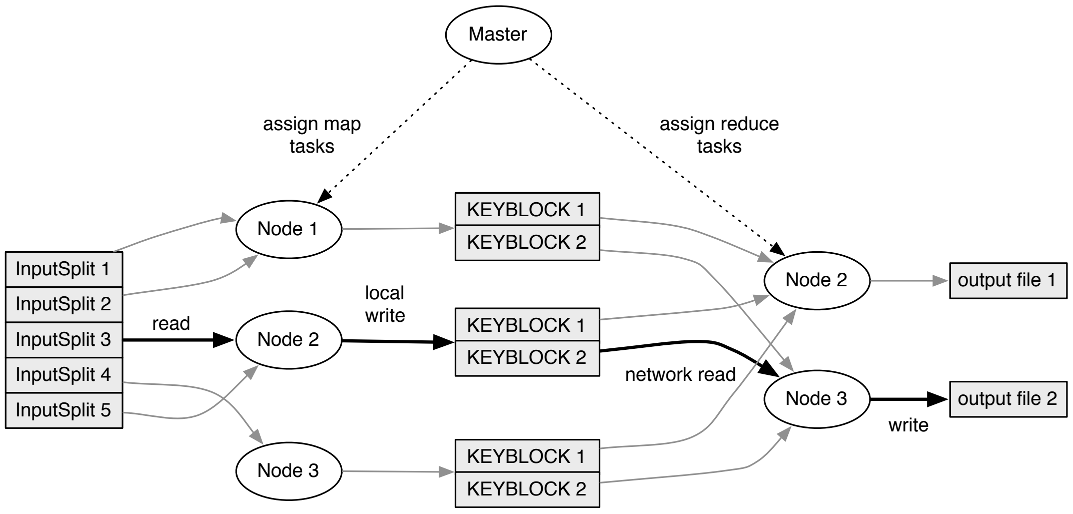


Figure 3.2: *MapReduce* data flow as depicted in the original *MapReduce* paper [30] with notations changed to our terminology

Sets	
$\mathcal{T}$	Input to a *MapReduce* job
$\mathcal{I}$	Set of all input splits
$\mathcal{O}$	Output from a *MapReduce* job
$\mathcal{B}$	Set of blocks in a distributed file system
$\mathcal{H}_g$	Set of hosts with local copies of block b_g
$\mathcal{H}_i$	Set of hosts with most of I_i stored locally
$v'_{k'}$	the set of all values in v' that are part
	of a key/value pair where the key is k'
$\mathbb{K}^{\mathcal{T}}$	set of keys that actually exist in
	K for a *MapReduce* job
$\mathbb{K}_i^{\mathcal{T}}$	set of keys that actually exist in
	I_i for a *MapReduce* job
$\mathbb{K'}_{\ell}^{\mathcal{T}}$	set of keys in K' that are assigned
	to KEYBLOCK$_\ell$ for a given job
Elements	
I_i	the i^{th} input split
b_g	the g^{th} block in a set
k, k'	a key in spaces K or K', respectively
KEYBLOCK	a partition of K'
KEYBLOCK$_\ell$	the ℓ^{th} KEYBLOCK
$\langle k, v\rangle$	a *key/value* pair in $K \times V$
$\langle k', v'\rangle^i$	a *key/value* pair in $K' \times V'$ created by
	the *Map* task processing I_i
RR(I_i)	the application of a RecordReader to an I_i
r	the number of *Reduce* tasks
m_i	a particular *Map* task

Table 3.1: Summary of common symbols

Figure 3.2 shows an illustration of the data flow for a *MapReduce* job (also called a "query" or "program"). When a *MapReduce* job begins executing, a central coordinator partitions the specified input data, $\mathcal{T}$, into a set $\mathcal{I}$ consisting of units (subsets) called *InputSplit*s, denoted I_i. An I_i is typically defined as byte-ranges in one or more files (e.g., bytes 1024 -

2048 in a particular file). Each split is assigned to, and processed by, one *Map* task. The *Map* task then employs a file-format specific library, called a RecordReader, that reads the data indicated by that *Map* task's assigned I_i and translates that data into 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 logically distinct keyspace, K', where "logically distinct" indicates that specific values in this domain do not necessarily correspond to the same values in a different domain (e.g., the data at coordinate {2,4} in the input is not necessarily related to the data at coordinate {2,4} in the intermediate data). A partitioning function then maps the key for every intermediate key/value pair to a specific keyblock, where a "keyblock" is a partition of the keyspace K' for the given *MapReduce* job.

Each *Reduce* task is assigned a keyblock and processes all intermediate data where the key portion of the key/value pair is within said keyblock. Prior to the application of the *Reduce* function, *Reduce* tasks perform a merge sort of all their data, combining all key/value pairs with equal k' keys into a key/value pair consisting of a single instance of the key and a list containing the values of all those pairs (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 (this property is one of the few guarantees that the *MapReduce* framework makes). It also has the secondary effect that all keys in a given keyblock are processed in their sorted order.

As a *Reduce* task applies the *Reduce* function, it emits a new set of values, which are written to the underlying storage system as the final result of the *MapReduce* job. In Hadoop, the default storage systems is a distributed file system referred to as the Hadoop Distributed File System (HDFS). The total set of data written as output from all *Reduce* tasks for a given *MapReduce* job is denoted $\mathcal{O}$.

The design of *MapReduce* provides a few simple guarantees:

1. All data in the input will eventually be assigned to some *Map* task
2. All values for the same k' will be processed by a single *Reduce* task and at the same time
3. Output is produced atomically (i.e., only output from one instance of a given task is ever exposed to the system or included in the final output)

The *MapReduce* framework is free to partition data and schedule tasks as it sees fit, typically taking data locality into consideration, as long as it fulfills these guarantees. The lack of ordering guarantees among tasks of the same type (*Map* or *Reduce*) as well as the inability to associate tasks with physical servers enables a higher degree of flexibility and scalability (both up and down) than is possible with more rigid parallel frameworks (e.g., MPI). This flexibility also enables a very simple fault-tolerance model based on re-executing failed tasks.

3.3 *MapReduce*'s Communication Model

While a significant body of work relating to formal definitions of the *MapReduce* job model and its relation to other computing models exists [30, 40, 69, 74], little attention has been given to *MapReduce*'s internal communication model. In general, this is reasonable as there are points in the data flow of a *MapReduce* job where it is not possible to reason about the relationship between a function's input and its output, necessitating the use of worst-case assumptions. These assumptions result in the tasks that make up a *MapReduce* job having very little communication with each other. *Map* tasks are assigned their InputSplit by a central coordinator and write their output to local storage. *Reduce* tasks are informed of *Map* task completion by the same central coordinator and read their assigned data as resources allow. Only after all *Map* tasks have completed, as indicated by the set of *Map* task completion messages, are *Reduce* tasks free to begin processing their assigned data.

3.3.1 Blackboxes in the *MapReduce* Data Flow

There are two points in the flow of data through a *MapReduce* program where it is difficult, or impossible, to correlate a task's input with its output (or vice versa). These points present a roadblock to any meaningful optimization of communications within a *MapReduce* job. In this subsection, we discuss those two points in detail.

In running a *MapReduce* job, the user specifies a set of input data and a RecordReader. The *MapReduce* framework splits up the specified data into a set of InputSplits $(I_1, I_2, ..., I_i)$, each of which are read by an instance of the specified RecordReader that then outputs key/value pairs for use as input to a *Map* function. In practice, the input to a RecordReader is usually expressed as byte-ranges while the output is a set of key/value pairs where the key is in some logical keyspace. This mismatch between the format of a RecordReader's input (byte-ranges) and output (keys) prevents reasoning about any relationships between the two without taking drastic steps, such as invoking the RecordReader, providing it with a byte-range to process and observing its output. This is the first point at which the *MapReduce* framework masks the flow of data. A definition for the data flow through a RecordReader is shown in Formulation (3.1).

$$\begin{aligned}&\text{for } m_i = |I_i|\ ,\\ &\mathrm{RR}(I_i)\ = \left\{\langle k_1, v_1\rangle^i, \langle k_2, v_2\rangle^i, ..., \langle k_{m_i}, v_{m_i}\rangle^i\right\}\\ &\langle k_j, v_j\rangle^i \in K \times V \quad \forall j \in \{1, 2, ..., m_i\}\end{aligned} \tag{3.1}$$

The second point at which the *MapReduce* data flow is opaque is the assignment of intermediate key/value pairs to KEYBLOCKs. This process represents a partitioning of the keyspace for the intermediate data (K') where each subset (KEYBLOCK) will be assigned to a *Reduce* task. In Hadoop, the assignment of a given key/value pair of intermediate data to a KEYBLOCK is done via modulo arithmetic over a binary representation of the key. In much the same way as the RECORDREADER, it is difficult, if not impossible, to reason about the relationship between this function's input and output. In this case, the difficulty is caused by the interaction of the hash function with the key and the implementation specifics that come into play (e.g., a given class's *hashCode()* [52] implementation). A definition of the hash function is shown in Formulation (3.2).

$$hash : K' \rightarrow \{\text{KEYBLOCK}_\ell \mid \ell \in \{1, 2, ..., r\}\} \tag{3.2}$$

The goal of this partitioning is to create a roughly even distribution of intermediate keys across KEYBLOCKs (issues of skew in the distribution of data relative to keys is deferred to Section 9). A side-effect of using a modulo function is that it precludes reversing the process with any accuracy; it is not possible to take a key/value pair in a KEYBLOCK and derive which *Map* task generated that data.

3.4 The Array Query Language

Hadoop queries are typically expressed by specifying the input file(s) and the class (function) that should be applied to each key/value pair in the input. Queries conducted on scientific data are usually applied to groups of inputs (multiple values) and may require skipping records in a predictable pattern or excluding certain ranges of data. The need for a more expressive way to specify queries in Hadoop prompted us to create a simple language for our research.

At a high-level, our array-based query language, which borrows heavily from AQL [76], 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). Note that the language is not intended to be a contribution of this thesis, but rather serves to expose the semantics of queries necessary to perform certain types of optimizations.

3.4.1 Simple Example

One common task when working with scientific data, especially when the content is initially unknown, is to evaluate *ad hoc* aggregation queries. For example, consider the following query:

Example 1: *"What is the maximum observed temperature in the dataset represented by the shaded sub-array in Figure 3.1a?"*

Figure 3.3 depicts a possible execution of this query via 3 *Map* tasks that each process their own portion of the shaded region and a single *Reduce* task that receives 3 inputs and produces the resulting 1×1 array, containing the value 9, as the final output.

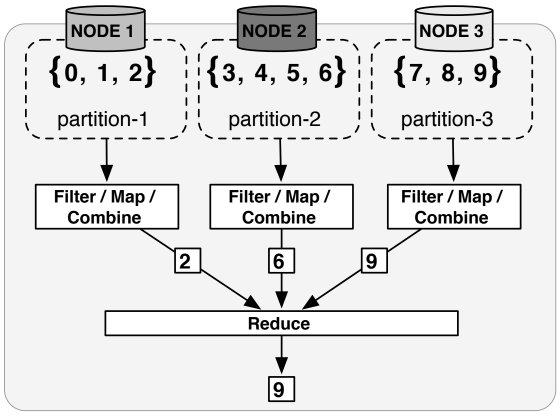


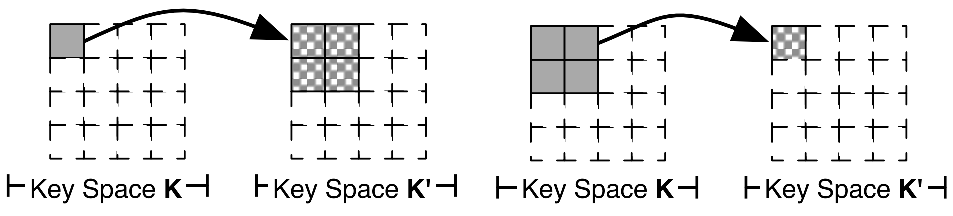
Figure 3.3: Execution of the *max* aggregate function in *MapReduce*.

3.4.2 Constraining Space and Extraction Shape

A constraining space is defined as the data that a given query should be applied to and is denoted $\mathcal{T}$. It is necessary to use a constraining space when specifying that only a portion of a dataset should be processed by a particular query.

Our query language incorporates the idea of an extraction shape, which is a concrete representation of how a given *Map* function translates keys in its input, K, into keys in its output, K'. Specifically, the extraction shape is logically tiled, in a given order, over $\mathcal{T}$ with each instance representing a unique k' key in K'. The inclusion of an extraction shape in our query language addresses the need expressed earlier in this section for queries to be applied to groups of values. Furthermore, the existence of an extraction shape enables *MapReduce* to translate any $k \in K$ to a $k' \in K'$ by using the extraction shape to project $k \rightarrow k'$. Defining an extraction shape for a *MapReduce* program that is applying a query over structured scientific data is straightforward and can be represented efficiently via an n-dimensional array in the same way as the shape component of an array is specified [18].

As an example, consider a query over a 2-dimensional dataset that is down-sampling 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 would be $\{2, 2\}$ (indicating every 2x2 input shape translates into a single element in the output). An example of this translation can be seen in Figure 3.4(b) as well as an extraction shape that represents an up-sampling in Figure 3.4(a). Strided access (reading data at regularly spaced intervals) can be described by adding an additional n-dimensional array indicating the stride lengths (and other patterns, such as nested strides, can be represented as well).



(a) 1 value in K mapping to 4 values in K' (b) 4 values in K mapping to 1 value in K'

Figure 3.4: Examples of an extraction shape mapping values in K to values in K'.

From a *Map* task perspective, Algorithm 1 gives a simplified representation of how the extraction shape is used during the processing of an InputSplit. The input to the *Map* function is the extraction shape and the data indicted by the InputSplit. First, the *Map* function extracts relevant array data based on the extraction shape using *ExtractInput()*. A *Group ID* is assigned to the data contained in an unique instance of the extraction shape. Then, the *Map* task is applied to the repartitioned data (*ArrayData*) and the resulting value(s) written out as intermediate data with the *Group ID* serving as the key.

For some queries, the application of the extraction shape will not produce such regular output. For value-based queries, such as requesting all of the data where the value exceeds some threshold, the *Map* task output may be a single-cell, indicating one position where a threshold exceeding value was found, or all the cells in an array that meet the threshold (in the latter case, a list of values may be returned if more than one value in the array exceeds the threshold).

```
Algorithm 1: Map()
    Input: Extraction Shape, Input Split
    (GroupID, ArrayData) ← ExtractInput(Extraction Shape, Data)
    value ← Map(ArrayData)
    Emit(GroupID, value)
```

3.4.3 Formal Model

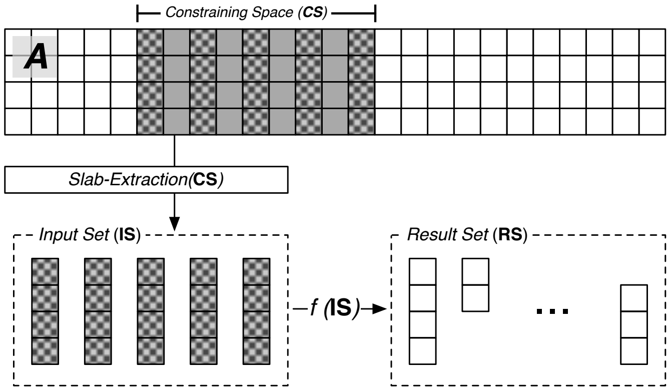


Figure 3.5: Illustration of the query language semantics.

Formally, the language is defined as a 3-tuple $(\mathcal{T}, SE, F)$. First, $\mathcal{T}$ denotes a contiguous sub-array called the *constraining space* that limits the scope of the query. Second, the *slab extraction* function SE, which uses the extraction shape to generate a set of sub-arrays, IS, that are referred to as the *input set* and where each sub-array $s \in IS$ is also contained in $\mathcal{T}$. 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:

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

3.5 Types of Queries

For the purposes of this research, we categorize queries as being of one of three types. Those that:

1. operate over fixed units of cells, based on the coordinate of the cells

    - Example: Average of a latitude by longitude area. Each area will result in exactly one value being output.

2. operate over the values of the cells, such that the coordinate can be used to organize the values

    - Example: filtering all values in each time-step of an experiment that exceed a specified threshold. Zero, one or more results can be returned for each time-step.

3. operate over the values of cells, using the values for organizing the results

    - Example: sort all of the cells in a time-step by their values with the coordinates effectively becoming the values.

The work included in this thesis, to be presented in following sections, can be applied to queries that fall into types 1 and 2. For these two classes of queries, the *MapReduce* framework can comprehend the number and value of the keys that will be produced as the data is passed through the *MapReduce* framework. This is possible because, in both cases, the keys are derived from the coordinates of the input and those are readily extracted from the metadata for the dataset. Queries of type 3, on the other hand, use the values of each input as their key. As a result, the dataset can have any key representable by the given data type (int, float, long, etc.) and a given value may appear zero, one or more than one time. This unpredictability renders our approaches ineffective and would require an indexing of the values of a dataset in order to achieve the same results described in this thesis. We do not endorse indexing in the general case, due to the issues outlined in Section 2.2.

3.6 Applicability of This Work

At a high level, the work presented in this thesis can be considered an attempt to optimize *MapReduce* programs when there exists some sympathetic alignment between a query and the dataset it is being applied to. This optimization is, in general, the result of using more informed logic within *MapReduce*, rather than defaulting to using worst case assumptions. Queries that are antagonistic to the alignment of the data being operated on cannot be optimized in any meaningful way that we have found without violating the ethos of *MapReduce*. In these cases, our work must devolve to using the original approaches outlined in [30].

In the same way that human intervention is typically required for some data management tasks (specification of indexes, schema design, query optimization), data in our system may need to be reformatted if the inefficiencies caused by the misalignment of queries to data exceeds the cost of data reorganization. Solutions other than restructuring a data set have been suggested, including storing replicas with different layouts [147] and aligning data to fit the query while presenting the original layout via indexes [116]. In the same spirit, the database community has investigated divergent replicas [29]. We contend that adapting the layout of data that is not well aligned to the queries being run on that data is best suited to a higher-level process (some sort of storage manager or query optimizer) and is beyond the scope of this thesis.

Chapter 4

Processing Data in Scientific File Formats

It is difficult to leverage the structure of data stored in scientific file formats because that structure is hidden behind the simple APIs exposed by access libraries. This section describes our research into overcoming this barrier and how we enabled processing of scientific data with Hadoop (a discussion on why we chose to base our implementation on Hadoop can be found in Section 1).

4.1 Data Model Incompatibilities

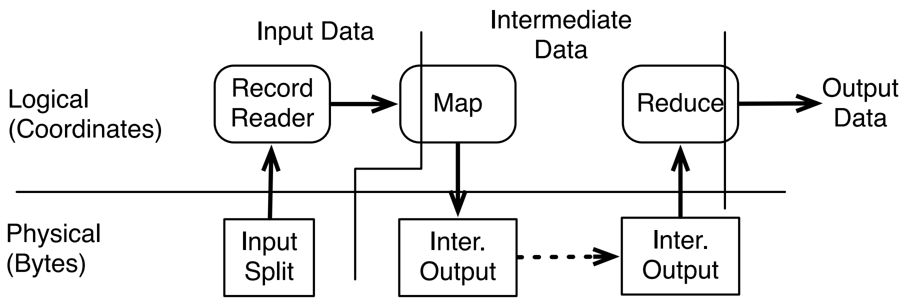


Figure 4.1: Variations in the abstraction used throughout the execution of a Hadoop query. The dashed line indicates network transfer.

The first issue that arises in extending Hadoop for scientific data is the question of how to specify InputSplits. Scientific access libraries require that data accesses be specified as sets of coordinates (Section 3.1.1) while many RecordReaders in Hadoop define their

InputSplits as ranges of bytes (Section 3.2.1). Rather than translating from a set of coordinates into the corresponding byte-ranges, which would require complicated file format specific code, we chose to elevate the abstraction at which InputSplits are defined from byte-ranges to coordinates. A depiction of the abstraction-level transitions that occur during a Hadoop query is shown in Figure 4.1 while Figure 4.2 shows the same query when InputSplits are specified as coordinates. While necessary to enable the use of scientific access libraries in Hadoop, using logical coordinates as the unit of definition for InputSplits yields additional benefits that are further explored in Section 6.2.

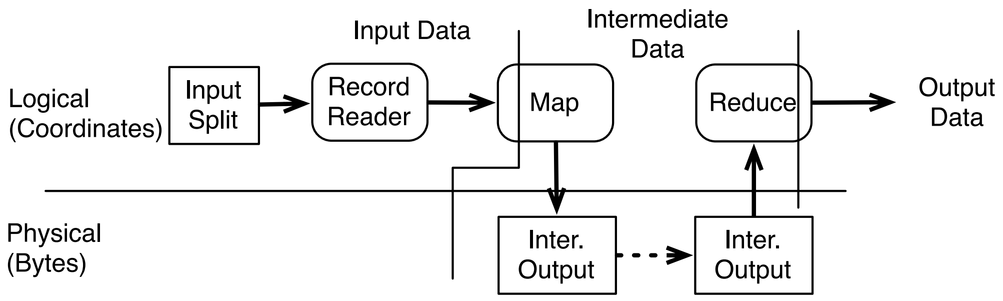


Figure 4.2: Variations in the abstractions used to interact with data throughout the execution of a Hadoop query after elevating InputSplits to the coordinate abstraction. The RecordReader calls the access library which internally converts coordinate accesses into reads of the underlying byte-stream. The dashed line indicates network transfer.

4.2 Data Locality

One of the impetuses for the development of *MapReduce* was the observation that network bandwidth was a limiting factor for data-intensive computations on large clusters. In recognition of this, *MapReduce* strives to place computations on nodes that contain a majority of the specified data locally, thereby reducing network resource requirements. This reduction in network resource usage enables *MapReduce* to process otherwise infeasible amounts of data and is one of the main benefits of *MapReduce*.

Generally, *MapReduce* is deployed on top of a distributed file system with *Map* and *Reduce* tasks executing on the same nodes that also host the file system. Files are usually stored as fixed-size blocks (byte-ranges) that are replicated and distributed among the nodes. Formally, a file is composed of a set of d blocks, $\mathcal{B} = \{b_1, b_2, \ldots, b_d\}$, where each block b_g is associated with a set of hosts $\mathcal{H}_g$ that each store a copy of b_g locally. The data contained in a block b_g are accessible indirectly through the file system interface, either remotely via a network connection or locally on a host $h \in \mathcal{H}_g$. Additionally, the *MapReduce* framework assumes that the underlying file system is capable of exposing the set of hosts $\mathcal{H}_g$ for any block b_g.

4.2.1 Partitioning and Placement in Hadoop

In Hadoop, InputSplits are usually represented as byte-ranges that are typically aligned to blocks in the underlying storage system (e.g., if the file system has a block size of 64 MB, then reasonable InputSplits could be 0-64 MB, 64 - 128 MB, etc.). We refer to this process as "partitioning". Achieving high rates of data locality is relatively easy in this situation: query the file system for the list of hosts ($\mathcal{H}_g$) that possess the block that a given InputSplit was aligned to and then instruct the scheduler to prefer those hosts. We refer to this second process as "placement". The combination of these two processes is shown in Figure 4.3.

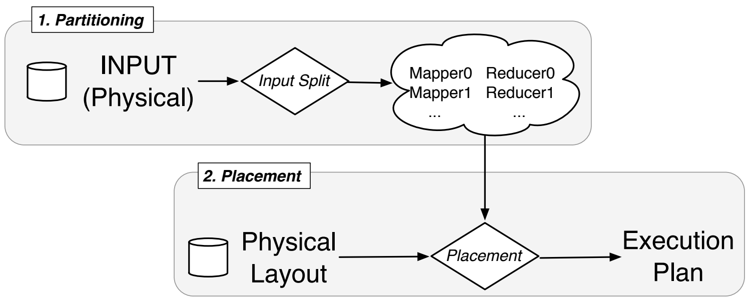


Figure 4.3: Partitioning and placement in Hadoop.

4.2.2 Partitioning and Placement for Scientific Data

The decision to define InputSplits as sets of coordinates, rather than byte-extents, coupled with scientific file format abstracting away data placement in underlying byte-streams renders the previous approach for achieving data locality ineffective. Therefore, we are faced with the responsibility of finding a new means for achieving data locality. In other words, we must find a process for creating sets of coordinates that, when passed to a scientific access library, results in localized accesses of the byte-stream in the underlying file(s). This requirement means that some amount of information describing how scientific data is laid out in the underlying distributed file system is now needed during partitioning. This new process for the partitioning and placement of InputSplits is shown in Figure 4.4. The line labeled **L** is a contribution of our work that, during partitioning, utilizes knowledge of the physical layout of the scientific data that we are generating logical level (coordinates) InputSplits for.

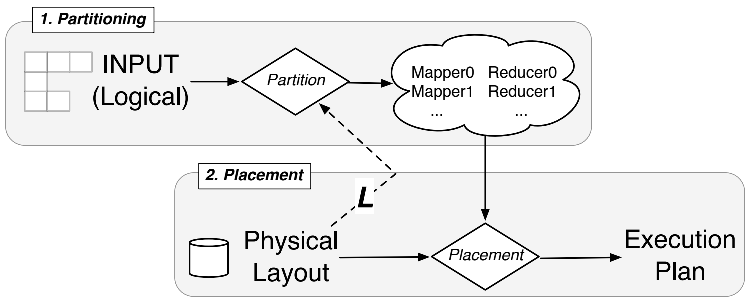


Figure 4.4: Partitioning and placement when InputSplits are defined at the logical level.

In considering how to achieve data locality while specifying InputSplits as coordinates, an "obvious" solution was not readily apparent. Consequently, we designed, implemented and evaluated three possible approaches, each requiring a different depth of knowledge about the internal structure of the file format containing the data being processed by Hadoop.

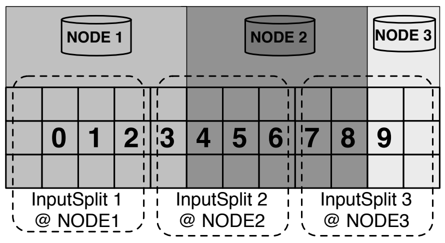
4.2.2.1 Baseline Partitioning and Placement of Scientific Data

First, we present a simple solution for the partitioning and placement of scientific data that will serve as a baseline against which further optimizations can be compared. We refer to this solution as the *Baseline* 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 produce inefficient IO that results in long execution times.

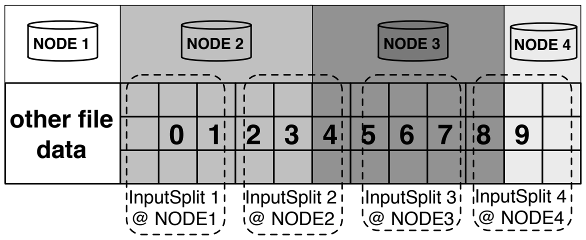
As has been mentioned, scientific access libraries use a black-box design in which file layout information is obscured, thereby frustrating automatic optimizations that rely on such knowledge. Thus, most users of the libraries resort to the manual construction of InputSplits, making the quality of such partitions dependent on how much file layout information is known by the user. Therefore, 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 splits the logical input into a set of InputSplits, one for each physical block of the input file. Consider Figure 4.5a which shows a 3×12 array stored within a file occupying three physical blocks, located on nodes $NODE1$, $NODE2$, and $NODE3$. Using knowledge that the file format stores data in column-major order, a reasonable partitioning strategy is to form three equally sized InputSplits with each containing four columns. These InputSplits are shown in the figure using dashed frames, and are labeled as *InputSplit 1,2,3*.

The baseline placement heuristic is a round-robin assignment of InputSplits to physical locations. The result of a round-robin placement is shown in Figure 4.5a using the notation *InputSplit x@NODEy* (e.g., , *InputSplit 2* is processed on node *NODE2*). The example shown in Figure 4.5a illustrates the difference in logical and physical InputSplits. In the figure, *InputSplit 1* references 4 columns all contained in the block on $NODE1$, while *InputSplit 3* references 2 columns from the block on $NODE2$, and 2 columns from the block on $NODE3$. In general, blocks are distributed over all nodes in a cluster, resulting in the *Map* task that processes *InputSplit 3* reading at least half the specified data remotely. For infrastructures with network bottlenecks, and when IO is dominated by *Map* task reads, this misalignment of logical partitioning with physical layout can have a significant impact on performance. However, simple data sets with well-defined physical layouts (e.g., column-major ordering) that are stored with typical file block sizes are usually only slightly misaligned. As a result, the *Baseline* partitioning strategy can perform well. Unfortunately, achieving a good partitioning in the general case can be difficult as, in practice, few files contain a single, well-aligned variable.



(a) Baseline partitioning strategy applied to a file containing a single array.



(b) Same scenario when other data shares the same file.

Figure 4.5: Effects of file composition on the *Baseline* strategy

Consider now the task of partitioning the same logical array shown in Figure 4.5a, but assume that the array is stored in a file containing additional data. This additional data could be header information such as data attributes or even other data sets stored within the same file. A depiction of this new file is shown in Figure 4.5b and there are two things to note. First, since there are now four blocks storing the larger file, but the logical space of the array (constraining space) is unchanged, the size of the InputSplits become smaller with the *Baseline*

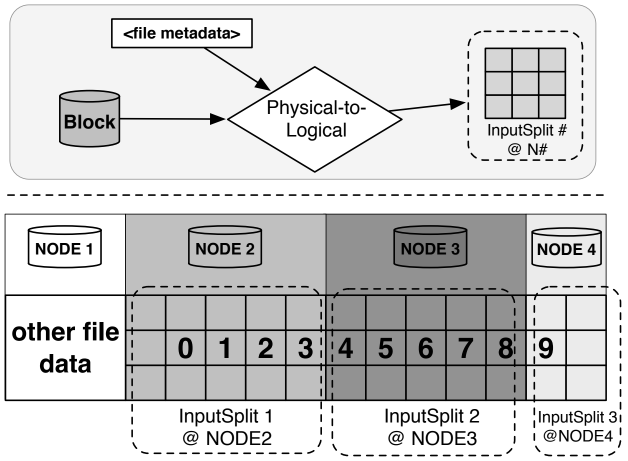


Figure 4.6: Physical-to-Logical Partitioning

strategy. The second, and more important, difference is that a typical round-robin placement of InputSplits to blocks can result in very poor data placement because of the shifting factor introduced by the additional data in the file. For instance, the data referenced by *InputSplit 1* will be read entirely over the network when it is processed at $NODE1$.

To illustrate the trade-off between data set complexity and user sophistication we perform two experiments, both of which evaluate a query over an identical logical array using the *Baseline* partitioning strategy. The array used in the first experiment is stored in a file by itself and in the second experiment the array is stored in a larger file along-side other arrays. We measured the percentage of reads that are satisfied by disks local to *Map* tasks, thereby evaluating the quality of the *Baseline* partitioning and placement strategy. When a single array is stored in a file, the *Baseline* partitioning and placement achieves 71% read locality, but when multiple arrays are stored in a single file the read locality drops to 5%. This degradation in locality is due to the additional variable(s) that are present in the file, but not in the query, causing misalignment during the placement phase.

4.2.2.2 Physical-to-Logical Translation

The second approach we considered is to directly translate the byte-range represented by a physical block into its equivalent logical representation by replicating the internal logical of a particular scientific file format. More specifically, the file format specific knowledge is combined with a new API call we added that takes the coordinate for a specific cell in the logical space and returns an offset in the underlying file to search out the first and last logical cell stored on a block in the underlying distributed file system. The search for the first, or last, offset is similar to a binary search of the logical coordinate space with the goal of finding the coordinate that translates into a specific offset in the file. Once the first and last logical coordinates for a block are found, they are used to form an InputSplit. At the top of Figure 4.6, 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, InputSplits are precisely aligned with physical block boundaries. Therefore placement is trivial; an InputSplit is matched with the block from which it was generated.

Despite the precision of this technique, it can be difficult to implement for complex file formats and would require a different implementation for each file format as well as every different on-disk representation for a given format. As an alternative to this complexity, we next introduce a more general purpose technique for constructing InputSplits.

4.2.2.3 *Chunking & Grouping*

The third technique that we investigated for partitioning and placing InputSplits of scientific data is referred to as *Chunking & Grouping*. This technique decomposes the input into many fixed-size units called chunks and, for each chunk, a random sampling of byte-stream locations is taken using the same new call that we mentioned in Subsection 4.2.2.2. The sampling technique allows Hadoop to then group chunks into flexibly defined InputSplits with increased locality of reference.

Figure 4.7 illustrates how *Chunking & Grouping* is used to create InputSplits. At the top of the figure, fixed-size chunks are grouped together, based on the sampling process, into InputSplits such that each InputSplit references primarily data within the same physical block in the distributed file system. Next we describe *Chunking & Grouping* in more detail.

Chunking. The first step is the decomposition of input at the logical level into a set of chunks (i.e., fixed-size, contiguous, non-overlapping sub-arrays). The set of chunks that cover the entire input space is given by $\mathcal{C}$, where each chunk $c \in \mathcal{C}$ 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 InputSplits can be created, but results in more overhead due to the necessity of managing many small chunks. A larger block size creates less of a metadata burden while potentially resulting in reduced rates of local data accesses. A detailed evaluation of chunking strategies is beyond the scope of this thesis, but we provide the parameters for our experiments in Section 4.3.

Grouping. Grouping is the process by which chunks in the set $\mathcal{C}$ are combined to form InputSplits. The goal of grouping is to form InputSplits that reference data located in the fewest number of physical blocks. Thus, the problem of creating InputSplits is equivalent to grouping chunks $c \in \mathcal{C}$ by block, such that each group maximizes the amount of data referenced in the block that the group is associated with. The resulting InputSplits are what we refer to as the logical-to-physical mapping, defined by the set $LTP = \{(b_0, \mathcal{I}_0), (b_1, \mathcal{I}_1), \ldots, (b_m, \mathcal{I}_m)\}$, which associates a physical block b_g with an InputSplit $\mathcal{I}_i$ that is 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 s cells is selected from the logical space represented by a chunk c 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 an 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_g with the highest frequency is chosen and the chunk being considered is added to the InputSplit $\mathcal{I}_i$ associated with that block.

Sampling is the dominant cost of *Chunking & Grouping*. In our evaluation section we use a sampling ratio of 0.01% on a file containing 35 billion logical coordinates resulting in the sampling operation being performed approximately 3.5 million times. Microbenchmarks show that our implementation of sampling for NetCDF-3 files can achieve 600,000 samples per second. The sampling process is parallelizable and, except for a small amount of metadata, does not need to access data on disk. Furthermore, the sampling results can be cached and reused for subsequent queries because sampling is consistent for a given file (since its layout on the distributed filesystem will not change),

Example. First we consider the set of chunks E, consisting of the 6, 3×2 sub-arrays, shown at the top of Figure 4.7 (indicated by dashed boxes). We refer to these chunks by their position in the figure (i.e. $1 \ldots 6$). A random sampling is performed for each chunk $e \in E$. 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 the block on $NODE1$ or the block on $NODE2$. Consequently, remote data access will be required regardless of which node is assigned chunk *3*. In instances where a chunk is not evenly split across two blocks, uniform random sampling is an effective way to select the block with a majority of the data and thereby minimize the amount of remote reads for that chunk. The result of chunking and sampling is shown at the bottom of Figure 4.7. The final LTP mapping

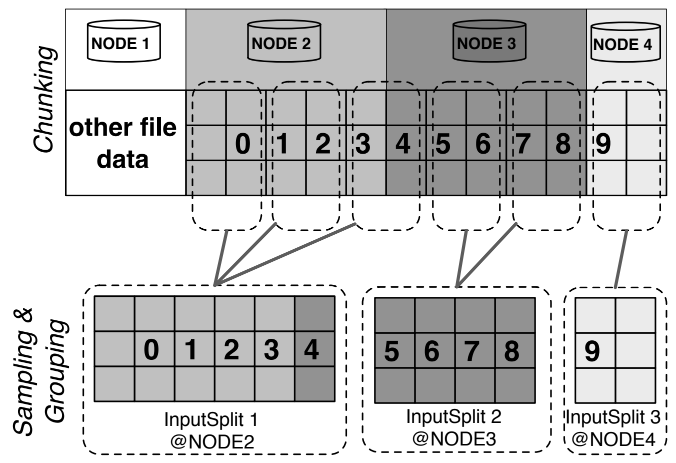


Figure 4.7: *Chunking & Grouping* example.

is given as:

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

Once computed, the LTP mapping can 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. Since each InputSplit is associated with a block in the underlying distributed file system, placement is trivial (the InputSplit uses the same host list associated with the given block).

4.3 Data Locality Experiments (Query 1 & 2)

In order to measure the efficacy of the three proposed solutions, we executed two different queries over a synthetic data set modeled after environmental simulations. The metadata for the dataset is shown in Figure 4.8 and can be thought of as fifteen years of daily windspeed measurements over 0.5 ° longitude by 1 ° latitude regions and at 50 different elevations. A description of the cluster used for this experiment is presented in Appendix A.

Two exemplary queries are shown in Figures 4.9a and 4.9b. Figure 4.9a finds the median temperature over two days for a range of 10 elevation steps over a 18 ° latitude by 36 ° longitude geographical area and Figure 4.9b uses an average function to down-sample from a daily, 0.5 ° latitude resolution dataset to a dataset with weekly measurements at a 1 ° latitude

```
dimensions:                variables:
    time = 5475;               int pressure(time, lat, lon, elev);
    lat = 360;
    lon = 360;
    elev = 50;
```

Figure 4.8: Metadata for the NetCDF dataset used in the data locality experiments.

resolution. In both queries, the constraining shape is used to eliminate the first and last 10% of the input data from the query.

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

(a) Query 1

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

(b) Query 2

Figure 4.9: Query declarations for data locality experiments

Test Name	Query 1	Query 2
	Local Read(%)	Local Read(%)
Baseline	9.3	3
Chunking & Grouping	80	84
Physical-to-Logical	88	93

Table 4.1: Locality results for query 1 and query 2.

Table 4.1 shows the locality achieved by the three methods. We define data locality as the ratio of data read by *Map* tasks where that data was stored on the same node that the *Map* task was executed on relative to the total amount of data read by all *Map* tasks. The *Baseline* approach does not do well, achieving locality rates of 9.3% and 3%, respectively. *Chunking & Grouping* achieves 80 and 84% locality on the two queries while *Physical-to-logical* achieves 88 and 93% locality. Our experiments show that *Chunking & Grouping* provides approximately 90% of the data locality realized by *Physical-to-Logical* while using a more general approach to partitioning and placement.

All experiments were conducted using the default Hadoop scheduler, which will opt to place an InputSplit on a computer that does not possess the data locally rather than let that server sit idle. This has the side-effect of rendering 100% data locality highly improbable bordering on functionally impossible. We did conduct experiments with a weighted-scheduler and configured it to be heavily biased in favor of data locality. Using this approach, both *Physical-to-Logical* and *Chunking & Grouping* saw data locality improvements, with *Physical-to-Logical* achieving greater than 99% locality. However, the total query run-times were negatively impacted in all tests with this scheduler. In light of this, we did not continue to use that scheduler nor do we include those numbers in this thesis.

4.4 Section Summary

This section describes why scientific data cannot be efficiently processed using the baseline approach and how perturbations in input partitioning at the logical level can lead to poor performance. Thus, any efficient solution needs to overcome the restrictions imposed by scientific access libraries opaque IO processes.

In light of the results reported in this section, we choose to use the *Chunking & Grouping* approach for subsequent experiments. The generality of the approach coupled with the relative efficacy presents a preferable option given that our goal is to create a system appropriate for use with a wide variety of file formats and problem domains. More specifically, the additional engineering and format specificity required by the *Physical-to-Logical* approach did not seem warranted given the incremental increase in data locality that it provides.

Chapter 5

Working Within the Current Communications Model

This section focuses on extending *MapReduce* for scientific data with the assumption that the current *MapReduce* communication model, described in Section 3.2.1, is maintained. The changes made to Hadoop to support the work in this section involved sub-classing well-known interfaces that are commonly altered and can therefore be thought of as non-invasive.

The content of an InputSplit not only affects data locality rates; other processes in the *MapReduce* framework that consume *Map* task output are also influenced by what data is specified as an input to a *Map* task. For example, the method of constructing an InputSplit can be adjusted to provide performance benefits. In several of the following subsections, we explore the interaction of InputSplit composition with processes beyond the *Map* task.

5.1 Combiners

As a *MapReduce* program executes, data is transferred between HDFS, local disks and other tasks over the network. A common optimization for *MapReduce* queries is to utilize a *Combiner* function that performs local data reduction on *Map* task output. This results in a (possibly significant) reduction in the amount of data transferred across the network when intermediate data is read by a *Reduce* task. Algorithm 2 defines how a *Combiner* interacts with *Map* task output when applying a *max* function. Note that (ideally) multiple {*Group ID*, ArrayData} tuples will be combined by the same *Combiner* execution, providing the aforementioned reduction in intermediate data size. In practice, *Combiner*s are often *Reduce* functions that are used to combine the output of multiple *Map* tasks while they are still local to the server that executed the *Map* task(s).

```
Algorithm 2: Combine() / Reduce()
    Input: GroupID, ArrayData
    Max ← Maximum(ArrayData)
    Emit(Group, Max)
```

The execution of a *max* function over three InputSplits is shown in Figure 5.1. Since partition-1 and partition-2 happened to both execute on the same node ($NODE1$), a *Combiner* can take their outputs (2 and 6 respectively), combine them, and write a single value (6) to local storage. In this example, the *Combiner* reduced both the amount of data written to local storage and data sent over the network.

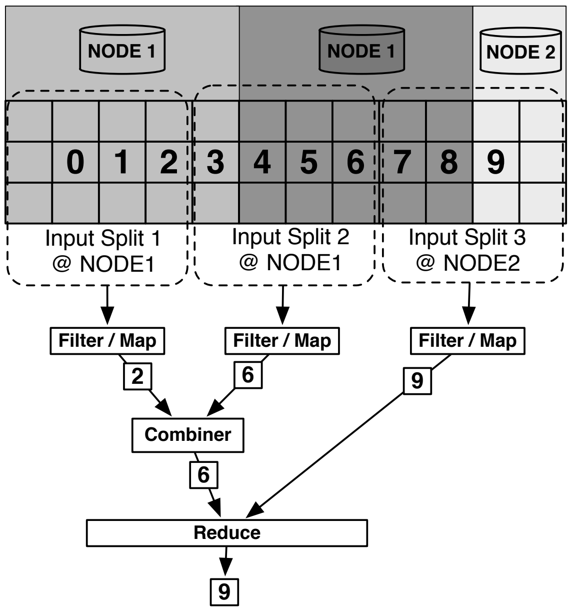


Figure 5.1: Example of a *Combiner* applying a *max* function to the output of two *Map* tasks on $NODE1$.

5.1.1 Holistic combiners

Employing a *Combiner* is straight-forward when the function being applied is distributive because the order in which sub-results are combined has no bearing on the validity of the result. The outputs of *Map* tasks are combined at each node and those values are combined during the *Reduce* phase to produce a final result. In contrast, holistic functions, such as *median*, require all data be evaluated at the same time in order to produce a correct result. For Hadoop, this precludes using a *Combiner* for holistic functions since all inputs for a given k' key must be sent to the *Reduce* task in order to guarantee correctness as the *Reduce* task is the only point where *MapReduce* guarantees that all values for a given key will be present.

Our survey of other attempts at constructing general frameworks for parallel scientific computing indicates that holistic functions are more common in scientific computing than they are in general computing. Given this observation, we placed an emphasis on researching what could be done to support the use of a *Combiner* for holistic functions. The resulting work yielded an opportunistic approach that improves performance over the existing system (in which a *Combiner* cannot be used).

In our holistic *Combiner*, we use the extraction shape to determine how many values exist for a given intermediate key (k'). The holistic *Combiner* processes each key and if the correct number of values are present, the holistic function is applied. If, instead, an insufficient number of values are present, the *Combiner* is not applied and all of the values are stored as intermediate data. These values will be routed to the same *Reduce* task as the other values for the given key and the *Reduce* task will apply the holistic function. A holistic *Combiner* is made possible by virtue of our use of an explicitly specified extraction shape in the query language. Without knowledge of the extraction shape, a *Combiner* would not know how many values were expected for a given key and therefore could not safely apply a holistic function.

In Subsection 5.3, we discuss an approach for constructing InputSplits that trades a decrease in data locality for an increase in the rate at which the holistic combiner can be applied. A performance evaluation is included in that discussion.

5.2 NoScan

Typical *MapReduce* queries that process unstructured data, such as log-processing, must read data stored on hard drives and then filter out irrelevant portions in memory. The structured nature of scientific file formats enables us to avoid these block scans when constructing requests at the logical level by including only the data necessary to complete the current query. This technique, referred to as *NoScan*, prunes input partitions to eliminate unnecessary segments of the logical space, thereby reducing the total amount of data read during query execution to the minimum required by the query.

Using knowledge of the *constraining space* component of query **Example 1** (Section 3.4.1), the InputSplits that were constructed previously in Figure 4.7 can be trimmed such that it contains exactly the components of the logical model that are required to complete the query.

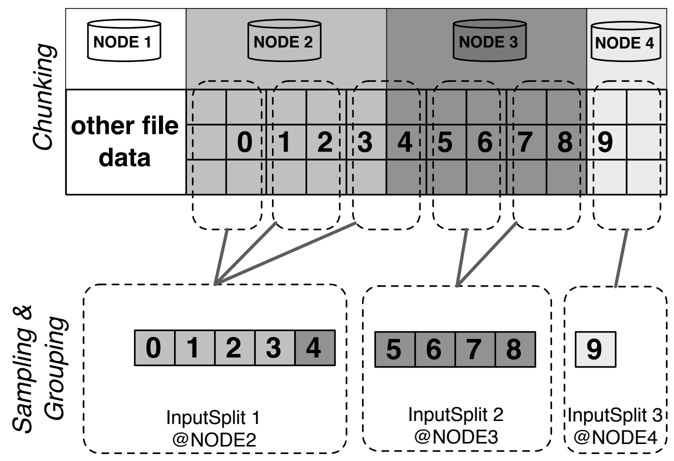


Figure 5.2: *Chunking & Grouping* with *NoScan*

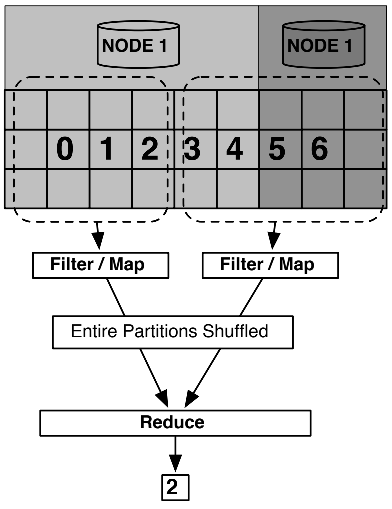
Figure 5.2 illustrates this trimming process, in which each of the partitions are reduced to only the sub-arrays required by the query. Since this process occurs during InputSplit construction, the data represented by the excluded portion of the logical space will never be read from disk. Put simply, *NoScan* is able to remove from consideration data that does not contribute to the query, reducing the total amount of data read by *Map* tasks.

We evaluated the impact of the *NoScan* optimization with query **Example 1**, whose constraining space has 80% selectivity, and the results show that 80% of the total data was being read (106 GB vs 132 GB) when NoScan was turned on. The impact on run-time can be seen in Table 5.2: Baseline + NoScan ran 11 minutes ($\dot{8}5$%) faster than just Baseline is used and 4 minutes (13%)faster in the experiments with a *Combiner*.

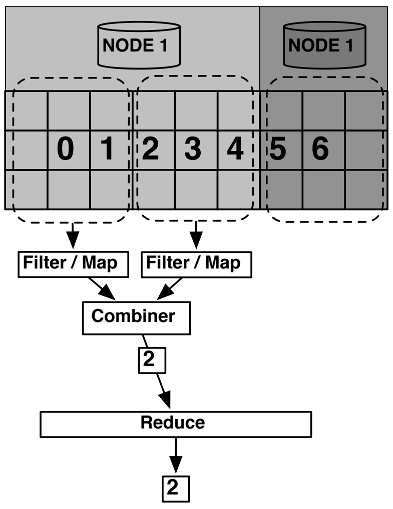
5.3 Holistic-aware Partitioning

While the holistic *Combiner* can evaluate a holistic function when the entire input is contained in the InputSplits on a single node, the partitioning of the input space ($\mathcal{T}$) is usually unaware of the query being processed, and thus the ability for the holistic *Combiner* to provide a benefit is probabilistic; it only occurs when all of the InputSplits containing portions of a given instance of the extraction shape happen to be scheduled on the same node. In cases where the set of all InputSplits containing portions of a given instance of the extraction shape are not processed by *Map* tasks on the same node, the application of the specified function must be deferred until the key/value pairs for that key are processed at a *Reduce* task.

To account for small misalignments that would otherwise prevent a *Combiner* from



(a) Query-agnostic partitioning



(b) Query-aware partitioning

Figure 5.3: Holistic-aware partitioning example

being used for holistic functions, we experimented with incorporating knowledge of the query into the InputSplit generation process. Specifically, we generate InputSplits that are either even multiples or even divisors of the specified extraction shape. This allows an InputSplit to contain multiple full instances of the extraction shape or multiple InputSplits to hopefully be assigned to the same node so that their output can be combined. Figure 5.3b shows how adjusting the size of the InputSplits allows all of the data for the lighter region to be placed on $NODE1$, thereby enabling the *Combiner* to reduce the data prior to it traversing the network. Note that Hadoop does not guarantee that both InputSplits will be assigned to $NODE1$. A different execution of the same query may result in the middle split being assigned to another node, in which case the holistic *Combiner* could not be employed. In practice, it is best to make InputSplits that are multiples of the extraction shape if possible as only key/value pairs that are in the same InputSplit are guaranteed to be processed by the same *Map* task (and therefore the same *Combiner*).

5.4 InputSplit Manipulation Experiments

A thorough description of the cluster used to run these experiments can be found in Appendix A.

To evaluate the query run-time impacts of the different partitioning and placement

Test Name	Local Read (%)	Temporary Data (GB)	% CPU Utilization	Run Time (Minutes)	Time σ (%)
No Holistic Combiner					
Baseline	9.3	2,586	34.7	129	7
Baseline +NoScan	9.2	2,588	34.3	118	3
Chunking & Grouping	80	2,608	24.3	145	5
Physical-to-Logical	88	2,588	29.9	138	5
Holistic Combiner with Baseline					
Baseline	9.5	107	79.1	31	2
Baseline +NoScan	9.5	107	80.7	27	1
NoScan +*HaPart*	8.8	107	81.3	27	7
HaPart	8.6	107	79.3	31	0.8
Holistic Combiner with Local-Read Optimizations					
Chunking & Grouping +*HaPart* +NoScan	70.7	116	84.7	25	0.4
Chunking & Grouping +NoScan	79.3	188	83.1	26	1
Physical-to-Logical +NoScan	88.1	196	82.8	27	6

Table 5.1: **Overview of Query 1 Results** Run-times of tests without the holistic Combiner optimization are dominated by writes to temporary storage. All other runs are bound by CPU (with `iowait` times $< 1\%$). Runtimes are averages from 10 executions with the standard deviation, as a percent of the average, is reported in the right-most column. Abbreviations: *HaPart*: holistic-aware partitioning.

approaches as well as the optimizations for holistic queries, we present results from query 1 (Figure 4.9a), which applies a median function, and query 2 (Figure 4.9b), which applies an average function. The holistic combiner and holistic-aware partitioning have a significant impact on run-time for query 1 — roughly an order of magnitude. As Table 5.1 shows, tests using the holistic Combiner optimization are largely CPU-bound while others are not, implying that those tests are waiting on IO. We were admittedly surprised by the extent of the performance impact of the *Combiner* until we realized that even in a cluster constrained by network bandwidth, intermediate data size ("Temporary Data" in the table) can have a significant impact on total query execution time due to node-local IO contention.

A large volume of intermediate data causes a significant increase in temporary storage writes due to buffer spills and external sorting, which can occur at each step of a *MapReduce* computation. In our experimental setup, each server uses a single disk (shared with the OS) for temporary data while HDFS data is striped over 3 SATA drives. This configuration can lead to a relative IO bottleneck for temporary data. In the case of Query 1, successfully applying the *Combiner* at the *Map* node, rather than at the *Reducer*, results in a reduction of 25,920 data values, and their associated metadata, to a single value. This dramatic reduction in key/value pairs results in a commensurate reduction in temporary data written, shown in the "Temporary Data" column of Table 5.1, that translates into significant reductions in query execution time.

An interesting result gleaned from these experiments is that it is sometimes preferable to allow *Map* task read locality to be diminished if it results in an increase in *Combiner* efficacy. Compare the results for the *Chunking & Grouping +HaPart* +NoScan experiment with the results for the *Chunking & Grouping* + NoScan experiment. When holistic-aware partitioning is used, the rate of data read locality drops from 79.3% to 70.7% and the size of the Temporary Data produced drops from 188 GB to 116 GB. Total run-time also dropped by 1 minute, representing a performance improvement of approximately 4%.

Some combinations of optimizations were not tested because their goals would have been diametrically opposed. For example, testing *Physical-to-Logical*, which strives to create InputSplits that are perfectly aligned to blocks in the underlying distributed file system, in conjunction with holistic-aware partitioning, which reduces data locality in an effort to increase the efficacy of combiners, would have required allowing one behavior to override the other, thereby producing a mediocre result of questionable utility. We strove to produce experimental results that would highlight the characteristics of optimizations, and combinations of optimizations, that were complimentary, thereby showcasing the largest impact possible for the given approach(es).

Test Name	Local Read %	Initial Read (GB)	Temporary Data (GB)	Run Time (minutes)	Time σ (%)
Baseline	3	132	569	66	2
Baseline +NoScan	3	106	569	63	2
Chunking & Grouping +NoScan	84	106	498	55	3
Physical-to-Logical +NoScan	93	106	498	56	3

Table 5.2: **Runtimes for Query 2** All experiments produced 7.7 GB of *Reduce* task output and all tests use a *Combiner*. Values are averages over ten runs.

Table 5.2 presents results for Query 2 with all the combinations from Table 5.1 that include a *Combiner*. (HaPart query results were omitted as that optimization is not enabled for non-holistic functions).

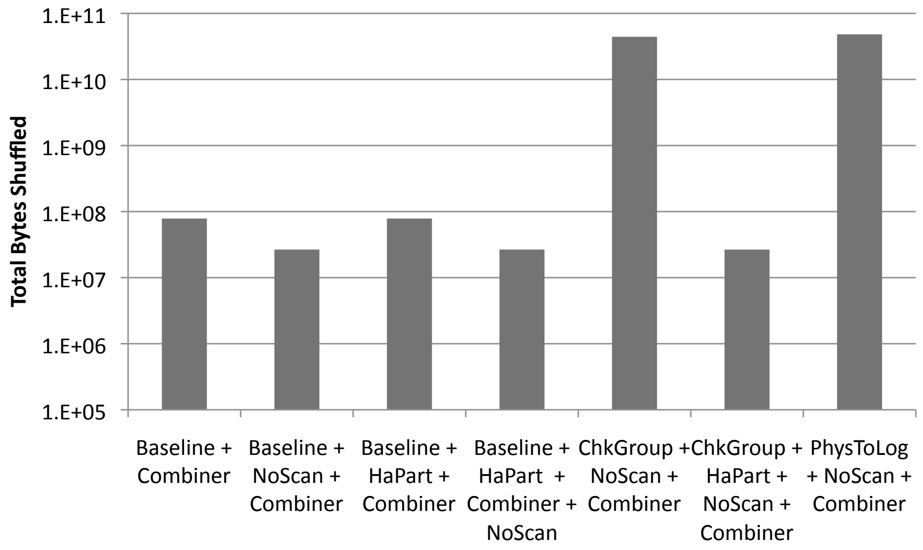


Figure 5.4: Bytes shuffled in tests with holistic combiner (note the y-axis log scale).

The interplay between the different salient metrics and their effects on run-time become more evident when comparing the results from both queries. Table 5.3 shows select experiments from both queries as well as an execution of a single-threaded program that completes Query 2 serially by reading data stored locally on a single hard drive. As the amount of intermediate data written (Temp Data) and *Reduce* output increases, so does the observed run-time. This is expected, as increases in both translate into more IO, increased network traffic and more data that must be processed in the *Combine* and *Reduce* functions.

5.5 The Nature of Scientific Data

The sympathetic alignment of data stored in scientific file formats to the queries run over them, visible in our test data, stems from scientific data typically correlating to phenomena in the natural world. Events or measurements that are "near" each other (for some value of near) in the logical model are often related and therefore evaluated together. As an example, consider a time series: time-steps that are near each other in the logical model are temporally close and therefore more likely to be evaluated together than data that is more temporally distant. This idea is predicated on logical model proximity translating into proximity in the underlying storage system, but our observation is this assumption tends to be valid in practice.

Test Name	Local Read %	Temp Data (GB)	Reduce Output (MB)	Run Time (min)
Baseline +NoScan +Combiner				
Holistic	9.5	107	0.1	27
Regrid	3	569	7,735	63
Chunking & Grouping +NoScan +Combiner				
Holistic	79	188	0.1	26
Regrid	84	498	7,735	55
Physical-to-Logical +NoScan +Combiner				
Holistic	88	196	0.1	27
Regrid	93	498	7,735	56
Serial Program				
Regrid	100	NA	7,735	3,130

Table 5.3: **Impact of Data Volumes on Runtime** For comparable queries, run-time is dictated by the volume of data generated. The serial regrid program ran on a single node and accessed the data from local storage.

5.6 Section Summary

In summary, this section shows that our optimizations can provide significant performance improvements when applying a holistic function. Additionally, the nature of the query being applied, specifically the amount of data reduction produced by the *Map* task and *Combiner*, will have a significant impact on run-time. Our experiments also found that sometimes it is preferable to trade a reduction in *Map* task data locality in return for an increase in *Combiner* efficacy.

Chapter 6

Redefining the Communications Model

This section contains research into altering Hadoop's communication model via a deeper integration of the knowledge of the structure of the data being processed than in previous sections. Specifically, this knowledge is leveraged to make decisions at points that are typically data agnostic, requiring alterations to core portions of Hadoop. The modified portions include: data assignment to KEYBLOCKs, deciding when to start a *Reduce* task and during the scheduling of *Map* tasks. In altering Hadoop at a more fundamental level, our research enables novel performance optimizations and new functionality while maintaining the generality of the *MapReduce* framework.

6.1 Communication in Hadoop

To recap from Section 3.2.1, when a Hadoop job is submitted by a client, the central coordinator generates the corresponding InputSplits and then waits for nodes to request *Map* tasks, attempting to assign InputSplits to one of their preferred nodes so as to achieve data locality. After processing the data indicated by its InputSplit, a *Map* task produces a (potentially empty) data set for every *Reduce* task and then notifies the *JobTracker* which, in turn, notifies every *Reduce* task. After receiving a notification, *Reduce* tasks opportunistically copy intermediate data from completed *Map* tasks and merge that data with previously copied data. By copying data as *Map* tasks complete, network IO can be overlapped with the computation of other *Map* tasks, resulting in performance improvements. A *Reduce* task continues copying *Map* task output until the *Reduce* task has its assigned data from every *Map* task. At that point, it is free to begin processing its data, writing its final output to stable storage as it finishes. The all-to-all task communication pattern, which is based on a worst-case assumption of global data dependencies, is shown in Figure 6.1 with representative wall-clock run-times of the tasks shown in Figure 6.4a.

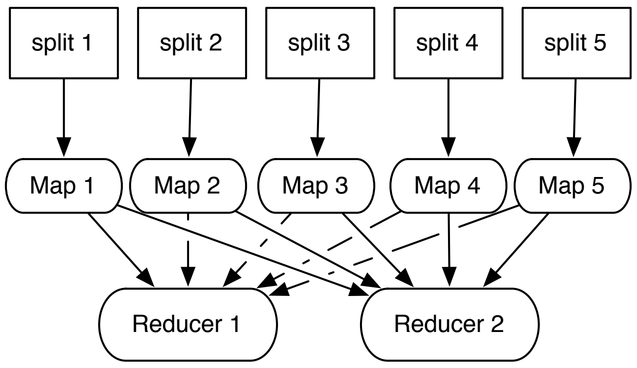


Figure 6.1: Default communications pattern for the *MapReduce* program in Figure 6.3.

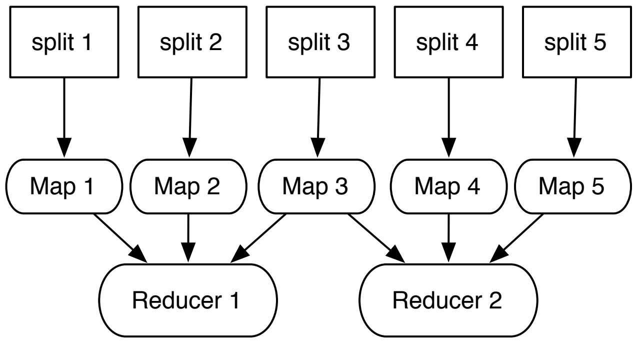


Figure 6.2: Enhanced communications pattern for the *MapReduce* program in Figure 6.3.

A *Map* task's generation of intermediate data highlights the fact that the *Map* and *Reduce* tasks process key/value pairs while the framework naively moves bytes (Section 4.2). As intermediate key/value pairs are produced, the key for each pair is passed through the hash function, the output indicating which KEYBLOCK the data is destined for and therefore which intermediate data set the key/value pair should be written to. When a *Map* task finishes processing its assigned input, it closes the file containing intermediate data and writes out a small amount of meta-data for each KEYBLOCK, including the number of bytes in the dataset, to a header. After a *Reduce* task has acquired its set of intermediate data from a *Map* task, it is still not aware of the contents of the data set, merely the data contained in the header. It is not until the *Reduce* task starts processing its data that the file containing the intermediate data is read and the identity of the keys contained within the file are known.

6.2 Communication Barriers

In Section 4.2, we discussed our research into achieving high rates of data locality while generating InputSplits at the logical level and we explored the impact of InputSplit construction on the application of *Combiners*, specifically holistic *Combiners*, in Section 5. Stepping back to consider the communications model for *MapReduce*, the generation of InputSplits are also of interest because they dictate what data a given *Map* task is guaranteed to see together since an InputSplit is either successfully processed in its entirety or rescheduled and re-executed on another node while no two InputSplits are guaranteed to be scheduled on the same node. Additionally, the RecordReader, which consumes InputSplits, represents one of the two points at which the flow of data through a Hadoop query is obscured (Section 3.3.1).

In Section 4, we show the necessity of defining I_i in terms of logical coordinates, creating a situation where both RecordReader input and output are defined at the same level of abstraction and also in the same logical space (coordinates in the logical space of K). This alignment of abstraction levels enables us to reason between a given I_i and the keys it will produce for the corresponding *Map* task, as well as correlating *Map* task inputs to a given I_i. In fact, I_i and the set of all keys that RR(I_i) (Figure 3.1) will produce are equivalent. Given that we can translate from the total input to the set of I_i that will be consumed by the set of all *Map* tasks, the system can, at the initialization of a Hadoop job, calculate the set of keys in K that will actually be processed by the set of all *Map* tasks; that set of keys is denoted $\mathbb{K}^{\mathcal{T}}$. This newfound knowledge is significant as $\mathbb{K}^{\mathcal{T}}$ can be used in place of the entire keyspace K when reasoning about the flow of data, thereby enabling us to make decisions based on the keys that actually exist in the data set being processed, rather than the set of all representable keys.

6.2.1 Correctness

The primary ordering constraint in *MapReduce* is that a *Reduce* task only processes data for a given key when it has all of the key/value pairs with that particular key. Given an inability to make assumptions about the behavior of the hash function for intermediate data, and the resulting assignment of data to keyblocks, a worst-case assumption of any *Map* task creating output for any *Reduce* task is required. This assumption necessitates a barrier between the end of the last *Map* task and the beginning of any *Reduce* task for a given *MapReduce* program. The barrier guarantees that all values for a given key will be processed at the same time with the side effect of preventing *Reduce* tasks from beginning to process their assigned data until the output from all *Map* tasks is available. Were a *Reduce* task to start

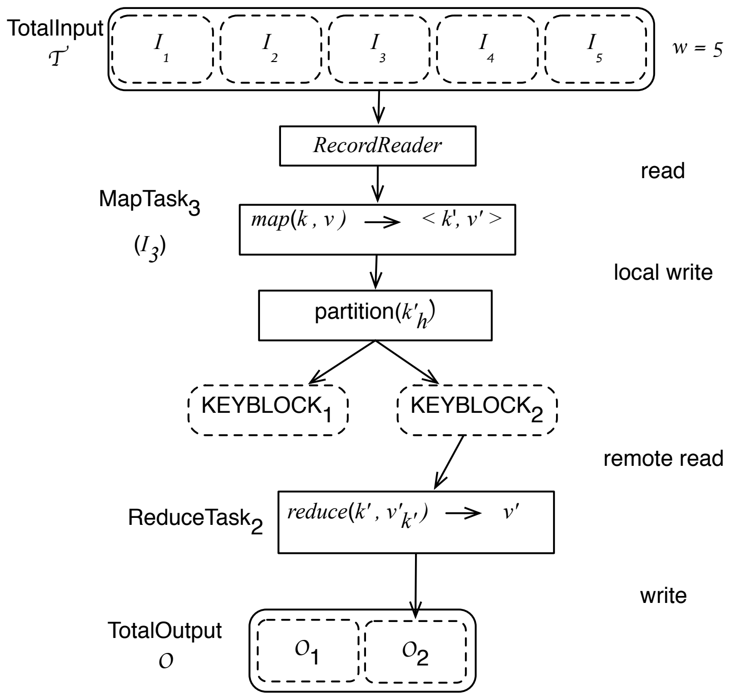


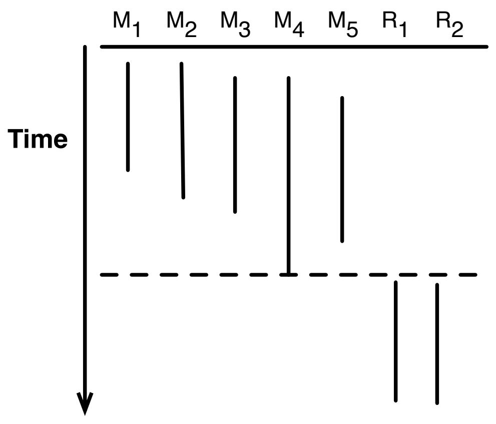
Figure 6.3: Annotated data flow for a *MapReduce* program.

prior to the completion of all *Map* tasks (i.e., the global barrier), then it may produce final results that are incomplete or incorrect if any of the remaining *Map* tasks for which it does not have intermediate data produces key/value pairs assigned to that *Reduce* task's KEYBLOCK. Figure 6.4a depicts the barrier, which was discussed in Section 6.1.

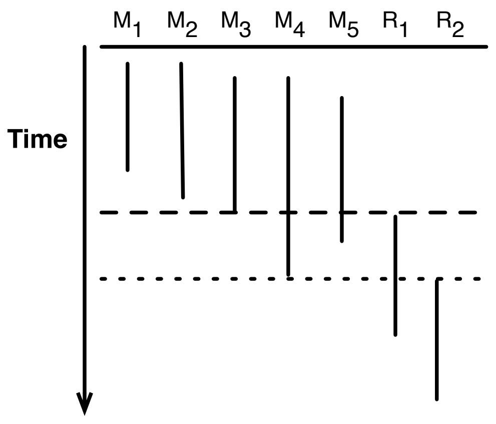
6.2.2 Skew in KEYBLOCK Sizes

KEYBLOCK sizes are a function of the distribution of the observed intermediate keys ($\mathbb{K}'^{\mathcal{T}}$) combined with the hash function. Assuming a modulo-style hash function, a relatively even distribution of keys in $\mathbb{K}'^{\mathcal{T}}$ translates into a similarly even distribution among KEYBLOCKs.

Skew in KEYBLOCK sizes can cause divergence in run-times between *Reduce* tasks, as shown here [109]. Given that a *MapReduce* job typically has many times fewer *Reduce* tasks than *Map* tasks and that the global barrier causes *Reduce* tasks to all start at the same time, a disparity across *Reduce* task run-times typically has a larger impact on total *MapReduce* job run-time than *Map* task skew. Chapter 9 includes references to previous work on avoiding skew that provide further details on the effect of skew on *MapReduce* job run-time.



(a) Hadoop task run-times (the dotted line is a global barrier)



(b) Same task run-times when using actual dependencies (note per-*Reducer* barriers)

Figure 6.4: (a): By default, *Reduce* tasks must wait for all *Map* tasks to complete prior to beginning their execution. (b): When *Reduce* tasks understand their actual data dependencies (e.g., R_1 depends on M_1, M_2, M_3 while R_2 depends on M_3, M_4, M_5), they can start after the last *Map* task they depend on finishes.

6.3 Replacing the Intermediate Data Partitioning Function

Given that we have access to the set of keys that will exist for a given query ($\mathbb{K}^{\mathcal{T}}$) and can map keys in that space onto the intermediate space (K') via the extraction shape, we can compute the set of intermediate keys that will exist, denoted $\mathbb{K}'^{\mathcal{T}}$. Given that most queries use their intermediate keyspace to order their output, we can also reason about the total output space ($\mathcal{O}$). For a particular $\mathbb{K}^{\mathcal{T}}$ and extraction shape, $\mathbb{K}'^{\mathcal{T}}$ can be calculated by dividing the length of each dimension in $\mathcal{T}$ by the entry in the corresponding dimension of the extraction shape. Since an extraction shape can map a single value in K into multiple values in K', multiple values in K into a single value in K', or a single value in K into a single value in K', $\mathcal{O}$ may be smaller, larger or the same size as $\mathcal{T}$, respectively. The $\mathcal{O}$ for a given $\mathcal{T}$ and extraction shape, abbreviated to 'es', is denoted $\mathcal{O}^{\mathcal{T}}_{\text{es}}$.

As part of extending Hadoop's communications model, we replace the default hash function with one that incorporates knowledge of $\mathbb{K}'^{\mathcal{T}}$, the extraction shape, and $\mathcal{O}^{\mathcal{T}}_{\text{es}}$. This new function, referred to as hash+, computes $\mathbb{K}'^{\mathcal{T}}$ and then partitions that, rather than K', into r KEYBLOCKs (where r is the configured number of *Reduce* tasks). This structure-aware intermediate data partitioning function provides several benefits, outlined in the following subsections.

6.3.1 Bounding Skew with hash+

Given that $\mathcal{O}_{\text{es}}^{\mathcal{T}}$ for a query over structured data is a fixed size (for the queries we're considering), hash+ can guarantee an upper bound on the skew in sizes between KEYBLOCKs. This is accomplished by the following process: 1) specifying an upper bound (possibly user-defined or derived from query details) on the permissible amount of variance between KEYBLOCKS, 2) creating an n-dimensional shape whose total size is smaller than the upper bound, 3) determining the total number of instances of the shape that exist in $\mathcal{O}_{\text{es}}^{\mathcal{T}}$, and 4) dividing the total count of those instances by the number of *Reduce* tasks. The final result is the number of instances of the shape (step 2) that should make up each KEYBLOCK. This process guarantees that each KEYBLOCK varies in size by at most one instance of the shape that was selected, which was chosen to be smaller than the initially specified maximum variance. This process is shown in Figure 6.5.

6.3.2 Creating Contiguous KEYBLOCKS with hash+

The hash+ function knows not only the size but also the contents of $\mathbb{K}'^{\mathcal{T}}$ and can therefore produce KEYBLOCKs consisting of keys that are contiguous in K'. This is accomplished by partitioning $\mathbb{K}'^{\mathcal{T}}$ in contiguous ranges in K', rather than using a modulo operator. Since most queries use the ordering of the intermediate keys to order their output, creating contiguous ranges of intermediate keys results in *Reduce* tasks writing out their final output in contiguous ranges. This stands in contrast to the output written when using the default, modulo-based approach. With the modulo function, a given *Reduce* task will write data spread throughout the output space ($\mathcal{O}$). Creating KEYBLOCKs that result in the associated *Reduce* task writing dense arrays, as opposed to writing randomly throughout $\mathcal{O}$, simplifies interacting with the total output at a later date and yields performance benefits as a result of requiring fewer writes than in the default case. The latter point assumes that the scientific library will convert logically dense arrays into efficient file accesses.

6.3.3 hash+ Performance

The hash+ function displays comparable run-times to the default hash class used by Hadoop, *org.apache.hadoop.mapred.lib.HashPartitioner*. We created a micro-benchmark that loaded a 25 MB file of HDF5 data consisting of 6,480,000 elements into memory and then hashed them with both functions. This test represents the time taken for a *Map* task to determine

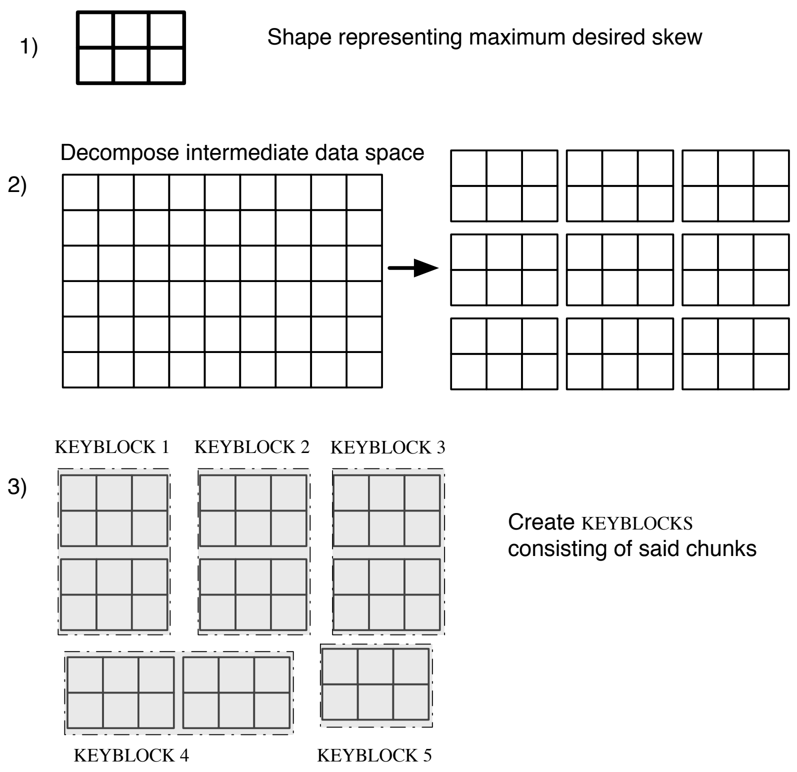
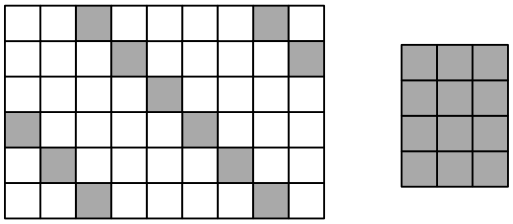

1)
Shape representing maximum desired skew
2)
Decompose intermediate data space
3)
KEYBLOCK 1
KEYBLOCK 2
KEYBLOCK 3
Create KEYBLOCKS consisting of said chunks
KEYBLOCK 4
KEYBLOCK 5

Figure 6.5: Bounding intermediate data size skew with hash+. In this example, we chose a maximum desired skew of 6 cells. The intermediate data space was broken into instances of our 6-cell shape, then the desired number of KEYBLOCKS (5) were constructed by combining the set of instances of our 6-cell shape.

the KEYBLOCK for 6.48 Million intermediate key/value pairs. Over ten runs, the default hash function has an average execution time of 200 ms and a standard deviation of 18.8 ms. Our hash+ function had an average execution time of 223 ms with a standard deviation of 21 ms, representing an approximately 11% decrease in performance when using our hash+ function. The effect in absolute terms is negligible, given that the measured execution times of *Map* tasks ranges from tens of seconds to tens of minutes. It is conceivable that hash+ could be optimized to take advantage of the fact that it assigns keys in contiguous ranges, and therefore may not need to recompute the KEYBLOCK for every key, but we found the current performance sufficient for our needs.

1	2	3	4	5	1	2	3	4
5	1	2	3	4	5	1	2	3
4	5	1	2	3	4	5	1	2
3	4	5	1	2	3	4	5	1
2	3	4	5	1	2	3	4	5
1	2	3	4	5	1	2	3	4

(a) KEYBLOCK assignment when using a modulo hash function



(b) *Reduce* task write patterns for modulo and hash+ KEYBLOCKS

Figure 6.6: (a): KEYBLOCK assignment of the intermediate dataset shown in Figure 6.5 if the default modulo hash function is used. (b): Comparison of the writes done by *Reduce* task 3 for a KEYBLOCK generated by modulo hash (left) and hash+ (right)

6.3.4 Clarifying *MapReduce*'s Dataflow

Given that hash+ is based on the total input ($\mathcal{T}$) and the extraction shape, its effects are deterministic. This means that the input to and output from the hash function can be correlated, removing the second point of opaqueness from Hadoop's communication model (Section 3.3.1). We are now able to reason about the flow of data from initial input ($\mathcal{T}$), through the intermediate keyspace (K') and into the output space ($\mathcal{O}_{\text{es}}^{\mathcal{T}}$). This, and the resultant optimizations it enables, is one of the primary contributions of this thesis.

6.4 Breaking Down Barriers

Combining the additional knowledge provided by the meta-data from scientific access libraries, the extraction shape and our hash+ function, it is possible to extend Hadoop to calculate the set of keys in K that will produce intermediate data for a given KEYBLOCK as well as the specific set of data that will be assigned to a given *Reduce* task. This ability to reason about elements in the initial input and their translation, via the *Map* and hash functions, into elements assigned to *Reduce* tasks is a powerful new feature.

The value $\mathcal{I}_\ell$ represents the set of I_i (InputSplits) that, when processed by a RECORDREADER and its associated *Map* task, will produce at least one intermediate key/value pair that will be assigned to KEYBLOCK$_\ell$. $\mathcal{I}_\ell$ is the actual data dependency of KEYBLOCK$_\ell$ and our modified version of Hadoop uses that as a barrier, rather than assuming a dependency on all I_i (as is classically done in Hadoop). By using this new, task-specific barrier, *Reduce* tasks can begin processing data assigned to their KEYBLOCK as soon as the last I_i in their $\mathcal{I}_\ell$ completes while still maintaining *MapReduce*'s correctness guarantees (detailed in Subsection

6.5.1). A definition of $\mathcal{I}_\ell$ is shown in Formulation (6.3) and the process for computing $\mathcal{I}_\ell$ is described in the remainder of this section. The ability to calculate $\mathcal{I}_\ell$ enables the more precise communications model depicted in the bottom portion of Figure 6.2. The effect of this new model on scheduling is shown in Figure 6.4(b) where *Reduce* task R_1 is allowed to start prior to *Map* tasks M_4 and M_5 completing because all of R_1's data dependencies are fulfilled.

When a *MapReduce* job begins, the input is specified and the range of keys in K that are present ($\mathbb{K}^{\mathcal{T}}$) is determinable via the meta-data provided by the scientific access libraries. Once $\mathbb{K}^{\mathcal{T}}$ is partitioned into $\mathcal{I}$, then a given I_i is itself $\mathbb{K}_i^{\mathcal{T}}$. With this additional information, we can re-frame Formulation (3.1) as Formulation (6.1).

$$\begin{aligned}
&\text{for } m_i = |I_i| \,, \\
&\mathrm{RR}(I_i) \;= \\
&\qquad \left\{ \langle k_1, v_1 \rangle^i, \langle k_2, v_2 \rangle^i, ..., \langle k_{m_i}, v_{m_i} \rangle^i \right\} \\
&\langle k_j, v_j \rangle^i \in \mathbb{K}_i^{\mathcal{T}} \times V \quad \forall j \in \{1, 2, ..., m_i\}, \\
&\bigcup_{j=1}^{m_i} k_j = \mathbb{K}_i^{\mathcal{T}} = I_i
\end{aligned} \tag{6.1}$$

An explicit extraction shape enables us to easily understand how a key in K translates into key(s) in K' and likewise from $\mathbb{K}^{\mathcal{T}}$ to $\mathbb{K}'^{\mathcal{T}}$. The term $\mathbb{K}'^{\mathcal{T}}_i$ denotes the keys in $\mathbb{K}'^{\mathcal{T}}$ that result from applying the *Map* function to the key/value pairs in the corresponding $\mathbb{K}_i^{\mathcal{T}}$. In practice, $\mathbb{K}'^{\mathcal{T}}_i$ is easily calculated by combining the extraction shape with $\mathbb{K}_i^{\mathcal{T}}$.

The set of all keys in $\mathbb{K}'^{\mathcal{T}}$ that will be assigned, via the hash+ function, to KEYBLOCK$_\ell$ is denoted $\mathbb{K}'^{\mathcal{T}}_\ell$. When *Reduce* tasks start up, they are supplied an ID, indicating their role amid the total set of *Reduce* tasks. A *Reduce* task can use that ID, the hash+ function, and $\mathbb{K}^{\mathcal{T}}$ to calculate its $\mathbb{K}'^{\mathcal{T}}_\ell$, which is necessary to determine I_ℓ for a KEYBLOCK, as shown in Section 6.5.1.

6.5 Early Results

The ability to create per-task barriers, described in the previous section, opens the door to actually using these smaller barriers within Hadoop and starting *Reduce* tasks once their individual barriers are satisfied. In this section, we explain how we implemented this in Hadoop and how we maintain all of the guarantees set out in the original *MapReduce* paper [30].

6.5.1 Maintaining Correctness

As has been discussed, an extraction shape represents how a key in K is mapped to a key in K'. This mapping is accomplished by a series of multiplications involving the K key and the extraction shape. Given this, it is trivial to also map from K' to K; the multiplication is simply inverted. The function that maps from K' to K is referred to as $ExSh^{-1}()$ (short for "inverse extraction shape").

Combining $\mathbb{K'}_\ell^{\mathcal{T}}$ (the set of all keys in K' that will be assigned to KEYBLOCK$_\ell$) with the ability to *Map* from K' to K, a *Reduce* task can compute the set of keys in $\mathbb{K}^{\mathcal{T}}$ that produce intermediate data destined for its KEYBLOCK, denoted $\mathbb{K}_\ell^{\mathcal{T}}$. A definition of $\mathbb{K}_\ell^{\mathcal{T}}$ is shown in Formulation (6.2).

$$\begin{gathered}\text{for KEYBLOCK}_\ell, \\ \mathbb{K}_\ell^{\mathcal{T}} = \bigcup k \mid \exists k' \in \mathbb{K'}_\ell^{\mathcal{T}} \text{ such that } ExSh^{-1}(k') = k\end{gathered} \tag{6.2}$$

The ability to calculate $\mathbb{K}_\ell^{\mathcal{T}}$ 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 3.4(b)). The set of keys in K that map to the same key in K' may fall 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 the original global barrier.

$$I_\ell = \forall I_i \mid \exists\ k \text{ such that } k \in \mathbb{K}_\ell^{\mathcal{T}} \ \wedge\ k \in I_i \tag{6.3}$$

We identified two methods for resolving the ambiguity as to the number of $\langle k, v\rangle$ that were combined to form a particular $\langle k', v'\rangle$:

1) $\mathcal{I}_\ell$, the set of inputs that will produce data destined for KEYBLOCK$_\ell$, can be computed and that can be used as a proxy for $\mathbb{K}_\ell^{\mathcal{T}}$, since it is a super-set (see Formulation (6.3)).

2) each key/value pair in the K' space can be annotated to include the number of elements in the K space that it represents. Each *Reduce* task can then keep a running tally of the number of $\langle k, v\rangle$ represented by the $\langle k', v'\rangle$ it receives. When it has accumulated data representing all $\langle k, v\rangle$ in its $\mathbb{K}_\ell^{\mathcal{T}}$, processing can safely begin.

SIDR uses the former method and also implemented the annotations required for the latter method as a means of validating the system's correctness.

In Hadoop, all $\langle k', v' \rangle$ produced by a particular *Map* task and assigned to the same KEYBLOCK are written into the same file. We added a field to the header data for that file indicating how many $\langle k, v \rangle$ are represented by the set of all $\langle k', v' \rangle$, for a given KEYBLOCK. As mentioned in Section 6.1, when a *Reduce* task retrieves its intermediate data, it only has access to the information in the header. With our additions to the header data, a *Reduce* task can now track the count of all $\langle k, v \rangle$ represented by the contents of the files containing its intermediate data without having to read and parse the data.

6.5.2 Benefits of Early Results

The ability for *Reduce* tasks to begin processing data as soon as possible allows for additional overlapping of computation with IO beyond that already present in *MapReduce*. Previous research [27, 28, 141] indicates that this additional overlapping will translate into shorter total run-times for *MapReduce* jobs. Secondary benefits include: resources that are typically consumed by a *Reduce* task while waiting for unnecessary *Map* tasks to complete are now made available for other work and the storage occupied by intermediate data for completed KEYBLOCKs can be released prior to job completion.

6.5.3 Altering Task Scheduling in Hadoop

In order to realize the benefits of starting *Reduce* tasks once their individual barriers were fulfilled, we were required to make alterations to the default Hadoop scheduler, *JobQueue-TaskScheduler*. As background, here is how the Hadoop scheduler normally operates. When a Hadoop job begins, all *Map* tasks are added to a tree structure with the lowest level of nodes corresponding to specific servers in the cluster and higher level nodes being aggregations of servers (rack, entire cluster). Tasks are first added to the node(s) in the tree representing servers where the data for that task is stored (to achieve data locality). That same task is then added to the parent of those nodes (representing the rack that each data-local server resides in) and then to the parent of that node (representing the pool of all servers in the cluster). When a server requests a new *Map* task, this tree is traversed, starting at the leaf-node representing that server and working its way up, with the first runnable task encountered being returned. This process results in tasks that would be local to that server being returned if they exist. Absent that, a rack-local task is returned if one exists or else, as a last resort, any runnable task is returned. In contrast, *Reduce* tasks are scheduled in monotonically increasing order of their IDs as slots become available. Additionally, *Reduce* tasks are not scheduled until some minimum number of *Map* tasks have started running.

In terms of *Reduce* tasks starting once their data dependencies are met, the interaction between the two default scheduling polices results in those dependencies being met haphazardly since no effort is made to co-schedule *Reduce* tasks with the *Map* tasks they depend on. In practice, the odds of a *Reduce* task having its data dependencies met was a function of the number of *Map* tasks it depended on combined with how many *Map* tasks had completed up to that point; in other words, it was strictly probabilistic.

In the SIDR paper [16], the scheduling process was altered such that a *Map* task was only available to be scheduled if at least one *Reduce* task that depended on it was already running (dependency information was generated by the *FileInputFormat* class during job submission and then read by *JobInProgress* when scheduling tasks). The actual scheduling algorithm was unchanged, but rather was not informed of *Map* tasks until those tasks would contribute to a running *Reduce* task. This co-scheduling creates the desired effect of correlating *Map* tasks with running *Reduce* tasks. Along with this change, we also configured Hadoop to start scheduling *Reduce* tasks immediately, since a *Reduce* task starting is now a predicate for *Map* tasks being scheduled to run.

Experiments conducted with our modified scheduler showed that even though we were artificially limiting the pool of *Map* tasks that were eligible to be scheduled at any point in time, rates of *Map* task data locality (*Map* tasks running on one of the hosts that our placement strategy had decided was optimal) was as high or slightly higher than with the default Hadoop scheduler.

6.6 (In)Applicability of Our Work in Hadoop

The approaches described in Section 6.4 and Section 6.5 are not applicable to Hadoop in the general case but rather are only possible given the additional knowledge we can extract from the meta-data in scientific file formats and for the classes of queries that we focus on (Section 3.5). This limitation stems from the fact that Hadoop's default hash function is unable to easily map either from K to K' or from K' to K during the initialization of a Hadoop job. A mapping from K to K' is made difficult by the fact that the modulo-based hash function operates on the binary representation of each intermediate key. This implementation-specific knowledge complicates computing the KEYBLOCK for each key and likely requires the serialization of each individual key (a computationally expensive proposition) independent of the serialization that occurs when the query actually executes. Creating a mapping from K' to K is even more difficult because the application of a modulo operator effectively destroys knowledge (specifically, what the hash function input was). To create this mapping for a given

Reduce task, every possible K that could map to that *Reduce* task must be computed (by running every representable key through the modulo function and maintaining a list of those that mapped to the task in question) and then the intersection of that set with $\mathbb{K}^{\mathcal{T}}$ needs to be calculated. This approach is fairly infeasible, given practical limits on computation and memory, and even if it were practical, it would require an entry per key be stored. Our use of contiguous ranges of intermediate keys coupled with an easily invertible function for assigning intermediate keys to KEYBLOCKs is what enables a tractable approach for creating per-task barriers.

6.7 Reduced Barrier Experiments

This section chronicles experiments conducted as part of the SIDR project that show the effects of changes made to Hadoop's task scheduling, intermediate hash function and communication model discussed in this chapter. Details on the cluster hardware can be found in Appendix A.

6.7.1 Query 3: Median Function

This experiment applied a median function over a 4-dimensional data set with dimensions of lengths $\{7200, 360, 720, 50\}$ and using an extraction shape of $\{2, 36, 36, 10\}$ (meta-data representing this file appears in Figure 6.7). The data set can be thought of as 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 18° latitude region, in cross-sections of 10 elevation steps. The total file size is 358 GB.

```
dimensions:            variables:
   time = 7200;           int windspeed(time, lat, lon, elev);
   lat = 360;
   lon = 720;
   elev = 50;
```

Figure 6.7: Metadata for the dataset used in queries 3 & 4.

We first compare Hadoop, SciHadoop [18], and SIDR [16] with the total number of *Reduce* tasks configured to 22, with the choice of 22 *Reduce* tasks owning to best practices dictating that 90% of the total node count is a reasonable number of *Reduce* tasks. Figure 6.8 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.

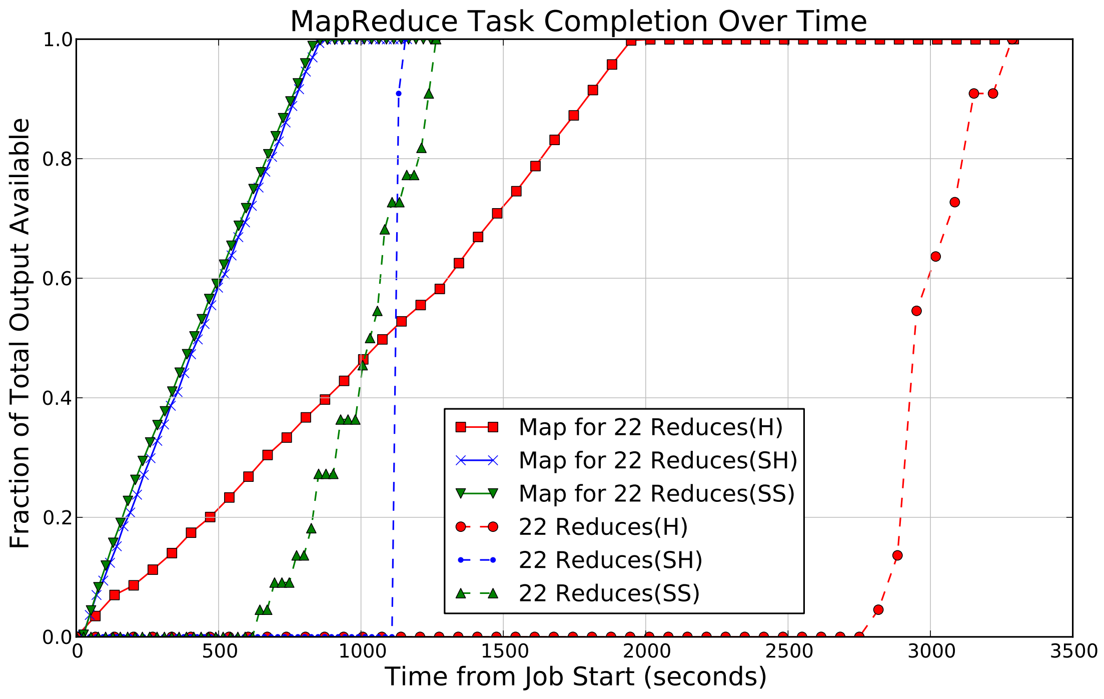


Figure 6.8: *Map* and *Reduce* task completion for Hadoop(H), SciHadoop(SH) and SIDR (SS)

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 [18]. The gap between the first result in SIDR and the first results in SciHadoop and Hadoop stems from 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. SIDR's slightly longer run-time is due to its use of contiguous KEYBLOCKS. The last SIDR *Reduce* task has to copy, merge and process all of the data in the last 4.5% (1/22nd) of the *Map* tasks. In SciHadoop and Hadoop, the output of those *Map* tasks would be spread evenly across all *Reduce* tasks. Following experiments show that SIDR can produce total query run-times that are shorter than SciHadoop's by increasing the number of *Reduce* tasks.

In early versions of our SIDR paper, reviewers were concerned that our changes to the

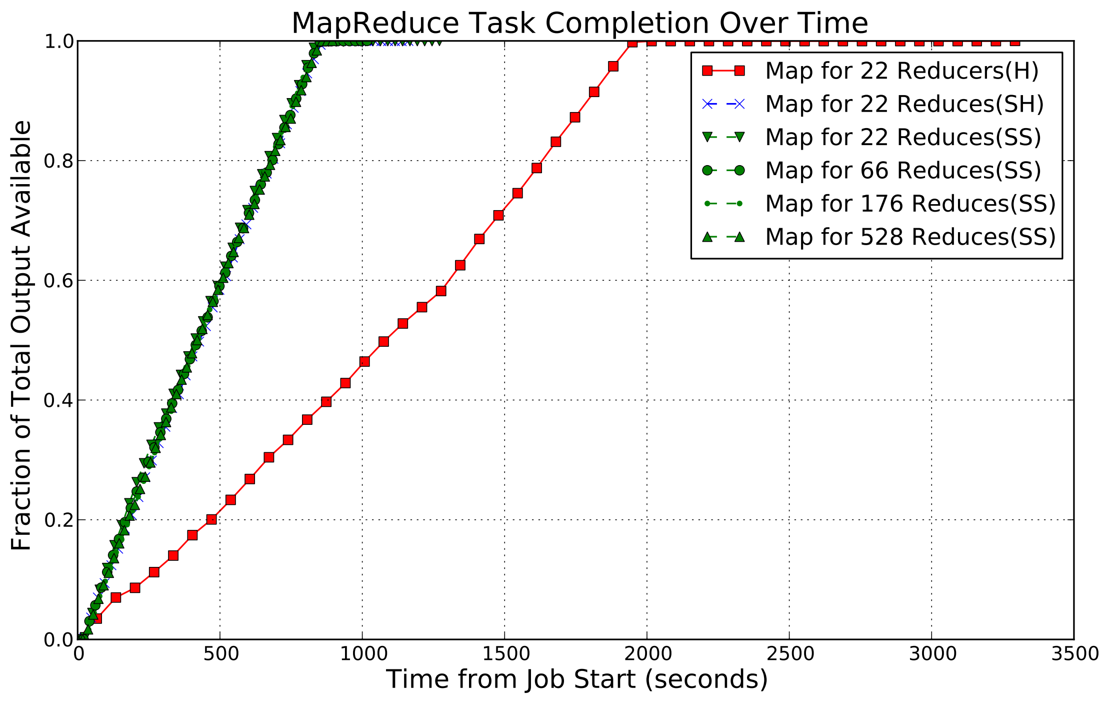


Figure 6.9: *Map* task completion for a fixed query run with Hadoop (H) using 22 *Reduce* tasks, SciHadoop (SH) using 22 *Reduce* tasks and SIDR (SS) running 22, 66, 176 and 528 *Reduce* tasks.

scheduler and communication patterns could have a detrimental effect on *Map* task execution times. Figure 6.9 shows a graph of *Map* task execution time as the number of *Reduce* tasks is varied. This graph does not change in any appreciable manner as the number of *Reduce* tasks varies, with the exception of Hadoop being less efficient than the other approaches. This result indicates that our changes do not have a detrimental impact on *Map* task performance. It also shows that varying the number of *Reduce* tasks does not affect the run-time of *Map* tasks.

Next, we present the same query and dataset while varying the number of total *Reduce* tasks. By fixing the query and the input, the output produced by the query will likewise remain fixed. Increasing the number of *Reduce* tasks will result in each *Reduce* task having a proportionately smaller amount of data assigned to its KEYBLOCK, which will commensurately decrease its data dependencies. Figure 6.10 shows this experiment where the query is run with 22 *Reduce* tasks for SciHadoop as well as 22, 66, 176 and 528 *Reduce* tasks for SIDR. Given the size of our dataset (348 GB) and the configured HDFS block size (128 MB), the partitioning scheme creates 2,781 InputSplits for this job (since there are roughly as many InputSplits created as there are blocks occupied by the specified dataset).

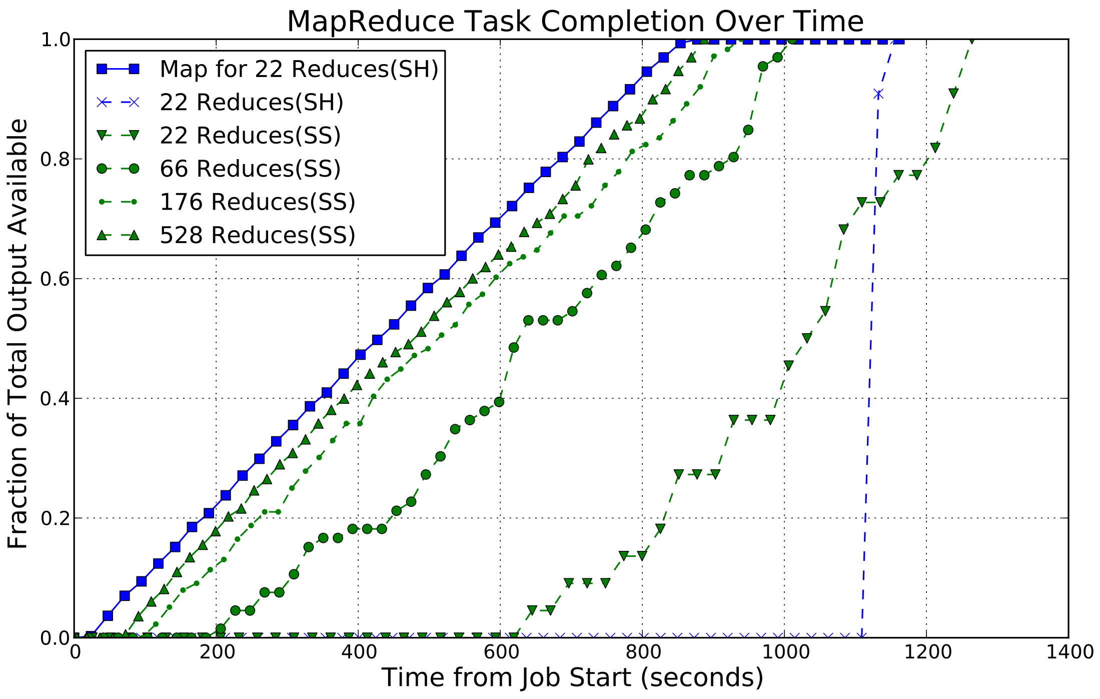


Figure 6.10: *Reduce* task completion for a fixed query run with SciHadoop (SH) using 22 *Reduce* tasks and SIDR (SS) running 22, 66, 176 and 528 *Reduce* tasks.

The results in Figure 6.10 display some interesting interactions that become apparent as the number of *Reduce* tasks is scaled. With SIDR, as the number of *Reduce* tasks increases, the time to first result and the total job execution time both decrease. The same query run with SIDR finishes 29% faster than with SciHadoop and nearly three times faster than Hadoop (shown in Figure 6.8). These performance improvements are attributable to decreasing the size of the data dependency for each *Reduce* task, thereby allowing *Reduce* tasks to start sooner and overlap more IO with computation. Additionally, as the amount of data that each *Reduce* task is assigned shrinks, there are fewer buffer overflows resulting in less disk-based merge-sorting (with its associated latency) and, we assume, increased cache efficacy. Results for Hadoop are omitted to allow us to present the graph at a finer resolution, thereby more readily exposing the variations between the different experiments. The *Map* and *Reduce* task completion graphs for SciHadoop are shown as points of reference. We omit the *Map* task completion graph for SIDR as Figure 6.9 shows that *Map* tasks for SIDR and SciHadoop display the same completion time graph regardless of the number of *Reduce* tasks.

While our modified version of Hadoop must still hold to the requirement that a given *Reduce* task cannot begin processing its data until all of its data dependencies have been satisfied (Subsection 6.5.1), tasks in our system are meeting their individual sets of data dependencies on a rolling basis (rather than a single, global barrier). Given this situation, the ideal task completion 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. Figure 6.10 shows that as the number of *Reduce* tasks increases, the corresponding task completion graph line converges towards that ideal with 528 *Reduce* tasks being close to optimal.

SciHadoop's reliance on the global barrier precludes it from realizing, in any meaningful way, a performance improvement via an increased number of *Reduce* tasks. This is caused by the requirement that all *Reduce* tasks must wait for the barrier (in order to guarantee correctness), so increasing the total number of *Reduce* tasks past the amount that can run concurrently results in the same amount of work being split into smaller units that then must queue up. Figure 6.11 shows the *Reduce* task completion times for Query 3 when the number of *Reduce* tasks is scaled for SciHadoop with the results for the same query run via SIDR with 168 *Reduce* tasks included for comparison. We omit all *Map* task completion lines except one, since they do not vary appreciably between the different experiments.

The most significant result that we see when scaling the number of *Reduce* tasks under SciHadoop is that total query run-time is largely unaffected. By comparing this to SIDR with 168 *Reduce* tasks, it is readily evident that breaking the global barrier is key to profiting from an increased number of *Reduce* tasks for a given query.

It is worth noting that SciHadoop produces its first result more quickly with 168 *Reduce* tasks (as opposed to 22) because it is splitting the same amount of intermediate data up into smaller chunks, so that once the global barrier is satisfied, a given *Reduce* tasks has less to process. This still leaves the global barrier as a hard limit on starting *Reduce* tasks (and therefore total query run-time).

6.7.2 Query 4: Filter Function

We next apply the same techniques to a different class of queries and find similar results in that both the time to first result and total run-time are decreased with SIDR, as compared to SciHadoop. For this query, we created a data set of the same size, $\{7200, 360, 720, 50\}$ (Figure 6.7), with values generated from a normal distribution via the *java.util.Random* class. The query applies a filter to the data that emits only values more than three standard deviations greater than the mean value. Query 4 uses an extraction shape of $\{2, 40, 40, 10\}$ out of con-

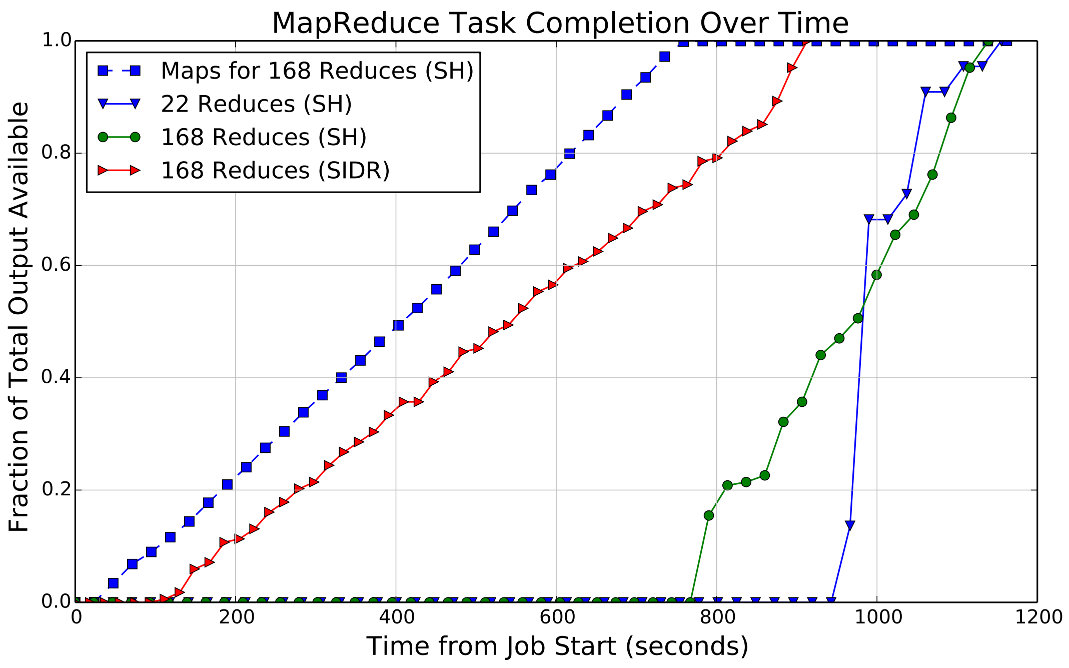


Figure 6.11: *Reduce* task completion for a median query run with SciHadoop (SH) using 22 and 168 *Reduce* tasks and SIDR running 168 *Reduce* tasks.

venience; results will contain a list of all the values in the extraction shape that are over the threshold. Since we're only returning values greater than three standard deviations, the total output is approximately 0.1% of the total data set (93.31 million values out of a 93.31 billion values data set).

Figure 6.12 shows the results for the fourth query when varying the number of *Reduce* tasks with SIDR and SciHadoop with 22 *Reduce* tasks (the SciHadoop *Map* task completion results for 22 *Reduce* tasks is show for reference). Again, Hadoop results are omitted in favor of showing more detail via a smaller x-axis range.

Query 4 outputs significantly less total data than Query 3. Because of this, the *Reduce* task completion lines converge towards optimal with fewer total *Reducer* tasks than Query 3 since each *Reduce* task has less data to process and therefore finishes more quickly, freeing up that server to process another *Reduce* task. Also, since the *Reduce* tasks represent such a small fraction of the total query execution time to start with (indicated by the slope of the *Reduce* task completion graph for SciHadoop with 22 *Reduce* tasks), there is little room for improvement in total execution time. This query is *Map* task intensive and the optimizations presented in this chapter are all focused on post-*Map* task aspects.

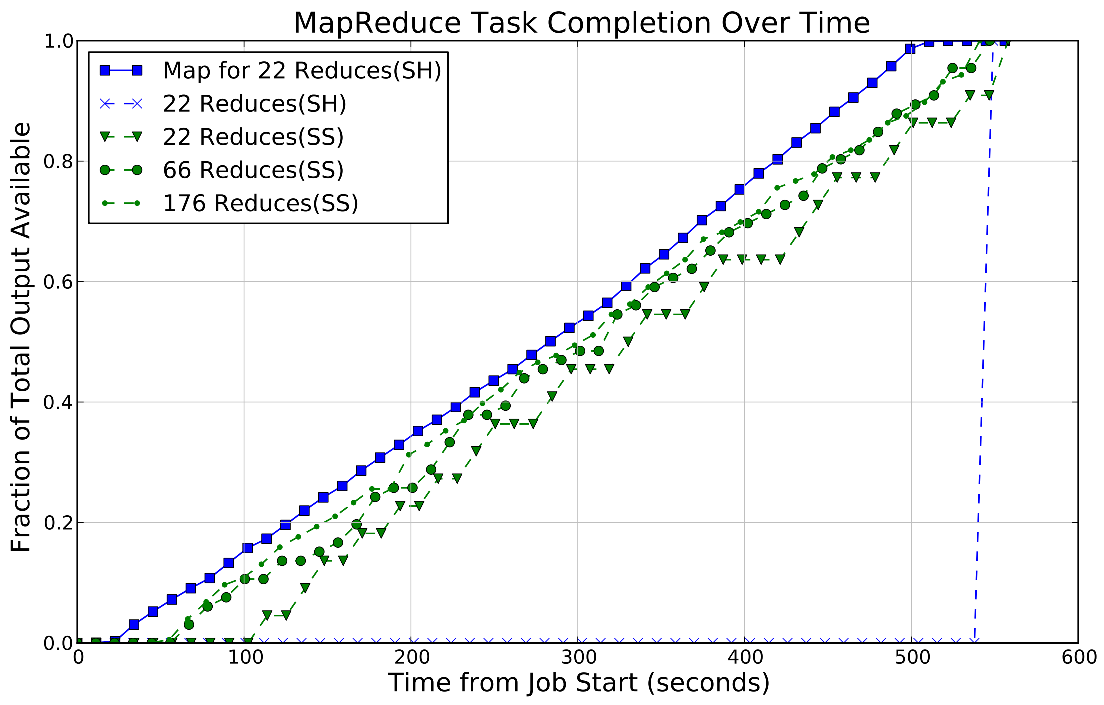


Figure 6.12: *Reduce* task completion for a filter query run with SciHadoop (SH) using 22 *Reduce* tasks and SIDR (SS) running 22, 66 and 176 *Reduce* tasks.

By contrasting the results of the two queries on data of the same size, it is evident that the nature of the query being applied has a significant impact on the number of *Reduce* tasks required to approach optimal performance. In general, the ideal number of *Reduce* tasks for a given query would be the minimum number required to achieve performance within some delta from the optimal line since each additional task carries with it a small fixed cost due to JVM startup and task initialization. At some point (again, query and dataset dependent), further increasing the number of *Reduce* tasks will begin to degrade performance because of this fixed, per-task cost.

6.7.3 Variance

Even with the Hadoop scheduler changes described in Section 6.5.3, there is still a lot of flexibility in how tasks are scheduled in our system. Figure 6.13 shows the amount of variance for *Map* tasks and *Reduce* tasks when 22 *Reduce* tasks are used as well as just *Reduce* task variance when 88 *Reduce* tasks are used. The graphed results are averages from 10 runs with error bars displaying the standard deviation for each data point. In SIDR, data dependencies are small(er) global barriers, so a *Reduce* task will have at least as much variance as the set of all *Map* tasks that it depends on. Increasing the number of *Reduce* tasks makes those sets of dependencies smaller (per *Reduce* task) and reduces the likelihood of a given *Reduce* task depending on several long-running *Map* tasks. This relationship is visible in Figure 6.13 where the experiment with 88 *Reduce* tasks shows significantly lower variances than the same query with 22 *Reduce* tasks but more than the *Map* tasks for the same experiment.

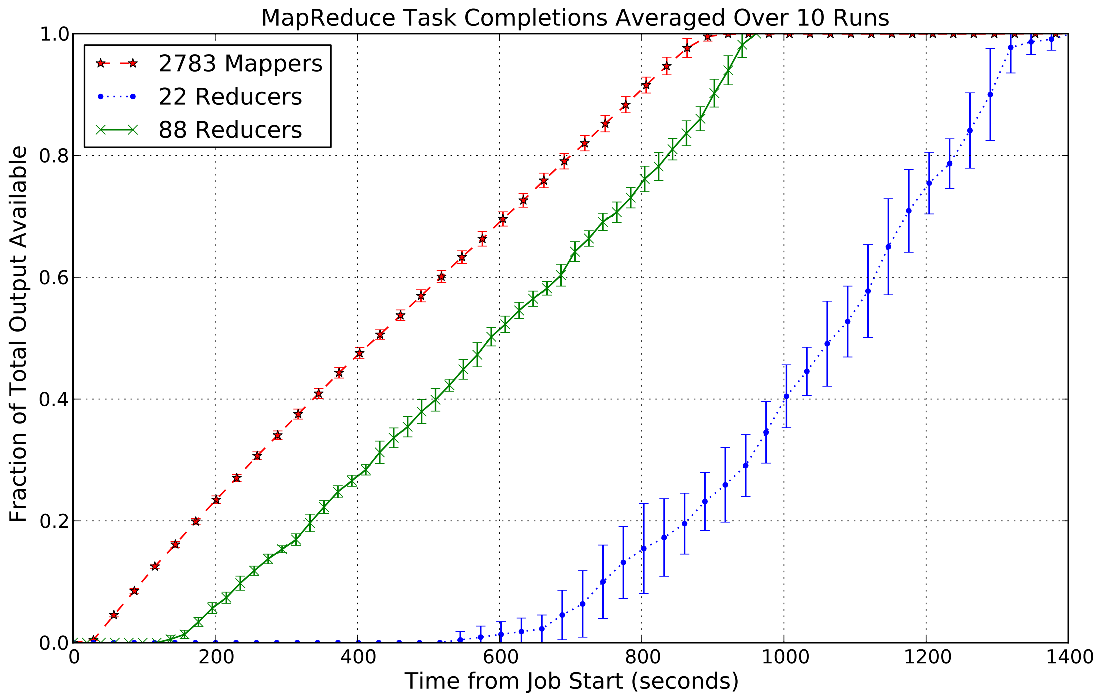


Figure 6.13: Variance of task completion, averaged over 10 experiments with standard deviation shown as error bars. SIDR *Map* task completion for 2783 tasks along with *Reduce* task completion for 22 and 88 *Reduce* tasks.

6.7.4 Contiguous Output

The hash+ function (Section 6.3) leverages query-specific knowledge to partition the output space into KEYBLOCKs that are both balanced in size and contain contiguous portions of the output space. This latter point is a significant improvement over the default when writing out structured data in a *MapReduce* program, assuming that writing logically contiguous data translates into contiguous file accesses.

The default partitioning approach taken by Hadoop results in each KEYBLOCK consisting of data residing throughout the total output space (Figure 6.6a). Writing a collection of data spread throughout a space (sparse data) is non-trivial. There are a few common approaches, one of which is to create a file representing the entire space and using sentinel values for the missing data points. If we consider a *MapReduce* program that took this approach, a few issues become apparent. Firstly, the size of the file written by each *Reduce* task is the same, namely the size of the total output, which means that increasing the number of *Reduce* tasks for a given query will increase the total amount of bytes written but not the amount of useful data. This creates a disincentive to increase the degree of parallelization of the query, which conflicts with the results in Section 6.5.2. Secondly, the time taken by a *Reduce* task to write its data will increase as the number of *Reducer*s increases, due to larger seeks between writes (caused by the default partitioning scheme using a modulo hash function). Table 6.1 shows the results of a micro-benchmark that simulates these issues by writing sparse data to a NetCDF file. For the experiment, we fixed the amount of data a single *Reduce* task would write at 24.8 MB and then scaled the total amount of output along with the number of configured *Reduce* tasks (i.e., between measurements we doubled the size of the total output and also doubled the number of simulated *Reduce* tasks). The table shows the size of the file created by one *Reduce* task and the time it took for the *Reduce* task to write its data (data generation and meta-data operations were not included in the timing). All time measurements are averages across 10 runs with standard deviation shown in parenthesis. As the table shows, even though the amount of data assigned to a given *Reduce* task is held constant, the total output written by each *Reduce* task increases (since space for the entire dataset is allocated by each *Reduce* task, even though they are each writing only a portion of it). The increasing run time is attributable to a combination of each *Reduce* task having to allocate a larger file and larger seeks between writes as the number of *Reduce* tasks is increased.

Hadoop Reduce Write Scaling		
Total Reduce Tasks	Avg Time in Seconds (Standard Deviation)	Output Size (MB)
20	6 (.6)	494
40	11.4 (.9)	988
80	24.2 (3.2)	1976
SIDR Reduce Task Write Scaling		
*	0.3 (.02)	24.8

Table 6.1: Scaling of write times and sizes for individual *Reduce* tasks

The bottom entry in the table shows the same test with a single *Reduce* task writing a contiguous portion of the output containing the same amount of useful output data as the other simulations. Since the single *Reduce* task will write the same amount of data, regardless of the number of *Reduce* tasks, the output file size and write time is constant across tests.

There are other methods for storing sparse data, such as creating a collection of coordinates and the corresponding data for each coordinate. This approach would create storage overhead for data written (since both the data and its coordinate are explicitly stored, rather than the coordinate being implicit, as is normally the case with scientific file formats) that would be a function of the total output size. This overhead would be a constant scalar, relative to the amount of useful data, and independent of the number of *Reduce* tasks. Alternatively, many file formats provide compression, which would, to some degree, ameliorate the impact of having many sentinel values. However, the creation of KEYBLOCKs of equal sizes and also consisting of contiguous data presents a superior solution to these alternatives.

6.7.5 Reducing Network Resource Usage

Hadoop Reduce Write Scaling		
Map / Reduce Count	Hadoop (# Connections)	SIDR (# Connections)
2783/22	61,226	2,820
2783/66	183,678	2,905
2783/132	367,356	3,031
2783/264	734,712	3,267
2783/528	1,469,424	3,760
2783/1056	2,938,848	5,106

Table 6.2: Network connection scaling

As mentioned in Section 6.1, every *Reduce* task will request an output file from every completed *Map* task, regardless of whether there is any actual output to transfer. Table 6.2 shows how the total number of network connections between *Map* and *Reduce* tasks scales as the number of *Reduce* tasks is increased for a fixed query. The efficiency of SIDR is the result of each *Reduce* task only contacting *Map* tasks that have produced data assigned to its KEYBLOCK and ignoring all other *Map* tasks. Compare this to Hadoop, where every *Reduce* task contacts every *Map* task. Additionally, there is a limit to the number of concurrent connections per *Reduce* task (10 by default in Hadoop 1.0), which can create an undesirable serialization of communication. The reduction in network connections required by SIDR diminishes the likelihood of this serialization occurring and represents a more efficient use of network resources.

Chapter 7

InMemory Shuffle

The motivating idea behind the work in this section is to eschew writing intermediate data to disk and rather persist it in memory, ideally for as short a time as possible, without impinging on the flexibility of the *MapReduce* model. Our hypothesis is that by transitioning from disk IO to memory accesses, we should see a reduction in the average execution time of *Map* tasks, since they'll spend less time writing their output (this is predicated on the assumption that writing to memory is faster than writing to disk). Additionally, we anticipate a reduction in the time taken to shuffle intermediate data, since the data will already be resident in memory when a *Reduce* task requests it, eliminating another disk IO.

In the *MapReduce* data flow (Section 3.2.1), *Map* tasks write their output to local storage in order to achieve two properties:

1. **asynchronous communication**. *MapReduce* and Hadoop make no attempt to coordinate the execution of *Map* tasks with *Reduce* tasks. As of Hadoop 1.0, there are two pools of resources allocated, one for each type of task, and each task type is effectively scheduled independently. *Map* tasks write their output to node-local storage and *Reduce* tasks request that data, as needed, at some later point in time.

2. **fault tolerance**. When a *Reduce* tasks fails, a replacement *Reduce* task is started by the central coordinator. This replacement task requests the data for its KEYBLOCK from every completed *Map* task (as well as those that subsequently complete). If *Map* tasks did not persist their output, then, in the event of a failed *Reduce* task, all previously completed *Map* tasks would need to be rerun to (re)produce their output for the replacement *Reduce* task.

The changes made to the communications model in Section 6 lead us to reevaluate the need for asynchronous communication as well as the impact of a *Reduce* task failure in our system. Our modified scheduler (Subsection 6.5.3) creates temporal locality between a given *Reduce* task and the *Map* tasks that it depends on, thereby increasing the probability that *Reduce* tasks will fetch their assigned data promptly after a given *Map* tasks completes. While this does not eliminate the need for asynchronous communication, it does provide a probabilistic bound on the window of time over which that communication will occur. Somewhat similarly, our ability to use *Reduce* task specific barriers (Section 6.4) limits the number of *Map* tasks that would need to be rerun in the event of a *Reduce* task failure, assuming that the intermediate data were not persisted. Based on these two observations, we experiment with altering Hadoop's shuffle phase to persist intermediate data exclusively in memory and only until every dependent *Reduce* task has successfully retrieved their assigned data. We refer to this work as SIDR-IM, since this is the same system detailed in the SIDR paper [16] with the addition of an **i**n-**m**emory shuffle mechanism (hence the "-IM" suffix).

7.1 Scheduling

For *Map* tasks that contribute to a single *Reduce* task, the existing scheduling changes suffice because that *Map* will, by virtue of our previous changes, not start until its sole dependent *Reduce* task is already running. For *Map* tasks that contribute to multiple *Reduce* tasks, we identified three possible approaches:

1. Do not schedule a *Map* task until all *Reduce* tasks that it will generate data for are running. This approach runs the risk of creating a shortage of *Map* tasks eligible to be run if the *Map* and *Reduce* task dependencies overlap antagonistically or, in the extreme, a deadlock is possible (similar to the Dining Philosophers problem [33]).

2. Alter the InputSplit generation algorithms to create splits such that all of the data in an individual split are in a single $\mathbb{K}_\ell^\mathcal{T}$, resulting in that *Map* task contributing to a single *Reduce* task and thereby precluding this scenario. Data dependencies between *Map* tasks and a *Reduce* task are a result of the structure of the data being processed, the extraction shape and the InputSplit generation process. It may be possible to take the lessons learned in creating the Holistic-aware Partitioning function (Section 5.3) and apply a similar approach to generating InputSplits such that each InputSplit produces data for fewer (one) *Reduce* tasks. With this approach, the InputSplit generation process would attempt to optimize towards two to three independent goals (e.g., data locality, minimizing *Map* / *Reduce* task dependencies and, optionally, combiner efficacy), which seems likely to result in mediocre results.

3. Schedule a *Map* task when at least one of the *Reduce* tasks that depend on it is running, then cache the resulting intermediate data until the other *Reduce* tasks that need it are scheduled and then request said data. The RAM used to cache this data will be occupied for an unbound (but likely relatively short) amount of time and the number of *Map* tasks in a query that need to cache intermediate data will depend on the combination of the input data set and the query being applied. This unpredictability means that this approach may cause *OutOfMemory* exceptions at run-time, that would result in a given task failing.

The in-memory shuffle mechanism in SIDR-IM is derived from the Hadoop Online Prototype [59] (HOP) code, forward ported to Hadoop 1.0. The HOP authors elected to use the third approach (albeit in a different context than the one that we have created). This choice seems reasonable to us as well, based on our observations of scientific queries appearing in our survey of related work. In essentially all of the scientific queries that we have encountered, *Map* tasks that would generate data for more than one *Reduce* task will do so for consecutive *Reduce* tasks. Combining this observation with the fact that *Reduce* tasks are scheduled in monotonically increasing order (Subsection 6.5.3), leads us to another case of temporal affinity that results in cached intermediate data being accessed by all interested *Reduce* tasks in prompt order and that memory then being freed.

7.2 Fault Tolerance

As mentioned previously in this section, if intermediate data are not persisted for the duration of the query, then a *Reduce* task failure necessitates rerunning *Map* tasks in order to regenerate the missing intermediate results. In Section 6.5.1, we described how our system can calculate I_ℓ in order to determine actual data dependencies for the purposes of starting *Reduce* tasks once those dependencies are met. For fault tolerance, this same mapping is cached during job initialization and then used to schedule already completed *Map* tasks for re-execution in the event of a *Reduce* task failure. This approach enables a *MapReduce* query to continue executing despite *Reduce* tasks failures and in the absence of persisted intermediate data.

7.3 Memory Management

One of the primary differences between our work in SIDR-IM and the approach presented in the HOP paper is the management of server memory while a query executes.

7.3.1 Memory Usage in HOP

The MapReduce Online [27] paper presents several approaches to efficient shuffling of intermediate data. The HOP authors implemented pipe-lining as a means to enable *online aggregation*. With pipe-lining enabled, *Map* task output is buffered and, once a threshold is reached, a combiner (if specified) is applied. The resulting output is directly transferred to the appropriate *Reduce* tasks with the added requirement that all *Reduce* tasks are running for the duration of the query. Intermediate data are integrated into the results as they arrive at the *Reduce* task, limiting this approach to supporting only distributive operators. While we do not support *online aggregation* in our work, pipe-lining is an interesting avenue for future work.

For non-pipe-lining execution (referred to as "blocking" [27]), a fixed-sized cache is used to store *Map* task output. Intermediate data is written to this cache if 1) the output is below a specified threshold and 2) there is sufficient memory remaining in the cache. If either of these requirements are not met, the data is written to the local filesystem on the node executing the *Map* task and then read again when a *Reduce* task requests it (exhibiting behavior identical to unmodified Hadoop). By using a fixed-size cache, HOP's approach creates predictable memory consumption, simplifying the specification of a cache size that can co-exist along side the configured number of concurrent tasks.

7.3.2 Memory Usage in SIDR-IM

In order to guarantee that intermediate data never resides on disk, and without placing undue restrictions on the InputSplit generation process, SIDR-IM persists all intermediate data in an in-memory cache and at no point evicts data to disk-based storage. This approach does present the potential for RAM to be exhausted on a given node and we chose to address this issue via the tuning of buffer sizes as well as limiting the number of tasks that may concurrently execute on a single node.

7.4 Implementation

As previously mentioned, SIDR-IM is based on the HOP source code [59]. In HOP, a memory manager class is created for each TaskTracker (one runs on each node that is used to execute *Map* and *Reduce* tasks). When a *Map* task completes, it registers its output with the memory manager on its node. Also, when a *Reduce* task starts, it registers itself with the memory manager on every node. In HOP, when a memory manager receives the output from a *Map* task, it sends that output to every previously registered *Reduce* task and also caches it for any *Reduce* tasks that may subsequently register. Likewise, when a *Reduce* task first registers, any existing *Map* task output is sent to it and an entry for that *Reduce* task is created so that any subsequent *Map* task output may be sent to it. This approach gaurantees that every *Reduce* task will receive its output from every *Map* task. HOP only discards intermediate data when the job completes, meaning that the amount of intermediate data consumed by these memory managers strictly increases as the query runs.

SIDR-IM leverages the knowledge of which *Reduce* tasks depend on data from which *Map* tasks to initiate transfers only where an actual data dependency exists, resulting in a dramatic reduction in total network connections (Table 6.2). Additionally, intermediate data is discarded once it has been successfully transferred to all dependent tasks. Ideally, this will achieve a relatively steady state, where intermediate data is being discarded at roughly the same rate as new intermediate data is being generated via *Map* tasks executing. The temporal locality induced by our changes to the Hadoop scheduler should contribute towards achieving this steady state.

7.5 Experiments, Part 1

These experiments focus on measuring the effects of using an exclusively in-memory shuffle in Hadoop. The hardware specs for the nodes used are listed in Appendix A and the query being executed is identical to Query 3 from Subsection 6.7.1.

7.5.1 Configuration

Given that SIDR-IM caches intermediate data in the case where a *Map* task generates output for more than one *Reduce* task (Subsection 7.1), we have to account for this added memory usage. Our research cluster (see Appendix A) consists of nodes that contain two dual-core CPUs and 8GB DDR-2 RAM. In previous experiments [16], we found that the ideal per-node configuration was to allocate 3 *Map* task slots, each with a 832 MB RAM limit, and 2 *Reduce* task slots, each with a 1344 MB RAM limit. For the experiments presented in this section, these limits were increased with the *Reduce* task memory limit increasing to 2560 MB and the *Map* task memory limit increasing to 1280 MB. The larger memory allocations were required for larger output buffers that we found were required to prevent spilling *Map* output to disk while also avoiding *OutOfMemory* errors. Concurrency limits were altered to be 4 *Map* tasks and 1 *Reduce* task.

7.5.2 Query 5

As a baseline, we ran the same query specified in Subsection 6.7.1 over the same 348 GB dataset so that we could compare our results to those in the previous section. Our nodes lack sufficient memory to run this query with the 22 *Reduce* tasks that would normally be used (recall the rule of thumb mentioned in Subsection 6.7.1); it is not possible to allocate enough memory for a *Reduce* task to hold all of the data that would be assigned to it. Via experimentation, we found that the smallest number of *Reduce* tasks that were required such that a given *Reduce* task could hold all of the data assigned to it was 168 (note, this is a dataset / query specific number).

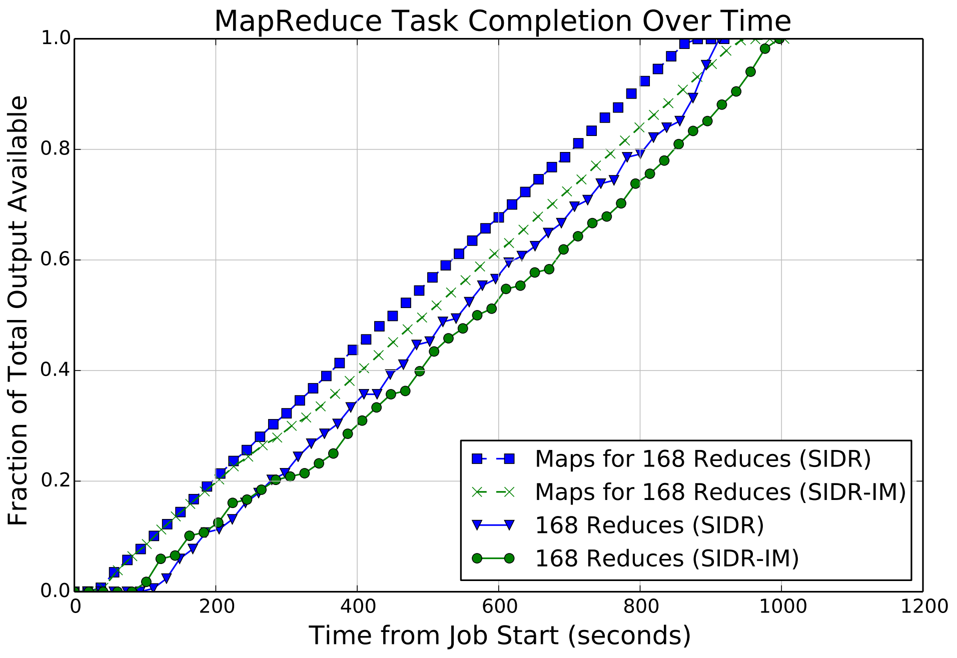


Figure 7.1: *Reduce* task completion for SIDR and SIDR-IM

Figure 7.1 shows the *Reduce* task completion graph for this query and it is clear that moving the shuffle phase to be exclusively in-memory did not improve total query execution time. In fact, it yielded slightly worse performance with the final result taking long to produce than with SIDR (although this may be attributable to SIDR-IM being more sensitive to tuning, since memory is more scarce).

After further research, presented in the following subsections of this chapter, we identified the reason that speeding up individual shuffles (Subsection 7.6) did not improve overall job performance. In unmodified Hadoop, *Map* task output is written to disk. On our cluster, we have configured more concurrent tasks executing per node than a given server has physical cores to execute. This over-subscription means that when a *Map* task goes to write its output to disk (not computationally dense), there are other tasks that can soak up the unused cycles. In effect, writing *Map* task output is not in the "critical path" as any inefficiencies in this process are hidden by the oversubscription on a given server. In SIDR-IM, the decision to write *Map* task output to memory moves this process into the critical path, since it is now entirely dependent on CPU and memory resources, for which there is heavy contention. This is why, even though the shuffles themselves are more efficient in SIDR-IM, the total job time does not change; we took a process that was not in the critical path and, while optimizing it, moved it into the critical path. In light of this discovery, we revisit previous research in Section 7.7 and find a means of reducing job run-time while persisting intermediate data in-memory.

7.5.3 Improvements in Shuffle Efficiency

While Figure 7.1 does not show an improvement in job run-time, SIDR-IM does yield improvements in the times taken for individual shuffles to occur. Table 7.1 shows the impact that our work has on both the time taken to shuffle intermediate data, the number of connections over which that data is shuffled, and the total time taken to execute a *MapReduce* job. The next subsection discusses these entries and provides further quantitative comparisons of shuffle performance under differing approaches.

Hadoop Shuffle Scaling			
Configuration (# Reducers)	Total Shuffle Time (Aggregate Seconds)	Total Shuffle Connections (# Connections)	Total Job Time (Wall Clock Seconds)
SciHadoop 22	19,280	61,182	1,173
SIDR 22	6,657	2,851	1,324
SIDR 168	11,161	3,337	1,141
InMemory 168	2,273	3,337	1,027
SIDR 528	6,885	4,510	920
InMemory 528	2,413	4,510	1,223

Table 7.1: Hadoop shuffle scaling

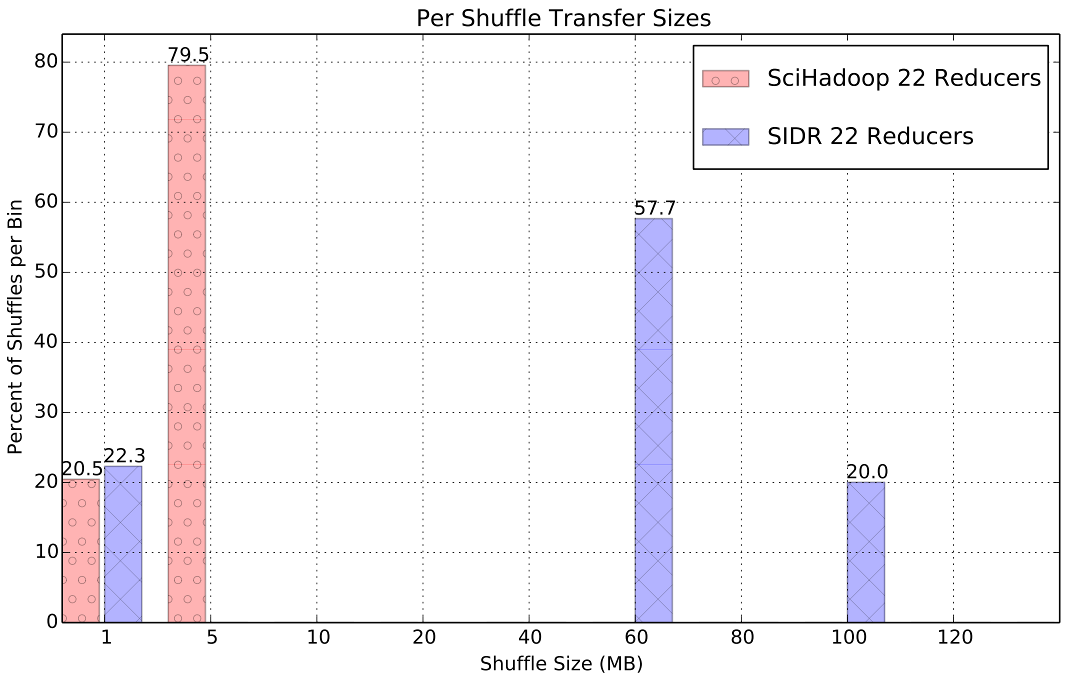


Figure 7.2: A histogram of per-shuffle transfer sizes for a Hadoop query run with SciHadoop and SIDR, both configured with 22 *Reduce* tasks

7.6 A Deeper Discussion of Shuffle Performance

While SIDR-IM's in-memory shuffle phase did not have an appreciable impact on total job execution time, the changes in the performance and characteristics of the shuffled data are interesting. In order to establish a baseline, we compare the characteristics of data shuffled with our previous projects (SciHadoop and SIDR). The hash+ function dictates which intermediate data is assigned to each *Reduce* task. A side-effect of the work done to reduce the per-task barriers (Section 6.4) is that there are fewer total *Map* to *Reduce* data dependencies (Table 6.2) but the total amount of data that must be shuffled is unchanged. Therefore, each of the shuffles that does occur must move a larger amount of data, a fact reflected in Figure 7.2.

The default hash function in Hadoop uses a modulo-based approach, resulting in all *Map* tasks generating approximately the same amount of output data for each *Reduce* task (with the exception of entries in the smallest bin, which are an artifact of how the keys are serialized; sometimes a *Map* task produces a very small amount of output for a *Reduce* task). The bin that all of the non-trivially small SciHadoop intermediate dataset sizes fall into spans the range of 1 MB - 5 MB. In contrast, hash+ constrains the number of *Reduce* tasks that each *Map* task will generate data for. This results in a bi-modal distribution where (non-trivial) shuffled data sizes are split across two buckets: the 60 - 80 MB range and 100-120 range. The difference between shuffles that fall in each of these buckets is a direct result of the degree to which the specified operator was able to be applied in the *Map* task (i.e., an increased rate of application yields smaller output *Map* task output sizes). Unfortunately, these are also the set of intermediate outputs that contribute to more than one *Reduce* task (hence the approximately 60 MB step-sizes visible in the memory pressure figures (Figures 7.10 and 7.11).

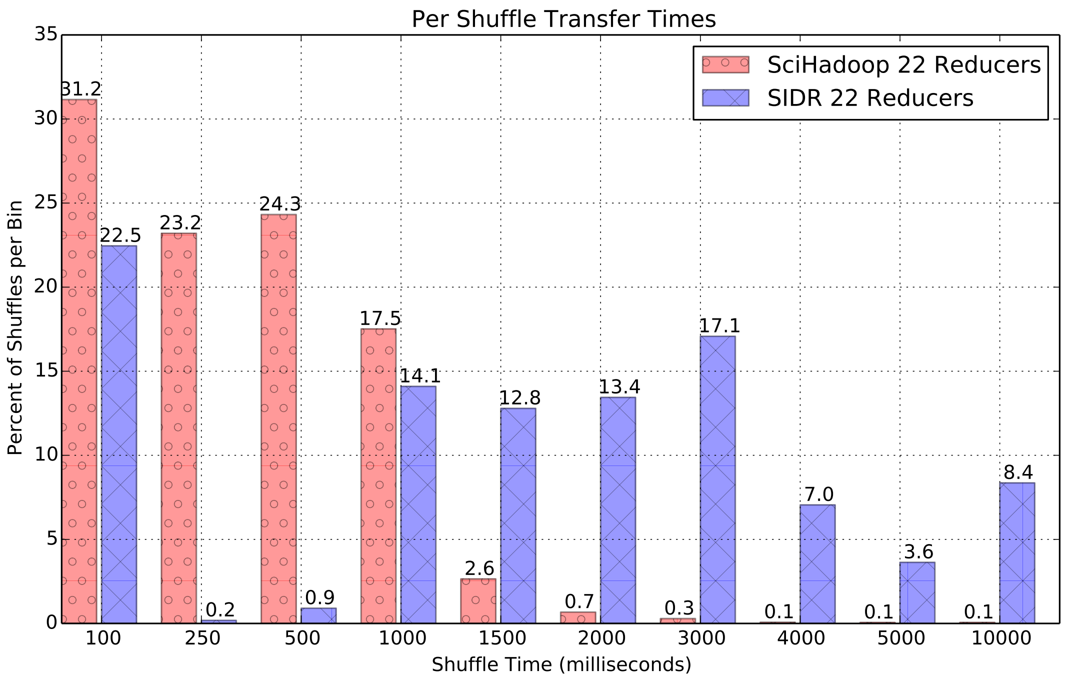


Figure 7.3: A histogram of per-Shuffle transfer times for a Hadoop query run with SciHadoop and SIDR, both configured with 22 *Reduce* tasks

The larger sizes of the SIDR output require a longer duration to transfer, compared to individual *Map* outputs generated by SciHadoop. Figure 7.3 is a histogram of the time required to shuffle the individual intermediate outputs whose sizes are graphed in Figure 7.2. While the per-intermediate output time is longer, the aggregate amount of time taken to transfer all intermediate outputs drops with SIDR, from 19,280 seconds to 6,657 (Table 7.1). We attribute this reduction in shuffle time to the reduction in the total number of intermediate datasets that translates into fewer unique datasets per *Map* task, resulting in a given *Map* task having to service fewer concurrent requests by *Reduce* tasks. That reduction in unique requests, in turn, results in fewer parallel reads being required of the single hard drive that services intermediate data reads and writes as well as the initialization of fewer connections, with their associated overheads. With SIDR, intermediate data requests result in large, consecutive reads issued to the hard drive, as opposed to the smaller reads from disjoint portions of the output file induced by the datasets generated under SciHadoop. We contend that it is this series of interactions that results in the reduction in total shuffle time. Our inference is predicated on the observation that mechanical hard drives exhibit better throughput in the face of large, continuous accesses than they do when presented with smaller accesses that require seeks to service. We would expect different performance results if the intermediate data was stored on some other medium, such as flash-based SSDs.

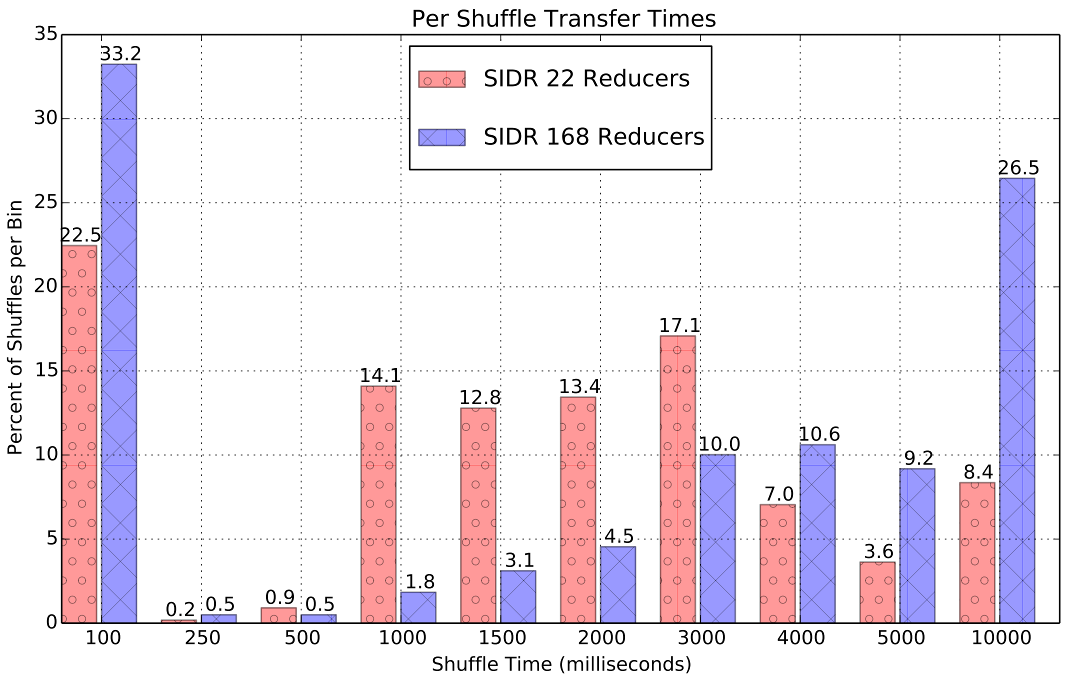


Figure 7.4: A histogram of per-Shuffle transfer times for a query run with SIDR using 22 and 168 *Reduce* tasks

Next, we look at the impact that scaling the number of SIDR *Reduce* tasks from 22 to 168 has on the shuffle metrics. The nodes in our cluster are configured to execute, at most, two *Reduce* tasks and 4 *Map* tasks concurrently. Given that the nodes possess 4 cores, this increase in total *Reduce* tasks will lead to 6 tasks (4 *Map* and 2 *Reduce*) running concurrently as opposed to 5 (4 *Map* and 1 *Reduce*) when only 22 *Reduce* tasks are configured. Figure 7.4 presents the histogram of the shuffle times for these two experiments. The added concurrent task increases competition for computational resources but also represents additional opportunity for overlapping computation with IO. The distribution of shuffle times shows an increase in the larger buckets, indicating a number of individual shuffles took longer under SIDR with 168 *Reduce* tasks than they did under SIDR with 22 *Reduce* tasks. This observation is reflected in Table 7.1, which shows that the aggregate shuffle time did increase. The notable exception is an increase in the number of intermediate data that are in the first bucket, representing the 0 - 100 ms range. This increase in shuffles with very short-duration time is caused by an increase in the number of *Map* outputs where the operator was very effective at reducing *Map* output size. Figure 7.5 shows the increase in the number of *Map* outputs whose total size is in the 0 - 1 MB bucket (from 56 under SIDR with 168 *Reduce* tasks to 1109 under SIDR with 168 *Reduce* tasks). Interestingly, the number of intermediate data that took a very short amount of time to shuffle (those falling in the <100 ms bucket) increased to a larger degree, from 618 to 1068, in the same tests. We attribute this larger than expected growth (short shuffle times increased by 450 shuffles while small dataset sizes increased by 278) to the increased number of total *Reduce* tasks resulting in an increased probability that a non-"small" intermediate data set was fortuitously generated on the same server running a *Reduce* task that depends on it. In this scenario, what would have been a network transfer instead results in a much faster in-memory copy.

While aggregate shuffle times increase, the total query execution time drops when increasing the number of *Reduce* tasks from 22 to 168 for SIDR, as shown in Table 7.1. This result indicates that the increased ability to overlap computation with IO more than compensates for the increase in aggregate shuffle time.

These comparisons between SciHadoop and SIDR's shuffle characteristics form the context in which we present the detailed experimental results for SIDR-IM. The choice of 168 *Reduce* tasks for the second SIDR experiment was dictated by the observation that 168 *Reduce* tasks is the minimum number required for successful execution of a task with SIDR-IM. The need for a larger number of *Reduce* tasks derives from the fact that increasing the number of *Reduce* tasks results in each individual *Reduce* task processing less data and therefore being dependent on fewer *Map* task outputs. This enables individual *Reduce* tasks to exhibit less variability in the time it takes them to request all of their intermediate data and less data being cached per *Reduce* task. The increase in predictability translates into a lower maximum in terms of memory needed to cache intermediate data at any point in the execution of a job and depending on fewer *Map* tasks results in each *Reduce* task requiring less RAM for caching intermediate data.

While the value of 168 *Reduce* tasks as the minimum for SIDR-IM was determined

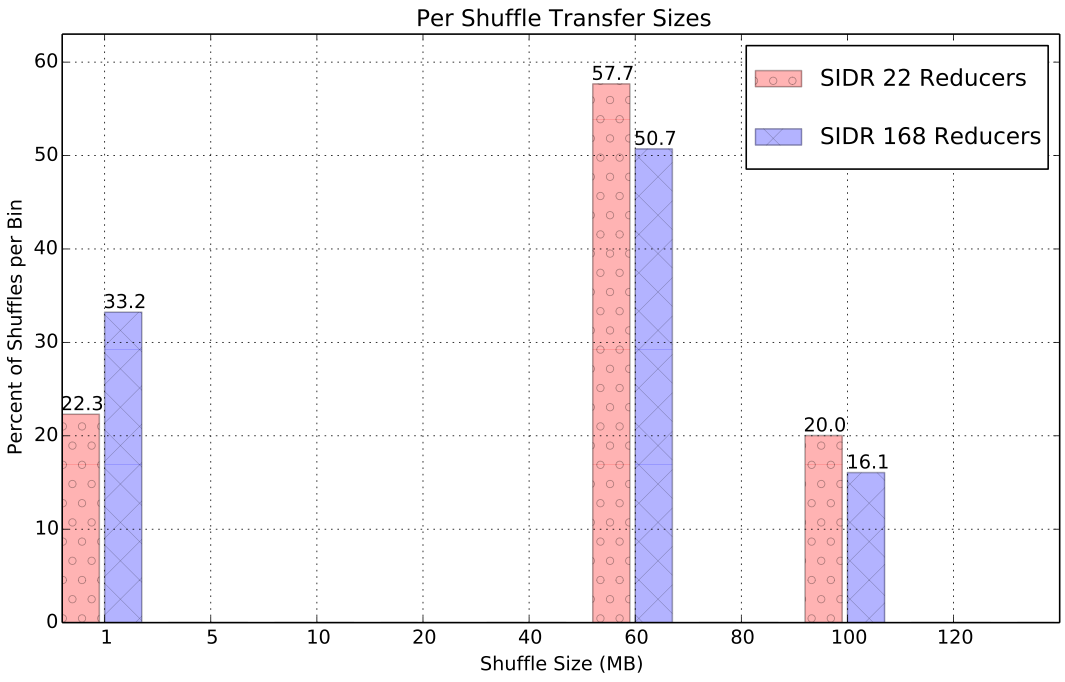


Figure 7.5: A histogram of per-Shuffle transfer sizes for SIDR with 22 and 168 *Reduce* tasks.

experimentally, we anticipate that it would be possible to come up with a fairly simple heuristic to compute this value based on the specific query, input data size and node memory capacity.

Figure 7.6 is another histogram of per-shuffle transfer times that contrasts SIDR with SIDR-IM (both with 168 *Reduce* tasks). The most striking result is that, on the whole, individual shuffles times improve significantly. All SIDR-IM bins under 1500 ms show increases and those over 1500 ms all decrease with almost no shuffles taking longer than 3 seconds.

A comparison of the shuffle sizes between SIDR and SIDR-IM is omitted because they are identical. The processes that dictate the sizes of intermediate data are unchanged between these two approaches and only the storage medium differs (disk-based vs purely in-memory).

Increasing the number of *Reduce* tasks for SIDR-IM (Figure 7.6) has a similar effect to that observed when scaling the number of *Reduce* tasks under SIDR (Figure 7.4) in that shuffle times increased, both in terms of the number of shuffles landing in the longer duration bins and in the aggregate shuffle time for the entire job. The degree to which shuffle times increased was less: scaling SIDR from 22 to 168 *Reduce* tasks saw aggregate shuffle times increase from 6,657 seconds to 11,161 seconds while scaling SIDR-IM from 168 to 528 *Reduce* tasks saw aggregate shuffle times increase from 2,273 seconds to 2,413 (Table 7.1). In contrast to the SIDR scaling experiments, total query execution time does not improve when we scale *Reduce*

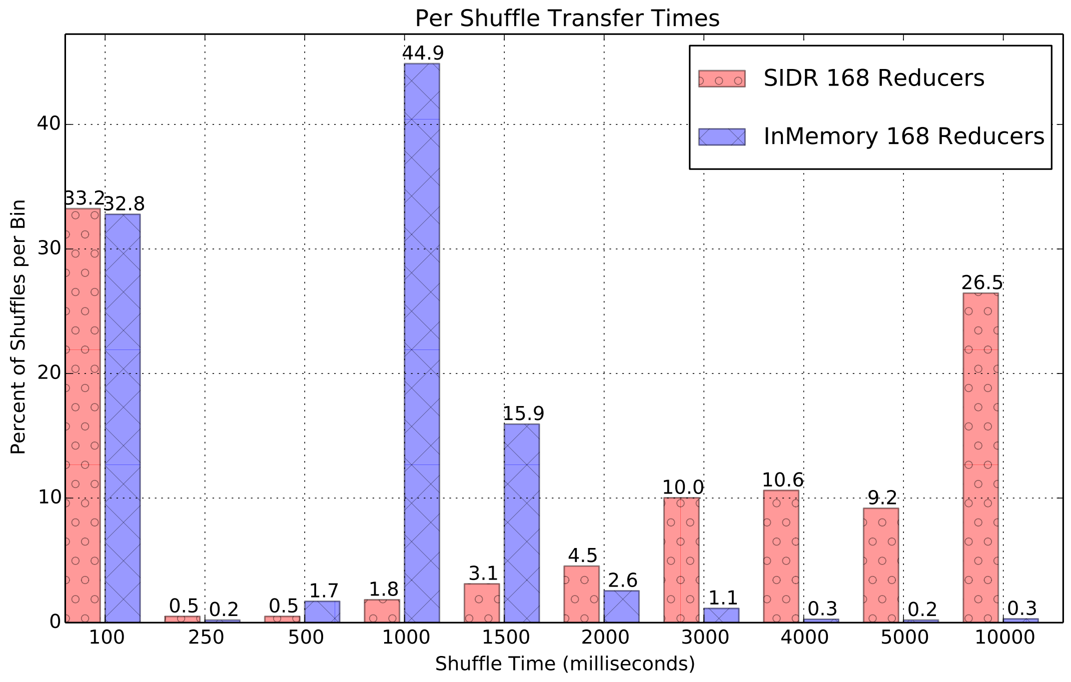


Figure 7.6: A histogram of per-Shuffle transfer times for SIDR and SIDR-IM, both with 168 *Reduce* tasks.

tasks for SIDR-IM. Total query execution time actually increases, from 1,027 for SIDR-IM with 168 *Reduce* tasks to 1,223 seconds for SIDR-IM with 528 *Reduce* tasks. Since SIDR-IM's shuffle is CPU and memory intensive, rather than disk IO intensive, increasing the number of tasks has no impact on Hadoop's ability to overlap IO with computation. Rather, it just splits the same amount of work into smaller units that compete in (essentially) the same manner for the same resources that their counterparts do when fewer *Reduce* tasks are specified. Furthermore, increasing the number of *Reduce* tasks incurs the additional overhead of spinning up (and then down) additional Java processes.

7.7 Query Aware Input Split Generation

Based on our findings from the end of Section 7.5, we revisited our Holistic-Aware partitioning approach from Section 5.3. This optimization alters the InputSplit generation process to favor the creation of splits that increase operator efficacy at the expense of diminished data locality. This approach yielded modest gains in earlier experiments, see Section 5.4 for details. Our improved understanding of the shuffle phase, developed as we investigated the (lack

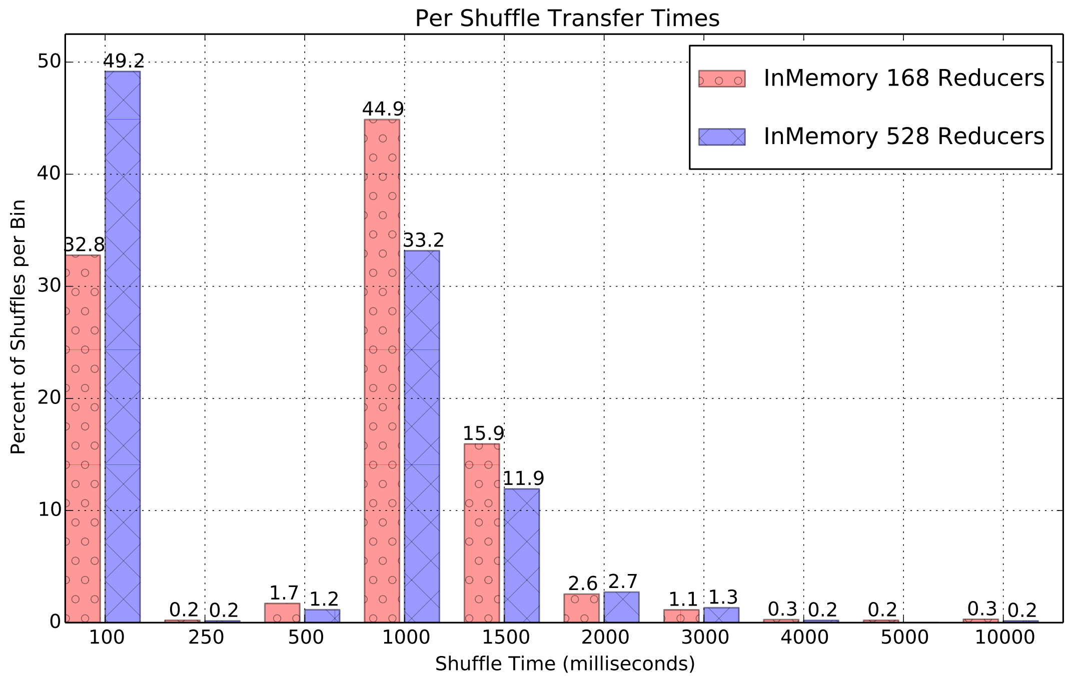


Figure 7.7: A histogram of per-Shuffle transfer times for SIDR-IM with 168 and 528 *Reduce* tasks.

of) performance gains for an in-memory shuffle, lead us to the realization that the relatively small degree to which job run-time benefited from this alignment of InputSplits is due to the fact that inefficiencies in the writing of *Map* task output can be obscured by the over-subscription of server CPUs. Because the inefficiencies are not contributing to the total run-time in a meaningful manner, then the impact of optimizing them will likewise be muted.

In SIDR-IM, we moved the writing of *Map* output into the critical path. Therefore, any alterations to the performance of this process should now be fully visible. This lead us to revisit the Holistic-Aware partitioning work (which we now refer to as QueryAware). Figure 7.8 shows the same query from Section 7.5, run with QueryAware both off and on. In this experiment, we see that job run-time is improved significantly (nearly 40% in this case); much more than the 4% (Table 5.1) seen in earlier experiments for this same optimization.

7.7.1 Map Phase Performance Analysis

In light of our previously flawed assumption that simply caching intermediate data in memory would improve job run-time, we wanted to identify the precise reason that the

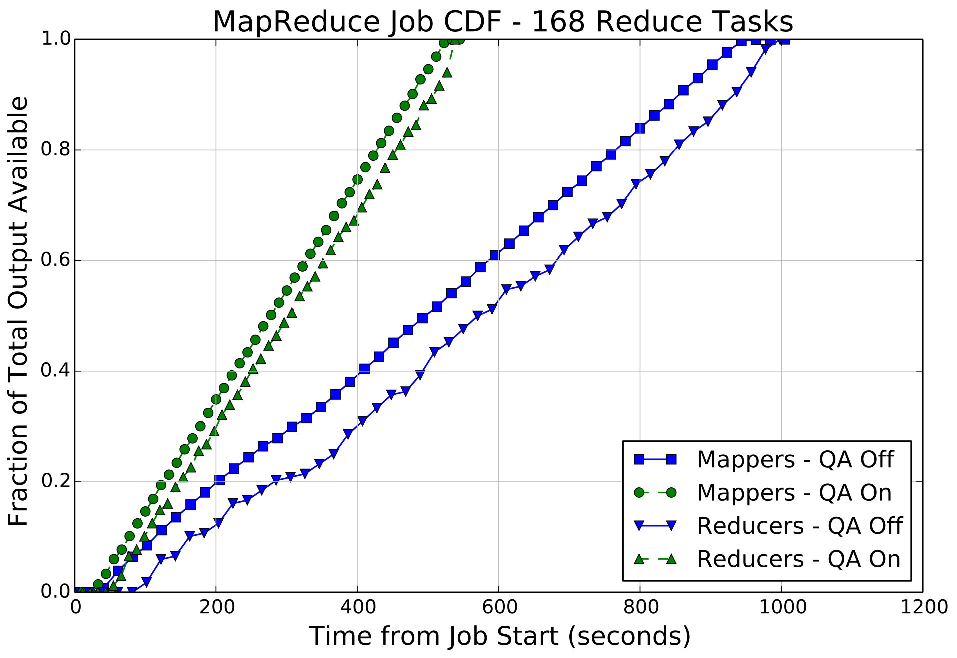


Figure 7.8: A comparison of *Map* and *Reduce* task completion lines for the same query run with QueryAware On and Off

combination of an in-memory shuffle with the QueryAware partitioning was yielding the runtime benefit visible in Figure 7.8. In order to accomplish this, we broke the *Map* task into four phases and then analyzed how long each phase took with and without these optimizations.

The four phases are:

- **Input IO**: time taken to read the data indicated by the InputSplit
- **Apply Map()**: time taken to apply the indicated map function to each input key/value pair in the data read during the previous phase
- **Register Output**: time taken to serialize each intermediate key/value pair
- **Commit**: time taken to track the serialized data as it is packed into buffers that will either be written to the local file system or buffered in memory

The highlight of Figure 7.9 is that the median time for an entire *Map* task to complete drops from a little over 24 seconds to just over 11 seconds when SIDR-IM and QueryAware partitioning are used in concert. These median values are represented by the red lines within the respective boxes. The top and bottoms of the boxes represent the 75th and 25th quantiles, respectively, and the "whiskers" that cap the dashed lines represent the 95th and 5th percentiles, respectively. The goal of this representation of the nearly 2900 *Map* tasks is to display the changes in not only the median observed *Map* execution times but also the alteration to their distributions and those of their constituent phases.

As we look at the per-phase breakdown, we see that the impact that using SIDR-IM with QueryAware partitioning had varies by phase. The "Input IO" phase improves somewhat (just over 600 ms) but we attribute this to an improvement in overall system efficiency leaving spare cycles that aid this phase. The same amount of total data is being read by the set of all *Map* tasks and the re-alignment of InputSplits should not have a noticeable impact on individual read times.

The performance of the "apply Map()" phase improves significantly as a result of the QueryAware partitioning generating InputSplits that lead to significantly higher operator efficacy. This increased efficacy is evident in the reduction in *Map* task output shown when comparing the second and third-to-last entries in Table 5.1. The speedup of the "apply Map()" phase stems from each *Map* task having significantly less data to serialize.

7.8 Intermediate Data Memory Pressure

As mentioned earlier in this section, caching intermediate data destined for non-running *Reduce* tasks increases memory requirements for a TaskTracker. Figure 7.10 graphs the memory consumed by intermediate data, on a single node, over the time span of a Hadoop job. The memory is allocated dynamically, as *Map* tasks complete, and is freed after the last *Reduce* task that requires that intermediate data successfully retrieves it. The same data dependencies used in to diminish the *Map* / *Reduce* barrier (Section 6.4) is cached and reused by the TaskTracker to determine which *Reduce* tasks will request output from which *Map* tasks. It is this aggressive approach to memory reclamation that enables the relatively steady state behavior shown in the figure. This TaskTracker stores a little over 18 GB of intermediate data over the duration of the query, which is more than twice the total memory our servers possess. SIDR-IM is able to cache this data with, at most, 350 MB of memory.

The maximum memory consumed by cached intermediate data at any given time is a probabilistic combination of the configuration of several components but the primary contributing factors are the number of concurrently executing *Reduce* tasks and the number of InputSplits that will contribute to more than one *Reduce* task. The configured number of shuffle threads contributes to a lesser degree as do tasks that are re-executing due to prior failures. Figure 7.11 shows the memory pressure exerted for the same query as Figure **??** run

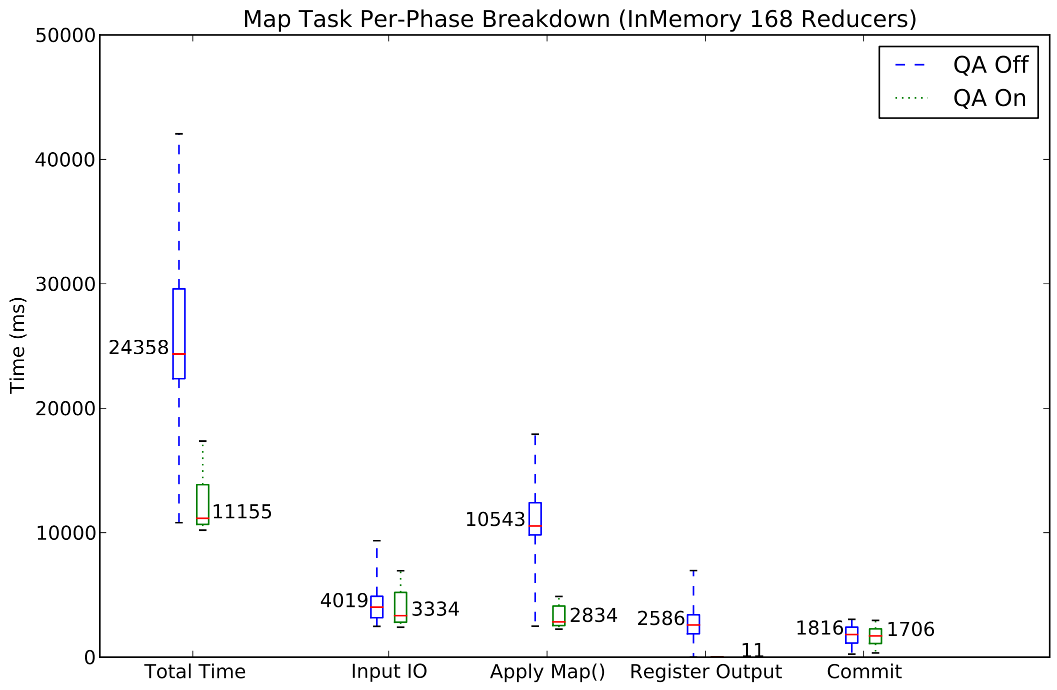


Figure 7.9: The total task run-time and per-phase breakdown of run-time for *Map* tasks. (The stated value and line is the median observed time, top and bottom of the box is 75th and 25th quantiles (respectively) and the top and bottom "whiskers" are the 95th and 5th percentile (respectively))

over the same data but with 528 *Reduce* tasks configured (note that each node will execute a maximum of 2 *Reduce* tasks concurrently). The maximum amount of memory used at any point in time is comparable to the immediately preceding figure, with the smaller peak resulting from those previously mentioned probabilistic interactions. Improved *Reduce* task performance reduces the duration that a given output remains in a cache, dropping the peak. For this query, dataset and cluster combination, the cached intermediate buffers are roughly 50 MB each, so this variance represents one fewer *Map* task output being cached at the peak compared to figure 7.10.

7.8.1 Alternate Approaches

HOP's approach to caching intermediate data has the benefit of consuming a predictable amount of RAM, where as SIDR-IM requires a variable amount that can, as mentioned, potentially result in run-time exceptions. One approach to correcting this unpredictability

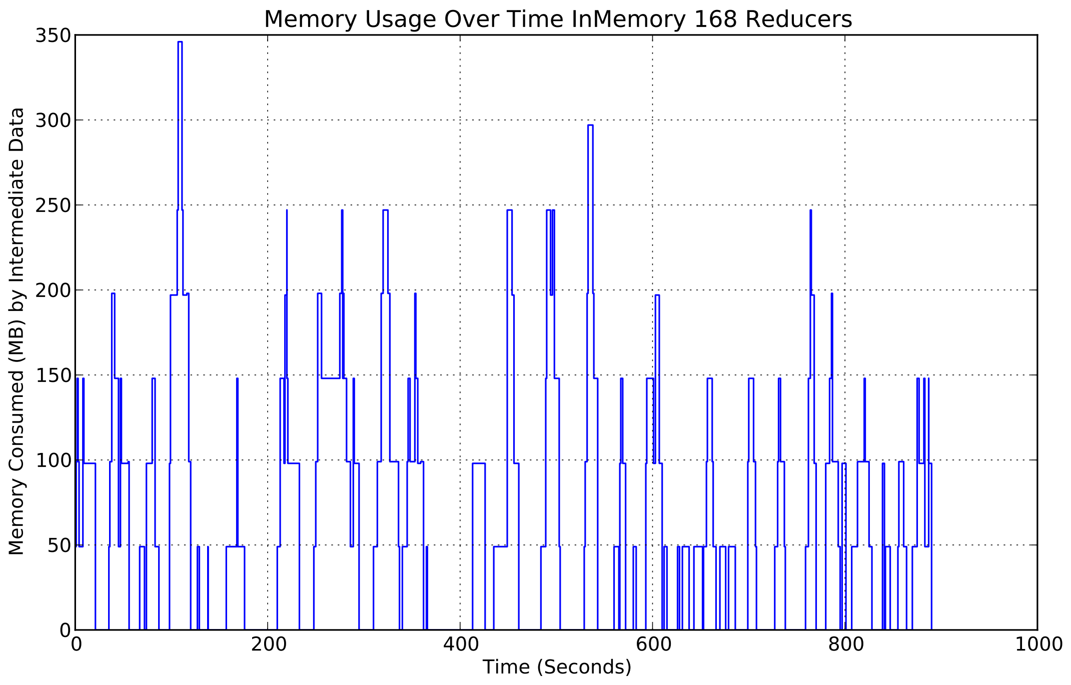


Figure 7.10: Amount of memory occupied by intermediate shuffle data, over time, for a single TaskTracker (SIDR-IM with 168 *Reduce* tasks).

would be to utilize a fixed-size cache and incorporate the allocation of the cache memory into the task scheduling process. We conjecture that it should be possible to fairly accurately estimate the amount of memory required to cache the output for a *Map* task (or at least calculate an upper bound with a reasonable amount of confidence) based on a combination of historical information for the same query and run-time observations. The scheduler could continue using its locality-based approach to scheduling *Map* tasks and add the caveat that the chosen TaskTracker have at least the estimated amount of necessary cache memory available, or else the scheduler would remove that TaskTracker from consideration and continue looking for a suitable host. We reiterate that only *Map* tasks that contribute to more than one *Reduce* task would require this extra constraint while *Map* tasks contributing to a single *Reduce* task can be scheduled via the existing approach since they will consume effectively no intermediate cache space (their output will be transferred to the cache and then immediately requested by the single already running dependent *Reduce* task). To further prevent the output of this class of *Map* tasks from consuming cache space in the TaskTracker, we could further alter SIDR-IM to have *Map* tasks directly transfer their output to the running *Reduce* task. The HOP authors took this approach in an early iteration of their work, but found that it had the undesirable

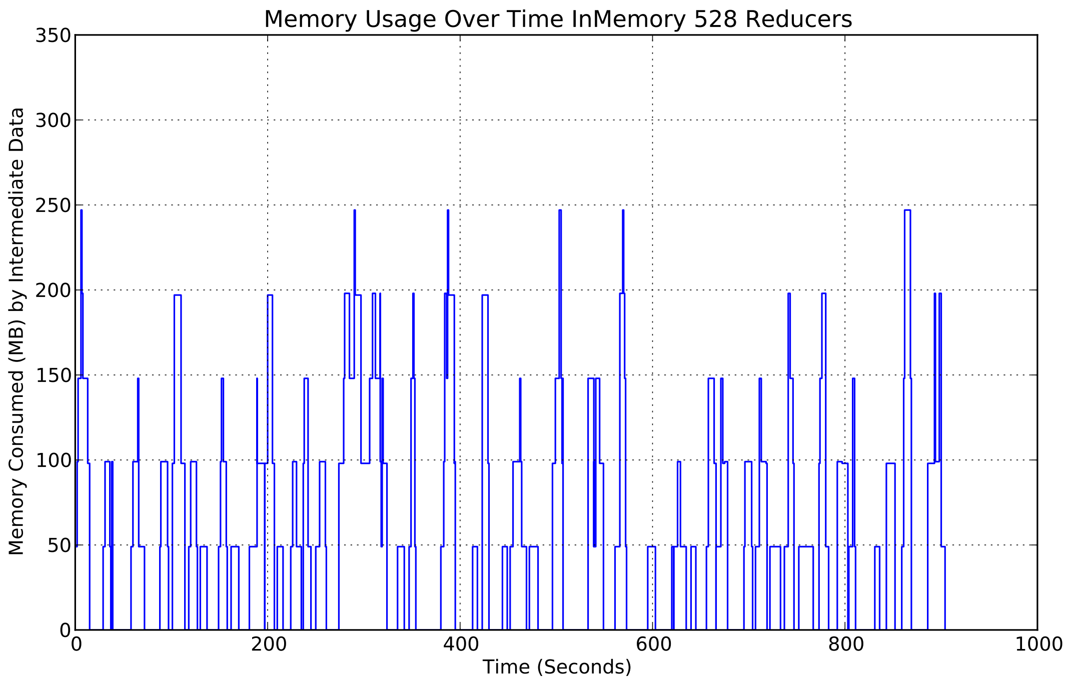


Figure 7.11: Amount of memory occupied by intermediate shuffle data, over time, for a single TaskTracker (SIDR-IM configured with 528 *Reduce* tasks).

effect of causing the given *Map* task to occupy its *Map* task slot for the duration of the transfer, causing individual task execution time to increase and driving down total cluster utilization. We share their view that this is a disadvantageous trade-off.

7.9 Further Observations

While persisting intermediate data in memory does not improve query execution time, we contend that this result makes a strong statement about the impact of our previous work; specifically, that per-*Reduce* task barriers enable us to profitably increase the number of *Reduce* tasks to the point where otherwise disk-based processes can remain memory resident in existing in-memory caches. What that means in the context of this experiment is that when we increased the number of *Reduce* tasks to the point that SIDR-IM could fit all of its intermediate data in memory, then SIDR could as well (since they are processing the same query that generates the same amount of intermediate data), albeit in caches. In both approaches, the *Reduce* tasks were doing in-memory merging and sorting, skipping the disk-based merge-sort needed when a *Reduce* tasks intermediate data exceeds its memory capacity. The only functional difference is that with SIDR-IM, *Map* tasks hand their output off to a memory manager rather than writing it out to local storage and that data is likewise not read from local storage when requested by a *Reduce* task. Furthermore, the temporal locality induced by our changes to the Hadoop scheduler in the SIDR work increases the likelihood that a *Map* task output in SIDR will still be resident in the node's filesystem cache when it is requested by a *Reduce* task, thereby providing memory-level latencies while also providing fault tolerance.

Chapter 8

Future Work

This section describes work that, while related to what we present, was deemed beyond the scope of this thesis.

8.1 Pipe-lining

The idea of pipe-lining *MapReduce* jobs, with the output of one serving as the input to the next, is a common concept that has been researched, both explicitly [70] and as part of larger projects [27, 156]. Oozie [95], an Apache incubator project, is a workflow scheduler for Hadoop jobs that makes pipe-lining straight-forward. Most "shared-nothing" systems can be used as part of larger workflows that use some degree of pipe-lining to accomplish more complicated actions than would be possible with a single query / program.

8.1.1 How Early Results Benefits Pipe-lining

Previous work on producing early results during the execution of a *MapReduce* program [28] noted that the goals of early results and pipe-lining are complementary. By sending early results to the next phase of a pipe-lined *MapReduce* program, it is possible to achieve additional overlapping of computation and IO as well as potentially receiving early results from the next *MapReduce* program.

The type of early results that our work produces, correct results for subsets of the total input, is a better fit for pipe-lining than the alternate form of early results, probabilistic results for the entire output based on some fraction of input processed, since the latter requires that the *MapReduce* program consuming the pipe-lined data be rerun as more accurate versions of the output are produced.

8.1.2 Anticipated Research Questions

We hypothesize that rerunning the pipe-lining experiments from MapReduce Online [27] with our version of early results would result in lower resource utilization (since subsequent *MapReduce* jobs would not need to re-execute as more output is sent to it). While this may or may not decrease the total query run-time, we would expect it to benefit other queries running on the same cluster, since more resources would be available to them.

8.2 Leveraging a Parallel Filesystem

The filesystem that ships with Hadoop, HDFS, supports append-only write semantics. The scientific file formats that we have used in our experiments and development (NetCDF and HDF5) update meta-data in their files as those files are written, even if the data itself is written in a sequential, append-only manner. Because of this incompatibility, our work to date has been unable to write out results in the same scientific file format that the input to the *MapReduce* program was read from.

Given the limitations of HDFS, we experimented with running Hadoop on top of the Ceph file system. Ceph is a general purpose parallel, distributed file system that supports mutable data. While not included in any published results, we have created an HDF5 Virtual File Driver (VFD) for Ceph. This allows us to conduct experiments that both read and write HDF5 files from a distributed file system (Ceph). In addition to enabling experiments that more closely resemble real-world use cases, we can now consider alterations to Hadoop that are not possible with HDFS.

8.2.1 Anticipated Research Questions

The addition of the extraction shape to our query language enables code in the *Map* task and *Combiner* function to determine if they possess all of the elements that will exist for a given instance of an extraction shape. This same knowledge that enables the holistic combiner can be leveraged to allow *Map* or *Combiner* tasks to write to the final output files if they determine that all the data that will contribute to the result for a given cell in the output file is present. Data written in this manner would bypass the *Reduce* task, thereby generating less intermediate data and IO which we conjecture would lead to a reduction in total execution time.

Writing to the final output from potentially many tasks, as opposed to just one task (the *Reduce* task assigned the KEYBLOCK for that file) will place different requirements on the distributed file system that stores persistent data. Lock contention and scientific access libraries' internal caching are of particular interest when considering this change. We envision a set of experiments to show how the ratio of final output written from *Map* and *Combiner* tasks affects the performance of *MapReduce* jobs. This effort will also require changes to Hadoop's process for committing output to HDFS, the code for rescheduling failed tasks and Hadoop's process for notifying *Reduce* tasks of completed *Map* tasks.

An issue resulting from moving to Ceph as the underlying distributed file system for our experiments is that the data placement function used by Ceph, CRUSH [150], does not necessarily store data on the node that writes it, as HDFS does, but rather calculates the correct nodes that will store said data. This functionality can be used to co-locate *Reduce* tasks on nodes that will store a copy of the output file those tasks will write, turning what would normally be writes to a remote server into local IO, for one of the replicas. A set of tests quantifying the effects of non-local *Reduce* task writes along with the impact of attempting to co-locate *Reduce* tasks on nodes that will store the data they produce would ideally be carried out as part of this effort.

8.3 Structuring the Unstructured

All of the work discussed up to this point has focused on approaches for processing structured scientific data. We hypothesize that it may be possible to apply similar approaches to unstructured data, with the general approach of creating artificial structure over the data. Specifically, we propose partitioning the data into separate sub-datasets, processing each of those sub-datasets independently and then integrating the results of those sub-datasets into the final result (likely via a *Map-Reduce-Merge* [156] type system).

8.3.1 Anticipated Research Questions

Our hypothesis is that the performance benefits gained from enabling *Reduce* tasks to start early as well as the anticipated benefits of the above proposed pipe-lining research will overcome the cost of adding an additional merge phase after the standard *Map* and *Reduce* phases. This hypothesis would require experiments to quantify the effects on query run-time induced by this work.

Given that we can not compute the data dependencies for unstructured data, we would have to resort to producing probabilistic early results for the entire dataset. However, our probabilistic results would be produced with knowledge as to which portion (in terms of byte-ranges) of the total dataset has been processed up to that point. This added knowledge could be useful for scientists attempting to understand the early results as they are produced.

This is a qualitative benefit that we would not plan to validate via experimentation.

Since we propose a partitioning of the dataset into sub-datasets, each dataset can be scheduled in any order. If a domain-specialist believes that interesting results are more likely to occur in a given region of the input, then that region can be scheduled first and when its results are available, the system can guarantee that those early results contain all the input in the specified sub-dataset. This is also a qualitative benefit whose value is not readily validated via experimentation.

Chapter 9

Related Work

The work presented in this thesis draws from several different research areas. We've split the related works into categories for ease of reference and comparison.

9.1 *MapReduce*

Phoenix [108, 157] is a *MapReduce* implementation that is optimized for multi-core computers, as opposed to distributed, networked systems, and several papers have built upon the original project. A follow-on project [131] added reservoir-based *Reduce* tasks to Phoenix. Some of the Phoenix-based work has looked into using MATE [68], which is derived from Phoenix, to process scientific data [147]. While it is vague in their paper [147], personal correspondence with the author confirmed that this system scheduled tasks at the file-level, requiring pre-processing of large datasets and the associated additional data movement and file management issues. We view this tactic as not viable at large scales, and therefore a major impediment to wide-spread adoption. Tile-MapReduce [24] is another attempt at optimizing *MapReduce* for multi-core systems. Its focus is extracting maximum performance on a single-host and it therefore does not contribute to the topics presented in this thesis.

Two closely related projects implemented server-side filtering and aggregation for data stored in NetCDF [128] and HDF5 [148] as alternatives to OPeNDAP [96]. This is comparable to issuing queries to a *MapReduce* cluster and then returning the results to a remote client.

Dryad [63] is a *MapReduce* competitor that has a more flexible communications model and can use a query compiler (DryadLINQ [159, 64]) that integrates information about the dataset, query and cluster into execution plan generation. Work on processing scientific data with Dryad [35] indicates that the parallel analysis enabled by the framework yields performance benefits. However, this research used input files that were all quite small (less than 33 MB) and therefore did not approach the issue of data locality or processing a subset of the data contained in a file. Their equivalent of the RecordReader simply uses a scientific access library to read an entire file and then passes that data, as a whole, to the application logic. This represents a fairly basic attempt at enabling the processing of scientific data and fails to utilize the structure present in the input.

Similar to Dryad, is Nephele / PACT [8] , which are a programming model and execution framework, respectively. The framework utilizes a three-stage process that incorporates a *Map*, *Reduce* and *Merge* phase (the *Merge* phase can come before or after the *Reduce* phase). The programming model supports contracts, where the operators applied in a given phase can inform the framework of certain properties that it will guarantee for the data, such as the output maintains the same keys as the input or that the output is partitioned in a certain manner.

Previous attempts at reducing the impact of the global barrier between *Map* and *Reduce* tasks in a *MapReduce* program serve as an interesting comparison point to our approach. In [141], the authors focused on distributive functions and required that all *Reduce* tasks be running prior to any *Map* tasks producing output. They then use reservoir-based versions of their functions to maintain a constantly updating result for each key. This approach, while interesting, has several drawbacks. The paper points out that this work requires a larger memory footprint, which may require resorting to a disk-based merge-sort. As dataset sizes increase, we see this becoming more of an issue. This approach is also limited to the types of functions it can apply (only distributive), and it is incompatible with our approach of increasing the number of *Reduce* tasks past the best practice of 90% of the nodes. Under this system, it is not possible to increase the total number of *Reduce* tasks, due to the requirement that they run throughout the duration of the query, thereby preventing the use of several of the optimizations that we have identified in our work.

Recent work on caching in Hadoop [5] recognizes that many large Hadoop clusters are multi-tenant and individual jobs typically cannot run all of their *Map* tasks at the same time. This results in "waves" of tasks running for a given job. The authors show that the total job only benefits when a waves-width worth of tasks were served out of the cache (with the optimal solution being that all tasks within a job read their input from the cache). An interesting application of this work to our research would be to adapt the PACMan system to apply their "all-or-nothing" model to the set of *Map* tasks that contribute to the same *Reduce* task. Since this set is almost always smaller than the set of all *Map* tasks for an entire job (and very possibly smaller than a "wave-width"), it may be possible to produce more granular scaling than was observed with PACMan in terms performance improvements relative to the amount of memory available for caching.

Work on extending *MapReduce* to produce early results [27, 28] has focused on online aggregation, a concept that originated in the database community [56]. In online aggregation, results are provided early in the query process and presented as a range with an associated probability. As the query continues to execute, the range of the answer shrinks and the probability increases, until the final answer is computed. In applying this approach to *MapReduce* [27, 28], the authors opted to output results at fixed points in the progression of the query (i.e., when 25%, 50%, 75% and 100% of the input had been processed by *Map* tasks). They did not attach a probability value to the results, letting the percent of the input processed serve as an indicator of the veracity of the early results. Early results of this type give an intuition as to the entire, final result of the query being run while our work on producing early results (Section 6.5.2) emits final results as they are produced.

CGL-MapReduce [36] used a modified version of *MapReduce* to process scientific data but depended on their content distribution network for communicating data between tasks and deferred failure recovery to its central coordinator. This work did not strive to achieve data locality and also did not leverage the structure inherent in the data being processed. Similarly, another research effort built a *MapReduce* implementation on Microsoft's Azure platform [49]. This second project also relied on the communications services unique to its environment (Azure) for communication between *Map* and *Reduce* tasks with resiliency provided by Azure's persistent queues service. Since this system was operating in a shared environment (cloud), data locality was not addressed. The nature of the data being processed was not leveraged but the research rather focused solely on using the parallelism provided by *MapReduce* to improve the performance of scientific analysis.

Sailfish [109] is a project that aims to reduce the impact of intermediate data skew in Hadoop. Their solution is to collect all intermediate data in rack-local files, evaluate the distribution of intermediate data by key and then assign intermediate data to *Reduce* tasks based on the observed intermediate key distribution. The Sailfish framework dynamically sets the number of *Reduce* tasks based on the size of the intermediate dataset. Their experiments displayed performance improvements for data exhibiting a significant amount of intermediate data skew. This approach introduces more serialization as *Reduce* tasks cannot start copying their data until after all the *Map* tasks have finished and the key distribution analysis has completed. Additionally, the failure of any of the rack-local nodes collecting intermediate data, while less likely than an arbitrary node-failure, necessitates rerunning all the *Map* tasks in the same rack that had written their output up to that point.

9.2 Databases

Databases have been used for processing large amounts of structured data for decades. However, the difficulty with which most databases scale horizontally, that is increasing the number of nodes running a database in parallel, was one of the motivations for the creation of *MapReduce*. The rise in prominence of *MapReduce* has inspired database manufacturers to revisit their designs and incorporate ideas, such as schema-less tables, from *MapReduce* into their systems as a means of enabling better scaling. At the same time, the shared nothing community has been integrating aspects of databases including indexing, views, and query optimizers [34, 77, 156].

A typical database use case is to ingest data from outside sources, reformat the data to match a schema, and then an experienced administrator configures indexes according to the anticipated workload for the data. These steps can lead to very efficient processing if the query is well aligned with the structure of the data. Queries that are not well-aligned to the structure of the data run less efficiently, with the worse-case resulting in the full data set being read in order for the query to be satisfied.

Scientific research often involves running ad hoc queries to suss out interesting aspects of the data being processed or queries that analyze data along different axes, thereby making those queries poor candidates for indexes. Additionally, the data used for scientific research can be unique or difficult to reacquire, for example sensor data or the results of an expensive simulation, so loading that data into a database either represents additional storage resource usage (if the original copy is kept in its original format for archival purposes) or else the scientist must trust the database to safeguard the fidelity of the dataset. Furthermore, if the data resides solely in the database, then existing tools are unable to access it, since they usually require a file interface and databases expose a query interface. All of these issues contribute to our view that a shared nothing style system interacting with binary file formats on top of a distributed file system is ideally suited for discovery science and providing efficient analytics for ad hoc queries.

Recent research delved into the unique characteristics of scientific data and how those properties can be leveraged to provide more efficient indexing of scientific data [98]. Another project [155] looked at building file indexes for scientific data and storing those in a distributed, in-memory hash ring in order to provide real-time support for efficient querying. The latter project suffers from the issue of requiring that the appropriate dimensions to index be known ahead of time and the utilization of in-memory storage for the indexes both bounds their size and raises fault-tolerance concerns.

Recent attempts to tackle the growing size of scientific data sets with databases include the Sloan Digital Sky Survey (SDSS) [129] and SciDB [127]. It is worth noting that the former project had the personal involvement of Jim Gray, one of the foremost authorities on databases, and focuses on the efficient execution of a set of pre-selected queries, rather than supporting efficient ad hoc queries. The latter is an on-going research project that represents the intersection of databases and array-based storage systems. SciDB's primary use case is the classic database model of data ingest and then enforcing its internal structure on that data. The project does aspire to eventually support queries over external data, but as their approach to doing so has not been published or implemented, we are unable to evaluate how it will serve the scientific community or compare to our work.

Other instances of databases being used for scientific data analysis include MonetDB/SQL [66] and a project that presents an array-based interface for querying data stored in a database [6] . A recent project [167] also extended MonetDB to treat arrays as first-class citizens. One of their key innovations is supporting structure-based operators that are very similar to our tiling approach that is defined via the extraction shape.

A 2005 paper [46] that discussed the applicability of databases, cloud and grid computing to scientific computations identified several issues inherent in using databases for processing scientific data. Among these issues were two that echo our concerns: a lack of set-oriented access to data and the inability to access the scientific data via existing tools.

9.2.1 Databases That Are Not Actually Databases

As previously discussed in Section 2, databases can provide very efficient query support by reorganizing data and creating indexes over that data. This requires *a priori* knowledge of the queries to be run and the data is typically only accessible via the query interface. Research into systems that fit this approach, reorganizing data for efficient query support, but that use an internal structure other than the row (or column) based format used by most modern databases has been conducted with the intent that these alternative representations would be more closely aligned to target problem domains and their data models. One of the first array-based data stores was RasDaMan[9], with ArrayStore [123] (and associated projects [121]) being a more recent implementation of this idea and this work [25] suggesting that those arrays be stored in files within HDFS. The latter looked at several approaches to decomposing and storing multi-dimensional data in terms of the effects on query performance. Similarly, SciDB [13, 126] and Pyramid [136] store their data as chunked, 2-dimensional arrays. In this way, they are database-like since the data is reformatted to the system's internal storage format and data model.

An early paper on an array-based query language [76] laid out a calculus of arrays and approaches to optimizing queries over arrays.

HAMA[118] is a BSP-based system designed for processing scientific data. It supports matrices and graphs as data models with data residing in a structured store (HBase in the literature). The work presented in this thesis could be used to support running HAMA over scientific data in their native file formats but that is beyond the scope of this thesis. All work presented in this thesis, both completed and proposed, is based on the *MapReduce* processing model.

Cumulon [60] is a framework for efficient execution of matrix computations within Hadoop that processes data stored in HDFS. An interesting aspect of this work is that the scheduler both assigns tasks to resources and allocates virtual machines on Amazon's EC2 service, based on time and cost constraints.

Some scientific computations are better modeled as graphs. While we have not addressed graph computations in our work, existing frameworks [83, 153] could likely be modified in a manner similar to the changes to Hadoop outlined in this work, thereby enabling those frameworks to access in-situ scientific data.

Research into distribution rules for queries over arrays [138] approaches the issue of presenting an array-based interface for scientific data analysis while storing the actual data in a classic relational database (MonetDB in this case). Similarly, RIOT [166] provides users of R with the ability to specify operations over arrays that are transparently stored in a traditional database. Yet another project [122] looked at methods for processing arrays in parallel within Hadoop (the data residing in HDFS).

SCOPE [22], Hive [133] and Tenzing [77] adopt SQL-like syntax and access their data as strongly typed columns (or rows) stored in files in a distributed storage system, Cosmos (part of Dryad [63]), HDFS and BigTable, respectively. It is worth noting that Tenzing can run over structured data, so it could be extended to read from scientific file formats by mapping multi-dimensional data to a table-based scheme. A solution of this variety would still need to address the issues of achieving data locality and mapping from the logical model to the byte-stream model described in this thesis.

HadoopDB[1] takes an interesting approach to mixing SQL with Hadoop: they run a database on each node and then use Hadoop for job scheduling and fault tolerance. Data are replicated among nodes just as it would be on HDFS, it simply resides in a database on each node. This approach has the data reorganization drawbacks already mentioned but does offer both good performance and scalability by merging aspects of two previously distinct approaches in a novel way.

Systems such as Piccolo [104] and Spark [161] exist at an intersection of simple, parallel frameworks and in-memory databases. They both provide a simple programming model with the framework taking responsibility for fault-tolerance, task scheduling and inter-task communications. Since they are memory-resident, they either fail to work or degrade to disk-based storage if the dataset being processed cannot readily fit in the collective memory of the cluster. In the latter case, they start to resemble *MapReduce* with the major exception that they assume a table model, rather than a file-model for interacting with storage. Either system could be extended to map multi-dimensional data to their table model and adapters for scientific file formats could be created, at which point they could be evaluated for their effectiveness at processing scientific data. This line of research is outside the scope of this thesis.

The Active Storage Fabrics Model [41] suggests using the main memory of a computing cluster as the storage medium for an in-memory database for the purpose of analyzing scientific data. While this would substantially improve performance over a disk-based analytics system, it also limits the input dataset size to the size of aggregate memory and has data movement costs.

Indexing within Hadoop has been shown to reduce total query times for subsequent queries [99, 3], assuming the correct dimensions to index are known ahead of time. Another project [146] combined the idea of content-addressable networks (CAN) with distributed, hierarchical R-trees. This allows queries to quickly (measured by the number of hops required) find desired data. This work generates many indexes on each node and then each node publishes the indexes that appear profitable given recently issued queries. The variability of which dimensions benefit queries over scientific data would likely render this approach less effective than was observed in the paper [146].

An index's ability to preclude data from needing to be read, as in [3], is very similar to our *NoScan* feature.

9.2.1.1 Resilient Distributed Datasets

One of the contributions of the Spark project is the idea of Resilient Distributed Datasets (RDDs) [160], which are immutable datasets whose keys are range-partitioned. Since RDDs are immutable and range partitioned, Spark can, given an operator (query), understand the relation between the input to an operator and that operator's output at the granularity of an RDD partition. This is fairly analogous to the process by which we compute $\mathcal{I}_\ell$ (Subsection 6.4). From the available literature [161, 38, 160], it appears that Spark categorizes dependencies between input and output RDDs as either one-to-one or one-to-all. We were unable to find an explanation of why they did this, but we assume that the authors found that most queries fall into one of these two behaviors, and those that do not were either so rare as to be inconsequential or that the impact on those queries was small enough to be tolerated. We consider Spark, and related works, to be excellent research projects in their own right(s), but we would have liked to see a further discussion on how RDD mappings are constructed and used in Spark. This would have enabled us to more intelligently contrast our work to theirs.

Given what is published about RDDs, we feel that our work on understanding how keys are mapped through different keyspaces in the *MapReduce* model is a more extensive treatment than those presented in the existing Spark [161], RDD [160] and Shark [38, 154] papers. Our intent in presenting formal definitions of how keys are translated between keyspaces as well as how our alterations to the *MapReduce* model affects these translations is to elucidate our lines of reasoning as well as the precise mechanics employed in our implementation.

The Spark platform has also served as the basis for a streaming computing platform [162]. In place of RDDs, this system used discretized streams as its unit of computation and resiliency.

9.3 MPI

Research into using MPI for processing large scientific datasets in parallel is underway at other institutions. One project created an HDF5 driver to write files to a distributed shared memory cluster that stored the data in an in-memory file system [124]. These "files" can then be analyzed very quickly with existing tools (ParaView in this case). This approach obviously has scalability limits (the aggregate amount of main memory) and is primarily targeted for analysis done immediately after a simulation executes, which is not the use case we are targeting.

Research into merging MPI with *MapReduce* has taken place with the goal being to run *MapReduce* on top of MPI [58] or, alternatively, using MPI within Hadoop [51]. Neither presents a compellingly novel approach given the previous systems already considered in the course of deciding to base our work on a shared nothing system.

9.4 Other Related Work

Other research groups have compared databases with *MapReduce*-based systems for analyzing large, scientific data sets [124, 102]. Some have focused on data from certain communities, such as Astrophysics [78], but they did not alter the underlying system, they simply used the stock implementation that we have previously described and benchmarked against.

The expressiveness of the *MapReduce* model [69] and how it compares to other models, such as bulk synchronous parallel (BSP) [40], and Dryad and Oracle [158], serve as good background material for understanding *MapReduce* itself.

One of the more interesting alternatives to *MapReduce* is Ceil [87]. It is a distributed data flow framework that allows for run-time, data dependent decisions in terms of data routing and barriers. While it is beyond the scope of this paper, it would be interesting to see how simple it would be to extend a Ceil-based system to incorporate our research into using scientific meta-data and query-awareness into the processing framework.

As Hadoop has matured, more concepts from the database community have found their way into *MapReduce*-based research projects. Indexing and co-partitioning (partitioning datasets so that related data from different tables reside near each other) were added to Hadoop as part of the Hadoop++ project [34]. Both features were implemented in such a way as to have minimal impact on query run-time (indexing is done incrementally as queries execute and co-partitioning occurred during data ingest, requiring *a priori* knowledge of which data would be queried at the same time). Building the indexes incrementally is attractive as it commits small amounts of work over time, resulting in often queried data realizing the benefits of an index without effort being wasted to build indexes on rarely accessed data. Co-partitioning during ingest is less attractive as we envision providing query support on top of primary storage (i.e., requiring no explicit ingest phase) but it may be possible to create co-partitions via a background process that is incorporated into the data placement or replication strategies. Somewhat similar to co-partitions, divergent replicas store redundant data in different arrangements with each being optimized for different access patterns.

While the issue of data locality for *Map* task input was well researched as part of this thesis, we do not plan to invest effort into improving *Reduce* task data locality. This topic has been investigated by previous research efforts [117, 70] in the context of shared compute clouds (virtual machines). In general, the goal is to schedule a *Reduce* task on a node that will execute at least one *Map* task that will contribute data to the identified *Reduce* task. This results in what would previously have been a network transfer of intermediate data turning into a node-local read. While this does represent a savings in resources, for large clusters and large datasets, the savings does not seem commensurate given the complexity added to task scheduling by correlating *Map* output with *Reduce* KEYBLOCK assignment and then scheduling those tasks to be co-located on the same node.

Recent research suggests that data locality is becoming less important as networking technology starts to catch up to disk-based storage throughput [4, 91]. While this may or may not hold true over the next several years, the paper points out that work on placing tasks where they can access their required data on local hard drives can be re-purposed to place tasks on nodes where the data is resident in the local memory (such as with RAMClouds [97]). The multiple levels at which data locality can yield performance benefits serves as further validation of the work presented in this thesis.

In discussing the issues inherent in using a database or a system that reorganizes data, we have claimed that access to the data in its original format is forfeited due to the reformatting. The work presented in the MRAP paper [116] (and a more recent, very similar paper [152]) can be used to overcome this issue; its indexing system could provide access to the data in its original format while internally storing it in some other layout that was better aligned to the anticipated query load. For example, pairing MRAP with the ArrayStore file format could be an interesting approach to providing improved query performance while still allowing access of the file in its binary format. The research required to quantify the trade-offs in doing so is outside the scope of this thesis.

Efforts to enable in-situ processing of data pre-date *MapReduce*, with the active disk [110, 2] work taking place in the mid-to-late 1990's. While a forefather of *MapReduce*, work in this area typically assumed that all required data was present on a given hard drive. Also, since the execution occurred on a processor attached to the hard drive, which is not the case in current systems, the programming environment was difficult to work within and there was little to no central coordination of processes.

Attempts at creating remote procedure calls (RPC) for active storage [120], published in 2002, represents preliminary work towards combining the concepts of active storage with distributed systems in a more centralized, coordinated manner. Another project in the active storage space is MVSS [80] (and, to a lesser extent, quFiles [140]), which provides in-situ processing of files by specifying views that, when read, transform the original data and return the derived content.

In addition to skew in per-key intermediate data sizes, another concern is computational skew, which is typically observed when *Map* tasks spend differing amounts of time processing datasets of equal sizes. Existing research attempts to create InputSplits representing equal computational requirements, rather than equal amounts of data, by leveraging a user-defined cost function [71]. This thesis does not include investigating this type of skew.

More recent works [72, 73] on addressing skew (both in size and computational load) looked at simply tracking the completion percentage of running tasks and then reassigning portions of long-running tasks input to idle tasks. This is a very general approach that could be applied to our work, but special care would be required to transfer contiguous ranges of intermediate keys when re-balancing intermediate data across *Reduce* tasks.

Integrating query support into a primary storage system results in processing tasks competing with existing storage services for resources (CPU, memory, etc.). While it is outside the scope of this thesis, some solution for providing isolation between these competing tasks will be required in a production in-situ processing system. Work on providing quality of service (QoS) guarantees for IO [103] serves as a good starting point for researching solutions to this issue.

A more tractable approach is to augment a file system to enable batch computing systems to issue hints or commands that inform the storage system of impending data accesses. BADFS [10] presented this idea along with supporting evidence that workload-informed decisions in the storage system yielded performance benefits. A more recent paper [137] builds on this, suggesting that data replication and layout be adjusted to support more efficient data accesses by batch computations.

In our discussion on pipe-lined *MapReduce* programs, we mentioned Oozie as one option for executing a pipe-lined query. It is also possible to express queries as a Pig Latin [94] program, which will generate a series of *MapReduce* programs to perform the specified work. Previous work into integrating scientific workflow systems with Hadoop include Kepler + Hadoop [145, 90]. Furthermore, multiple versions of Hadoop built specifically for queries that require multiple iterations of a given *MapReduce* query exist: Twister [37], HaLoop [14], and Iterative MapReduce for Azure Cloud [48]. These focus on lowering latency between *MapReduce* jobs, minimizing cross-job data movement and keeping data in memory between iterations. Another project [32] looked at modifying Hadoop to perform better in the face of dynamic (multi-stage with multiple communication patterns) scientific workflows.

Domain specific solutions, including [170], combine open-source components with an interface specific to a given topic, geoinformatics information systems (GIS) in this case. Similarly, [164] presents a SQL interface as a means to process GIS data with MPI and PnetCDF being used for the underlying data access. While this approach yields a solution that is custom-tailored to the data and queries common to that field, we would advocate building a general system and then layering a thin domain-specific interface on top of that.

Another domain-focused solution is MERRA [114], which is an entire computation and storage software stack built to serve climate data. It is a virtual machine image that connects to other installations over the internet (grid-enabled), uses iRODS [62] for data management and provides some data processing capabilities via supplied *MapReduce* queries. MERRA's storage layer is also similar to databases in that it extracts data from its native file format and restructures it (with the associated overhead and lack of access in its original format). While the *MapReduce* component in MERRA uses the standard Hadoop distribution, and therefore does not add to our discussion of extending *MapReduce*, the project serves as an example of how the work presented in this thesis could be utilized.

An early project that attempted to build a general, extensible data store that could support scientific data sought to use SQL as the general interface to the data and then delegated the storage and processing of that data to external, format-specific stores [12]. This work serves as a spiritual successor to ours, albeit with the significant difference that we chose *MapReduce* as the computational framework and model, as opposed to SQL.

The authors of a project enabling parallel particle tracing [100] presented some interesting solutions to dealing with iterative computations whose computational densities vary over time. This approach, which uses per-iteration statistics to adjust work distribution on subsequent iterations, could be integrated into a dynamic hash function for assigning intermediate data in an iterative *MapReduce* job.

The alterations to the Hadoop scheduler presented in this paper share the default scheduler's assumption of homogeneous per-node performance. Existing work in efficient scheduling of Hadoop jobs on heterogeneous clusters [165, 55] is equally applicable to our work.

Hadoop has been used as a means to parallelize existing serial analysis processes. As example, this work [61] utilized Hadoop as a means for fault-tolerant task execution with R [112] providing the actual data processing. Similarly, this work [163] uses Hadoop for task scheduling (and resource allocation) with actual data processing occurring via serial programs executed by a given task. This latter project incorporates HBase as a means to track data provenance and metadata.

Several projects [163, 66, 126, 106] conducted surveys of scientific workloads and we look to the information presented in these papers for both inspiration (in our early work) and, in the case of the more recent works, validation of our design decisions.

The *MapReduce* approach has been applied to computations in the realm of data visualization with some success [142].

We feel that our work fits well into the model of scientific computing laid out in a recent paper that sought to describe the requirements for a cloud computing system for science [107]. Specifically, our contributions would serve as the middle-ware layer in a scientific computing software stack.

The work presented in this thesis focuses on scientific file formats that have Java implementations available. For file formats with no Java implementation, Hadoop supports a more basic interface called Hadoop streaming. This form of Hadoop has been shown to be less efficient, due to its less granular approach to reading data (input data is presented as a byte-stream to the *Map* task which then must convert that data into key/value pairs). The MARISSA project [31] notes that several scientific file formats lack Java libraries and sought to improve the performance of Hadoop streaming for scientific data. Their work utilizes a shared filesystem for storage and seems to ignore data locality of any kind. While interesting for small clusters, the ability of the shared filesystem to serve an increasing load of non-coordinated accesses seems questionable.

Chapter 10

Conclusion

The work described in this thesis represents the first, to our knowledge, attempt to extend Hadoop to efficiently process scientific data while maintaining the goals of the original *MapReduce* project: high rates of data locality, flexible scheduling and fault tolerance. Furthermore, we have leveraged the unique properties of scientific data to further extend the *MapReduce* model, improving performance and adding new capabilities to Hadoop that were not previously possible. Our work to date has contributed to the current state of the art by producing new analysis of existing aspects of *MapReduce* (holistic combiners and *MapReduce*'s internal communication model), new theory for extending *MapReduce* to efficiently process scientific data, and thorough experiments showing the efficacy of implementations based on these theories.

Bibliography

[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] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, October 1998.

[3] Mingyuan An, Yang Wang, and Weiping Wang. Using index in the mapreduce framework. In *Web Conference (APWEB), 2010 12th International Asia-Pacific*, pages 52–58, April 2010.

[4] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[5] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.

[6] A. Andrejev, S. Toor, A. Hellander, S. Holmgren, and T. Risch. Scientific analysis by queries in extended sparql over a scalable e-science data store. In *eScience (eScience), 2013 IEEE 9th International Conference on*, pages 98–106, Oct 2013.

[7] Avro Homepage. `http://avro.apache.org/`.

[8] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: a programming model and execution framework for web-scale

analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[9] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. *SIGMOD Rec.*, 27:575–577, June 1998.

[10] J Bent, D Thain, A Arpaci-Dusseau, R Arpaci-Dusseau, and M Livny. Explicit control in a badfs. In *Proc. First Conf. Networked Systems Design and Implementation (NSDI)*, 2004.

[11] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[12] Stephen Blott, Lukas Relly, and Hans-Jörg Schek. An open abstract-object storage system. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 330–340, New York, NY, USA, 1996. ACM.

[13] 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.

[14] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[15] 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.

[16] Joe Buck, Noah Watkins, Greg Levin, Adam Crume, Kleoni Ioannidou, Scott Brandt, Carlos Maltzahn, Neoklis Polyzotis, and Aaron Torres. Sidr: Structure-aware intelligent data routing in hadoop. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 73:1–73:12, New York, NY, USA, 2013. ACM.

[17] Joe Buck, Noah Watkins, Greg Levin, Adam Crume, Kleoni Ioannidou, Scott A. Brandt, Carlos Maltzahn, and Neoklis Polyzotis. Sidr: Efficient Structure-aware Intelligent Data Routing in SciHadoop. Technical Report UCSC-SOE-12-08, UCSC, 2012.

[18] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 66:1–66:11, New York, NY, USA, 2011. ACM.

[19] Hoang Bui, Peter Bui, Patrick Flynn, and Douglas Thain. Roars: A scalable repository for data intensive scientific computing. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 766–775, New York, NY, USA, 2010. ACM.

[20] K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012.* ACM, 2012.

[21] Climate Data Specific Utilities: The cdutil Package. `http://www2-pcmdi.llnl.gov/cdat/manuals/cdutil/cdat_utilities-1.html#pgfId-998201`.

[22] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[24] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 523–534, New York, NY, USA, 2010. ACM.

[25] Lu Cheng, Pengju Shang, S. Sehrish, G. Mackey, and Jun Wang. Concentric layout, a new scientific data distribution scheme in hadoop file system. In *Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*, pages 231–239, July 2010.

[26] CMIP3 homepage. `http://www-pcmdi.llnl.gov/ipcc/about_ipcc.php`.

[27] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *Proceedings of the 7th USENIX conference*

on Networked systems design and implementation, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[28] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in MapReduce. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1115–1118, New York, NY, USA, 2010. ACM.

[29] Mariano P. Consens, Kleoni Ioannidou, Jeff LeFevre, and Neoklis Polyzotis. Divergent physical design tuning for replicated databases. In Candan et al. [20], pages 49–60.

[30] 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.

[31] E. Dede, Z. Fadika, J. Hartog, M. Govindaraju, L. Ramakrishnan, D. Gunter, and R. Canon. Marissa: Mapreduce implementation for streaming science applications. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–8, Oct 2012.

[32] Elif Dede, Madhusudhan Govindaraju, Daniel Gunter, and Lavanya 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.

[33] MPI. `http://en.wikipedia.org/wiki/Dining_philosophers_problem`.

[34] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, September 2010.

[35] J. Ekanayake, T. Gunarathne, G. Fox, A.S. Balkir, C. Poulain, N. Araujo, and R. Barga. DryadLINQ for Scientific Analyses. In *e-Science, 2009. e-Science '09. Fifth IEEE International Conference on*, pages 329 –336, dec. 2009.

[36] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 277 –284, dec. 2008.

[37] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *Proceedings*

of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[38] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 689–692, New York, NY, USA, 2012. ACM.

[39] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. Diskreduce: Raid for data-intensive scalable computing. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 6–10, New York, NY, USA, 2009. ACM.

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

[41] Blake G. Fitch, Aleksandr Rayshubskiy, Michael C. Pitman, T. J. Christopher Ward, and Robert S. Germain. Using the active storage fabrics model to address petascale storage challenges. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 47–54, New York, NY, USA, 2009. ACM.

[42] Fits homepage. `http://fits.gsfc.nasa.gov/`.

[43] Flume homepage. `http://incubator.apache.org/flume/`.

[44] General Utilities: The genutil Package. `http://www2-pcmdi.llnl.gov/cdat/manuals/cdutil/cdat_utilities-2.html#pgfId-998200`.

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

[46] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Rec.*, 34(4):34–41, December 2005.

[47] Grib homepage. `http://www.ecmwf.int/products/data/software/grib_api.html`.

[48] T. Gunarathne, J. Qiu, and G. Fox. Iterative mapreduce for azure cloud. *Proceedings of CCA11 Cloud Computing and Its Applications April*, pages 12–13, 2011.

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

[50] Hadoop homepage. `http://hadoop.apache.org/`.

[51] Hamster: Hadoop and MPI on the same cluster. `https://issues.apache.org/jira/browse/MAPREDUCE-2911`.

[52] Wikipedia *hashCode article.* *`http://en.wikipedia.org/wiki/Java_hashCode%28%29`*.

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

[54] Hdf5 homepage. *`http://www.hdfgroup.org/HDF5/`*.

[55] B. Heintz, Chenyu Wang, A. Chandra, and J. Weissman. Cross-phase optimization in mapreduce. In Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 338–347, March 2013.*

[56] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. SIGMOD Rec.*, 26(2):171–182, June 1997.*

[57] Hive homepage. *`http://hive.apache.org/`*.

[58] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. Towards Efficient MapReduce Using MPI. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of* Lecture Notes in Computer Science*, pages 240–249. Springer Berlin / Heidelberg, 2009.*

[59] Hadoop Online Prototype. *`https://code.google.com/p/hop/`*.

[60] Botong Huang, Shivnath Babu, and Jun Yang. Cumulon: Optimizing statistical data analysis in the cloud. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1–12, New York, NY, USA, 2013. ACM.*

[61] Hailiang Huang, Sandeep Tata, and Robert J Prill. Bluesnp: R package for highly scalable genome-wide association studies using hadoop clusters. Bioinformatics*, 29(1):135–136, 2013.*

[62] iRODS. *`https://irods.org`*.

[63] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.*

[64] *Michael Isard and Yuan 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, pages 987–994, New York, NY, USA, 2009. ACM.*

[65] *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.*

[66] *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, pages 13–, Washington, DC, USA, 2007. IEEE Computer Society.*

[67] *Z. Ivezic, J. A. Tyson, T. Axelrod, D. Burke, C. F. Claver, K. H. Cook, S. M. Kahn, R. H. Lupton, D. G. Monet, P. A. Pinto, M. A. Strauss, C. W. Stubbs, L. Jones, A. Saha, R. Scranton, C. Smith, and LSST Collaboration. LSST: From Science Drivers To Reference Design And Anticipated Data Products. In* American Astronomical Society Meeting Abstracts 213*, volume 41 of* Bulletin of the American Astronomical Society*, page 460.03, January 2009.*

[68] *Wei Jiang, Vignesh T. Ravi, and Gagan Agrawal. A map-reduce system with an alternate api for multi-core environments. In* Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 84–93, Washington, DC, USA, 2010. IEEE Computer Society.*

[69] *Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In* Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.*

[70] *H.T. Kung, Chit-Kwan Lin, D. Vlah, and G.B. Scorza. Speculative Pipelining for Compute Cloud Programming. In* MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*, pages 2026 –2034, 31 2010-nov. 3 2010.*

[71] *YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In* Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 75–86, New York, NY, USA, 2010. ACM.*

[72] *YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune in action: Mitigating skew in mapreduce applications.* Proc. VLDB Endow.*, 5(12):1934–1937, August 2012.*

[73] *YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome 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.*

[74] *Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei 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, pages 85–94, New York, NY, USA, 2011. ACM.*

[75] *LHC homepage.* `http://lhc.web.cern.ch/lhc/`*.*

[76] *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.*

[77] *Liang Lin, Vera Lychagina, and Michael Wong. Tenzing a sql implementation on the mapreduce framework.* Proceedings of the VLDB Endowment*, 4(12):1318–1327, 2008.*

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

[79] *LSST homepage.* `http://www.lsst.org/lsst/`*.*

[80] *Xiaonan Ma and A.L.N. Reddy. Mvss: an active storage architecture.* Parallel and Distributed Systems, IEEE Transactions on*, 14(10):993 – 1005, oct. 2003.*

[81] *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.*

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

[83] *Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In* Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.*

[84] *MIIPS FITS.* `http://www.chara.gsu.edu/~gudehus/fits_library_package.html`.

[85] *Million song dataset.* `http://labrosa.ee.columbia.edu/millionsong/`.

[86] *MPI.* `http://en.wikipedia.org/wiki/Message_Passing_Interface`.

[87] *Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In* Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.*

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

[89] *Netcdf homepage.* `http://www.unidata.ucar.edu/software/netcdf/`.

[90] *Phuong Nguyen and Milton Halem. A mapreduce workflow system for architecting scientific data intensive applications. In* Proceedings of the 2Nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, pages 57–63, New York, NY, USA, 2011. ACM.*

[91] *Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In* Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 1–15, Berkeley, CA, USA, 2012. USENIX Association.*

[92] *H. Ogawa, H. Nakada, R. Takano, and T. Kudoh. Sss: An implementation of key-value store based mapreduce framework. In* Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 754 –761, 30 2010-dec. 3 2010.*

[93] *Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In* USENIX Annual Technical Conference*, pages 267–273, 2008.*

[94] *Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In* Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.*

[95] *Apache Oozie Workflow Scheduler for Hadoop.* `http://incubator.apache.org/oozie/`*.*

[96] *OPenDAP.* `http://www.opendap.org`*.*

[97] *John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram.* SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.*

[98] *Aleatha Parker-Wood, Darrell D. E. Long, Brian A. Madden, Ian F. Adams, Michael McThrow, and Avani Wildani. Examining extended and scientific metadata for scalable index designs. In* Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 4:1–4:6, New York, NY, USA, 2013. ACM.*

[99] *Akshay Mittal Sharvanath Pathak and Trevor Bannard. Rhadoop: An improved execution environment for restricted mapreduce programs.*

[100] *T. Peterka, R. Ross, B. Nouanesengsy, Teng-Yok Lee, Han-Wei Shen, W. Kendall, and J. Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In* Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 580–591, May 2011.*

[101] *Pig homepage.* `http://pig.apache.org/`*.*

[102] *Steven J. Plimpton and Karen D. Devine. Mapreduce in mpi for large-scale graph algorithms.* Parallel Computing*, 37(9):610 – 632, 2011.*

[103] *Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with fahrrad.* SIGOPS Oper. Syst. Rev.*, 42(4):13–25, April 2008.*

[104] *Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In* Proceedings of the 9th USENIX conference on Operating systems

design and implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.*

[105] protobuf Homepage. `http://code.google.com/p/protobuf/`*.*

[106] Judy Qiu, Jaliya Ekanayake, Thilina Gunarathne, Jong Choi, Seung-Hee Bae, Hui Li, Bingjing Zhang, Tak-Lon Wu, Yang Ruan, Saliya Ekanayake, Adam Hughes, and Geoffrey Fox. Hybrid cloud and cluster computing paradigms for life science applications. BMC Bioinformatics*, 11(Suppl 12):1–6, 2010.*

[107] Lavanya Ramakrishnan, Keith R. Jackson, Shane Canon, Shreyas Cholia, and John Shalf. Defining future platform requirements for e-science clouds. In Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 101–106, New York, NY, USA, 2010. ACM.*

[108] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13 –24, feb. 2007.*

[109] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. Yahoo! Labs Technical Report YL-2012-002, 2012.

[110] E. Riedel, C. Faloutsos, G.A. Gibson, and D. Nagle. Active disks for large-scale data processing. Computer*, 34(6):68 –74, jun 2001.*

[111] Rob Ross. Meeting the Needs of Computational Science at Extreme Scale, January 2009.

[112] R Project. `http://www.r-project.org`*.*

[113] Scientifc Grand Challenge: Architectures and Technology for Exascale Computing. Technical report, US Department of Energy, 2011.

[114] John L Schnase, Daniel Q Duffy, Glenn S Tamkin, Denis Nadeau, John H Thompson, Cristina M Grieg, Mark A McInerney, and William P Webster. Merra analytic services: Meeting the big data challenges of climate science through cloud-enabled climate analytics-as-a-service. Computers, Environment and Urban Systems*, 2014.*

[115] SDSS homepage. `http://www.sdss.org/`*.*

[116] *Saba Sehrish, Grant Mackey, Jun Wang, and John Bent. Mrap: a novel mapreduce-based framework to support hpc analytics applications with access patterns. In* Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 107–118, New York, NY, USA, 2010. ACM.*

[117] *Sangwon Seo, Ingook Jang, Kyungchang Woo, Inkyo Kim, Jin-Soo Kim, and Seungryoul Maeng. Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment. In* Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –8, 31 2009-sept. 4 2009.*

[118] *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.*

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

[120] *Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Evolving rpc for active storage.* SIGARCH Comput. Archit. News*, 30(5):264–276, October 2002.*

[121] *E. Soroush and M. Balazinska. Time travel in a scientific array database. In* Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 98–109, April 2013.*

[122] *Emad Soroush and Magdalena 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.*

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

[124] *Jerome Soumagne, John Biddiscombe, and Jerry Clarke. An hdf5 mpi virtual file driver for parallel in-situ post-processing. In Rainer Keller, Edgar Gabriel, Michael Resch, and Jack Dongarra, editors,* Recent Advances in the Message Passing Interface*, volume 6305 of* Lecture Notes in Computer Science*, pages 62–71. 2010.*

[125] *M. Stonebraker and U. Cetintemel. "one size fits all": an idea whose time has come and gone. In* Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 2 – 11, april 2005.*

[126] *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.*

[127] *Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of scidb. In Judith Bayard Cushing, James French, and Shawn Bowers, editors,* Scientific and Statistical Database Management*, volume 6809 of* Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2011.*

[128] *Yu Su and G. Agrawal. Supporting user-defined subsetting and aggregation over parallel netcdf datasets. In* Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 212–219, May 2012.*

[129] *Alexander S. Szalay. The sloan digital sky survey and beyond.* SIGMOD Rec.*, 37(2):61–66, June 2008.*

[130] *Alexander S. Szalay and Jose A. Blakeley. Gray's Laws: Database-centric Computing in Science. August 2010.*

[131] *Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In* Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.*

[132] *K.E. Taylor. CMIP5 update and issues for the WGCM, 2010.*

[133] *Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework.* Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.*

[134] *Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In* Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM.*

[135] *http://www.top500.org.* `http://www.top500.org`.

[136] *Viet-Trung Tran, Bogdan Nicolae, and Gabriel Antoniu. Towards scalable array-oriented active storage: the pyramid approach.* SIGOPS Oper. Syst. Rev.*, 46(1):19–25, February 2012.*

[137] *Emalayan Vairavanathan, Samer Al-Kiswany, Lauro Beltrão Costa, Zhao Zhang, Daniel S. Katz, Michael Wilde, and Matei Ripeanu. A workflow-aware storage system: An opportunity study. In* Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 326–334, Washington, DC, USA, 2012. IEEE Computer Society.*

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

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

[140] *Kaushik Veeraraghavan, Jason Flinn, Edmund B. Nightingale, and Brian Noble. qufiles: The right file at the right time.* Trans. Storage*, 6(3):12:1–12:28, September 2010.*

[141] *A. Verma, N. Zea, B. Cho, I. Gupta, and R.H. Campbell. Breaking the mapreduce stage barrier. In* Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 235 –244, sept. 2010.*

[142] *H.T. Vo, J. Bronson, B. Summa, J.L.D. Comba, J. Freire, B. Howe, V. Pascucci, and C.T. Silva. Parallel visualization on large clusters using mapreduce. In* Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 81–88, Oct 2011.*

[143] *Jens-Sönke Vöckler, Gideon Juve, Ewa Deelman, Mats Rynge, and Bruce 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.*

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

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

[146] *Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In* Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 591–602, New York, NY, USA, 2010. ACM.*

[147] *Y. Wang, W. Jiang, and G. Agrawal. SciMATE: a novel MapReduce-like framework for multiple scientific data formats. CCGrid '12. ACM, 2012.*

[148] *Yi Wang, Yu Su, and G. Agrawal. Supporting a light-weight data management layer over hdf5. In* Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 335–342, May 2013.*

[149] *Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In* Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.*

[150] *Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: controlled, scalable, decentralized placement of replicated data. In* Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.*

[151] *B. Welton, D. Kimpe, J. Cope, C.M. Patrick, K. Iskra, and R. Ross. Improving i/o forwarding throughput with data compression. In* Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 438–445, Sept 2011.*

[152] *Qiangju Xiao, Pengju Shang, and Jun Wang. Co-located compute and binary file storage in data-intensive computing. In* Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*, pages 199–206, June 2012.*

[153] *Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In* First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.*

[154] *Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In* Proceedings of the 2013 ACM

SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.*

[155] Lei Xu, Ziling Huang, Hong Jiang, Lei Tian, and David Swanson. Vsfs: A versatile searchable file system for hpc analytics. 2013.

[156] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.*

[157] R.M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 198–207, Oct 2009.*

[158] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 247–260, New York, NY, USA, 2009. ACM.*

[159] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.*

[160] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI '12, pages 689–692, 2012.

[161] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion 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.*

[162] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.*

[163] Chen Zhang, Hans Sterck, Ashraf Aboulnaga, Haig Djambazian, and Rob Sladek. Case study of scientific data processing on a cloud using hadoop. In DouglasJ.K. Mewhort, NatalieM. Cann, GaryW. Slater, and ThomasJ. Naughton, editors, High Performance Computing Systems and Applications*, volume 5976 of* Lecture Notes in Computer Science*, pages 400–415. Springer Berlin Heidelberg, 2010.*

[164] Guangqing Zhang, Chuanjie Xie, Lei Shi, and Yunyan Du. A tile-based scalable raster data management system based on hdfs. In Geoinformatics (GEOINFORMATICS), 2012 20th International Conference on*, pages 1–4, June 2012.*

[165] Xiaohong Zhang, Yuhong Feng, Shengzhong Feng, Jianping Fan, and Zhong Ming. An effective data locality aware task scheduling method for mapreduce framework in heterogeneous environments. In Cloud and Service Computing (CSC), 2011 International Conference on*, pages 235–242, Dec 2011.*

[166] Yi Zhang, Weiping Zhang, and Jun Yang. I/o-efficient statistical computing with riot. In Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 1157 –1160, march 2010.*

[167] Ying Zhang, Martin Kersten, and Stefan Manegold. Sciql: Array data processing inside an rdbms. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1049–1052, New York, NY, USA, 2013. ACM.*

[168] 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.*

[169] Fang Zheng, Hongbo Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, Tuan-Anh Nguyen, Jianting Cao, H. Abbasi, S. Klasky, N. Podhorszki, and Hongfeng Yu. Flexio: I/o middleware for location-flexible scientific data analytics. In Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 320–331, May 2013.*

[170] Yunqin Zhong, Jizhong Han, Tieying Zhang, and Jinyun Fang. A distributed geospatial data storage and processing framework for large-scale webgis. In Geoinformatics (GEOINFORMATICS), 2012 20th International Conference on*, pages 1–7, June 2012.*

Appendix A

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, and Gigabit Ethernet, running Ubuntu 10.10 [15, 18] or 12.04 [17, 16]. SIDR built upon the SciHadoop code [45] that we ported from Hadoop 0.23 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 for 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 with 4 *Map* task slots and 3 *Reduce* task slots (testing showed these were the optimal settings for our cluster).