SDQuery DSI: Integrating Data Management Support with a Wide Area Data Transfer Protocol

Yu Su Yi Wang Gagan Agrawal
Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{su1,wayi,agrawal}@cse.ohio-state.edu

Rajkumar Kettimuthu
The University of Chicago and
Argonne National Laboratory
Argonne, IL 60439
{kettimuthu}@mcs.anl.gov

ABSTRACT

In many science areas where datasets need to be transferred or shared, rapid growth in dataset size, coupled with much slower increases in wide area data transfer bandwidths, is making it extremely hard for scientists to analyze the data. This paper addresses the current limitations by developing *SDQuery DSI*, a GridFTP plug-in that supports flexible server-side data subsetting. An existing GridFTP server is able to dynamically load this tool to support new functionality. Different queries types (query over dimensions, coordinates and values) are supported by our tool. A number of optimizations, like parallel indexing, performance model for data subsetting, and parallel streaming are also applied. We compare our *SDQuery DSI* with GridFTP default *File DSI* in different network environments, and show that our method can achieve better efficiency in almost all cases.

Categories and Subject Descriptors

H.3.1 [**Information Systems**]: INFORMATION STORAGE AND RETRIEVAL—*Content Analysis and Indexing*; C.2.5 [**Computer Systems Organization**]: COMPUTER-COMMUNICATION NETWORKS —*Local and Wide-Area Networks*

Keywords

data management; wide area networks; indexing; query processing; I/O performance tuning;

1. INTRODUCTION

As science has become increasingly data-driven, and as data volumes and velocities are increasing, scientific advances in many areas will only be feasible if critical 'big-data' problems are addressed - and even more importantly, software tools embedding these solutions are readily available to the scientists. Moving forward, the key challenge being faced by data-intensive science efforts is that while the dataset sizes continue to grow rapidly, disk speeds and wide-area transfer bandwidths are not coping up. Thus, software tools for dealing with scientific data must be enhanced to incorporate new approaches, for data-driven scientific advances to be maintained in the future.

Increasing data volumes and velocities are seen from a variety of data collection modalities. For example, in X-ray Photon Correlation Spectroscopy (XPCS), the detection of electric charge movement is done using a charge-coupled device (CCD). Though the current state-of-the-art CCDs operate at 60 frames-per-second, technology to produce 22,000 frames-per-second is expected by 2015, representing a 350 times increase in data volumes and velocity. On the other hand, with growing computational capabilities of parallel machines, temporal and spatial scales of simulations are also becoming increasingly fine-grained. The Community Earth System Model (CESM) is reducing the spatial scale from 1 degree to 0.125 degree, implying a factor of 64 increases in the output sizes.

As we stated above, wide area data transfer bandwidths are growing at a much slower pace, making it extremely hard for scientists to transport these rapidly growing datasets. Similarly, disk speeds are also not coping up, making it difficult for application scientists to manage and process large datasets. Support for management and analysis of scientific datasets has been a very active topic of research in recent years. On one hand, new database approaches like the SciDB system [24] focus specifically on scientific data. On the other hand, indexing techniques suitable for scientific datasets have also been developed [30, 26]. Use of database and/or indexing techniques can allow a subset of data of interest to be extracted from a repository, and such subsetting, if performed before a data transfer, can likely reduce the volume of the data to be transferred, and subsequently stored and analyzed at the client-side.

In most cases, the practical state-of-art of sharing and movement of scientific data remains very limited, in terms of any possible incorporation of (efficient) data management techniques. Though a variety of methods and protocols may be used for supporting data transfers (including scientists shipping CDs, which tends to be common even today!), GridFTP [8] and its Software as a Service (SaaS) version, Globus Online, are extremely popular. GridFTP provides additional security and performance over the default FTP implementations, including striped, streaming, and/or parallel, as well as more reliable and restartable data transfers. However, with an exception of integration of GridFTP with OPeNDAP [7] (which only provides limited flexibility and efficiency), the unit of data transfer for GridFTP is a single file. While enhancing and optimizing data transfer frameworks [2, 14, 16, 17, 18, 13, 4] has continued to be an active area of research, the ability to reduce data volume that needs to be transferred over the wide-area, by providing support for *user-defined data subsetting* at the server-side, is clearly needed.

In this paper, we address several important challenges that arise in integration of core data management functionality (efficient data subsetting) with a protocol for data movement over a wide-area network. Specifically, the issues that need to be addressed are:

- How should a system integrating basic data management support with a data transfer protocol be designed to allow easy use and integration with existing environments?
- How can users *view* a remote file, which uses one of the popular scientific data formats like NetCDF or HDF5, and conveniently specify the subset of the data that is of interest to them?
- In retrieving a subset of a file from a disk, when is it appropriate to use an indexing-based retrieval over a simple read

followed by in-memory filtering of data, and can this decision be automated?

- How can data retrieval and filtering be parallelized, to make use of multiple cores and the likely benefits from using multiple streams, to achieve efficient utilization of the underlying network?

This paper develops solutions for the above problems, and incorporates them in a tool we refer to as *SDQuery DSI* (Scientific Data Query Data Storage Interface), a GridFTP plug-in which supports flexible server-side data subsetting over HDF5 and NetCDF data formats. The GridFTP server is able to dynamically load this tool if it needs to perform subsetting before data transfers, and switch back to using other DSIs subsequently. Different subsetting predicates (queries over dimensions, queries over coordinates, and queries over values) are supported by our tool, which is made possible using existing metadata as well as bitmap indexing. Besides the basic functionality, we also include the following features: 1) we use a performance model to automatically select between indexing-based retrieval of data segments and full retrieval followed by in-memory filtering, 2) we support a *parallel streaming* technique, where different disk blocks are read concurrently and piped to different TCP/IP streams, and 3) we incorporate parallel indexing to perform indexing operations for each sub-block concurrently. Overall, while the data management functionality provided in this system has some similarities with other efforts (e.g., the NoDB approach [1] and the ADIOS project [15]), no other project has provided a high-level API for specifying subsetting conditions on NetCDF and HDF5 files, while also integrating (and optimizing) such subsetting functionality with a data transfer protocol.

We have extensively evaluated our implementation. We first compared our GridFTP version *SDQuery DSI* with the GridFTP default *File DSI* and show that our method can achieve better efficiency in almost all cases, with only exception being where a query outputs a large fraction of the original dataset and the network bandwidth is also very high. We also show that our performance model-based hybrid data reading method is effective, i.e., it can automatically choose between indexing-based direct access and the in-memory filtering method. The parallel streaming technique we have implemented is able to improve both data read and data transfer efficiency. Finally, we also show that the parallel indexing can improve the index operation performance.

2. MOTIVATING APPLICATIONS

Many data sharing and transfer scenarios involve increasing dataset sizes and significant benefits from subsetting data before transfers. Several of these scenarios arise in the context of applications where GridFTP is already being used for data transfers, though lack of integration of any data management solution with GridFTP is limiting efficiency.

Analysis of Climate Simulation Outputs: Climate simulations like the Community Earth System Model (CESM), or its predecessor Community Climate System Model (CCSM), are producing massive datasets. CESM project has been jointly sponsored by NSF and DOE, and its output is of immense value to a variety of researchers.

The current output organization involves keeping all the variables for the entire globe, for one time-slice, in a single NetCDF file. In the future, the organization is likely to be changed to one variable, entire time-series, and the entire globe, in one NetCDF file. However, most researchers focus on a specific geographical region (and often certain time-ranges). This involves spatial or spatio-temporal subsetting of data over a Cartesian (non-rectilinear) grid. Moreover, data needs to be transported over wide area networks. For example, one common operation is: one dataset is at one location (possibly within one organization), another dataset is stored across the WAN, and user wants the same geospatial subset from each, take difference of values (for certain attributes), and then visualize the results at their location. Several climate scientists extensively use GridFTP for such data movements, but currently spend unnecessary time because of its inability to select subsets at the server-side. Moreover, as simulation outputs scale, they face an enormous challenge.

Data Pipelines from Tomography: Two and three dimensional x-ray imaging studies of dynamical phenomena, with spatial resolution as small as tens of nanometers, are popular methods for new material characterization. However, when scientists approach a facility with state-of-the-art Tomography facility (such as the Imaging Group at Argonne), a large volume of data is generated, and this volume will increase rapidly in the near future (10 GB/second by 2015). Moreover, after this data is processed and a 3-d representation is created, the amount of data increases. Such processed data needs to be moved to the scientist's home institution for further studies, and slow wide-area networks are clearly a bottleneck. While some scientists use GridFTP, others move data physically after copying them to CDs, and in fact, spend additional days at the facility waiting for the data to be copied onto the CDs. Moving data by CDs is also highly unreliable - e.g., airports scanners frequently corrupt the data.

It turns out that a very high fraction of data is not even useful for scientists. After preprocessing, data is stored in the HDF5 format, and a simple query mechanism on a HDF5 file can help reduce the data sizes by a large number, and make transfer feasible and efficient using GridFTP.

Data Pipelines from X-ray Photon Correlation Spectroscopy (XPCS): XPCS is a powerful technique to probe the dynamics in materials, with classical applications including the Brownian diffusion in liquids, and more recent applications like understanding the effect of the changes in proteins on diseases. The state-of-the-art CCD detector which captures the phenomenon operates continuously at 60 fps (frames per second), streaming one million (1M) pixels in each frame, and thus producing 120 MB/sec of data. New detectors that are suitable for XPCS will be available in 2015, and will stream 1M pixels at 22,000 fps, facilitating better understanding of biological processes. However, current technologies are completely inadequate for moving data arriving at such rates to a compute-cluster within the organization, and subsequently to scientists over the entire world. GridFTP has been currently being used for data movement from the device to the compute cluster, and then to scientists' home organizations [20]. With support for subsetting, future data rates can be adequately handled.

3. SYSTEM OVERVIEW

This section gives an overview of the system we have implemented. Optimization methods are presented in the next section.

3.1 Background: Globus GridFTP

While the underlying ideas in our work on integrating data management support with a data transfer protocol are general, our implementation has been in the context of Globus GridFTP. Globus GridFTP has become an important high-performance data transfer tool for the scientific community. Currently, the GridFTP server is deployed on more than 3,500 servers all over the world and is responsible for an average of more than 10 million transfers every day, moving more than one petabyte of data. Its modular architecture provides a very convenient way for GridFTP-compliant clients to access to any storage system, provided that an implementation of GridFTP's Data Storage Interface (DSI) specific to this storage system is available. It also supports an eXtensible I/O interface, which allows GridFTP to target high-performance wide-area communication protocols such as UDT and RDMA-based protocols. Globus GridFTP is optimized to handle different types of datasets - from dataset containing one single, large file to those comprising a number of small files.

Figure 1 shows the general Globus GridFTP architecture. From the figure, we can see that it comprises three components: two Protocol Interpreters (PIs), which are the server and the client protocol interpreters, and the Data Transfer Process (DTP). PIs are used to handle the control channel protocol. Because GridFTP follows an asymmetric protocol exchange, the client PI is different from the server PI. The DTP is used to handle access to the actual data and its movement via the data channel protocol. These three components can be combined in various ways to create servers with different ca-

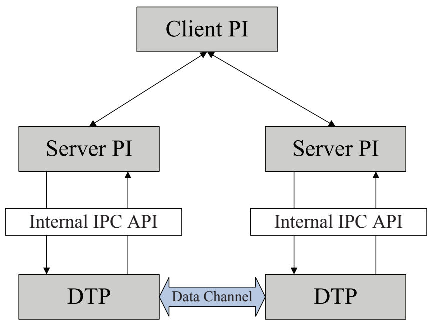


Figure 1: Globus GridFTP Architecture

pabilities. DTP can be further divided into a three-module pipeline: the data access module, the data processing module, and the data channel protocol module. The data access module provides an interface to the data sources (or *sinks*). The data processing module performs server-side data processing, if requested by an extended store/retrieve (ESTO/ERET) command. The data channel protocol module reads data from or writes data to the data channel.

In today's scientific cyberinfrastructure, there are a number of distributed storage systems. The protocols used and data access patterns across them vary substantially, as they all focus on meeting different requirements. To make GridFTP a general transfer protocol, Globus GridFTP provides a modular pluggable interface called the Data Storage Interface (DSI), which can be loaded and switched at runtime. When the GridFTP server requires service from the storage system, it first sends a request to the loaded DSI. To create a DSI, programmers need to implement a set of functions that are part of the API.

3.2 Overview of Desired Functionality

As we had summarized in Section 2, in many scenarios, scientists do not need to download the entire data file for analysis. They are only interested in a subset of the data, such as temperature within a specific area or a given value range. Hence, our goal is to integrate basic 'database-like' functionality of supporting user-defined subsetting with GridFTP's data transfer protocol. Our system design was motivated by the following requirements:

(1) *Support High-level Queries over Popular Scientific Data Formats:* Supporting data subsetting queries using a high-level language, over arbitrary flat or binary files, and without requiring data to be reorganized and/or loaded into a database system, is almost impossible. Thus, we focus on popular scientific data formats, and use the metadata associated with them to expose a high-level schema, which can be used to specify subsetting conditions. Our current implementation supports HDF5 and NetCDF, each of which is used across a number of scientific areas. There are several challenges in supporting a high-level query language on these, including how users can view the structure of these datasets and express their queries in an unambiguous way.

(2) *Support Variety of Subsetting Requirements:* Both HDF5 and NetCDF formats organize the data as a set of multi-dimensional arrays, which typically involve spatial and/or temporal dimensions and coordinates. Hence, subsetting situations that arise can be divided into three categories: queries based on dimensions, queries based on coordinates (*dimension scales* for HDF5 and *coordinate variables* for NetCDF), and queries based on values (*value-based queries*). Our system should be able to support all of these, and even a flexible combination of these three types, efficiently.

(3) *Interoperate with Existing GridFTP Server Binaries:* To make it easy for others to use our system, it is very desirable that reinstallation of the GridFTP server is not needed. Instead, current users of GridFTP should be able to simply download the additional functionality we are providing. As mentioned in Section 3.1, this can be achieved because GridFTP allows a new DSI to be loaded at run-

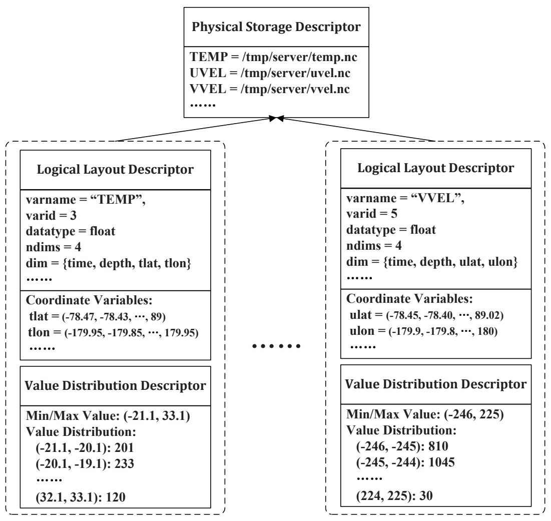

Physical Storage Descriptor
TEMP = /tmp/server/temp.nc
UVEL = /tmp/server/uvel.nc
VVEL = /tmp/server/vvel.nc
……
Logical Layout Descriptor
varname = "TEMP",
varid = 3
datatype = float
ndims = 4
dim = {time, depth, tlat, tlon}
……
Coordinate Variables:
tlat = (-78.47, -78.43, ···, 89)
tlon = (-179.95, -179.85, ···, 179.95)
……
Value Distribution Descriptor
Min/Max Value: (-21.1, 33.1)
Value Distribution:
(-21.1, -20.1): 201
(-20.1, -19.1): 233
……
(32.1, 33.1): 120
……
Logical Layout Descriptor
varname = "VVEL",
varid = 5
datatype = float
ndims = 4
dim = {time, depth, ulat, ulon}
……
Coordinate Variables:
ulat = (-78.45, -78.40, ···, 89.02)
ulon = (-179.9, -179.8, ···, 180)
……
Value Distribution Descriptor
Min/Max Value: (-246, 225)
Value Distribution:
(-246, -245): 810
(-245, -244): 1045
……
(224, 225): 30

Figure 2: A High-level NetCDF Data Schema

time. This way, the client can choose to download the entire file or perform subsetting before the download, and in the latter case, our DSI can be invoked and used.

(4) *Optimize for Different Subsetting Scenarios:* By supporting an index on an existing HDF5 or NetCDF file, we can retrieve from the disk only the subsets that are of interest to us, reducing I/O volumes. For queries where only a small fraction of the file needs to be retrieved, this is clearly advantageous. However, if a query is going to select a large fraction of the file, it may be more efficient to load either the entire file or the major data sub-blocks into memory, and then perform in-memory filtering, instead of performing a number of (possibly non-continuous) disk I/O accesses. Choosing which method will result in better performance is hard, but very desirable for efficiency.

(5) *Support Efficient Data Transfers after Subsetting:* It has been seen from many studies [9, 22] that in a wide-area network, using parallel TCP streams between single source and destination can improve the aggregate bandwidth achieved, over using a single stream. However, unlike the case when the entire file needs to be transferred, using parallel streams with data subsetting during retrieval is non-trivial.

3.3 Supporting Structured Queries: High-level Data Schema

One of the requirements we had listed earlier was *"support high-level queries over popular scientific data formats"*. We now describe how this requirement is met for NetCDF and HDF5 formats. Specifically, during the data storing process, we generate a high-level data schema that can be downloaded by users to guide their queries.

Figure 2 shows the data schema example of a NetCDF file, which is generated by the Parallel Ocean Program (POP) [11]. The three components of this schema are motivated by the following three requirements. First, query processing requires dataset physical storage information to locate the target data file for subsetting and downloading. Second, the users require the *logical layout information* of each variable to find the relationship among the variables, the dimensions and the coordinate values. Third, the users need to know the value ranges and distribution information, to help construction of value-based queries that can be meaningful.

Thus, returning to Figure 2, the three components of the scheme are: 1) Physical Storage Descriptor, which describes physical locations where each NetCDF variable is resident. By looking up this descriptor, users are able to specify which data file to subset and download. 2) Logical Layout Descriptor, which exposes the logic data layouts, including variable ids, data types, dimension names

ID	Value	e_0	e_1	e_2	e_3	i_0	i_1
		=1	=2	=3	=4	[1, 2]	[3, 4]
0	4	0	0	0	1	0	1
1	1	1	0	0	0	1	0
2	2	0	1	0	0	1	0
3	2	0	1	0	0	1	0
4	3	0	0	1	0	0	1
5	4	0	0	0	1	0	1
6	3	0	0	1	0	0	1
7	1	1	0	0	0	1	0
Dataset		Low Level Indices				High Level Indices	

Figure 3: An Example of Bitmap Indexing

and lengths, and coordinate values of the current variable. Specifically, coordinate variables, which are relatively small in size, are fully loaded to support the queries that are based on coordinate values. By looking up this descriptor, users are able to specify dimensions and/or coordinates based query conditions. This descriptor is generated by extracting the header of each NetCDF data file. 3) Value Distribution Descriptor, which describes the data values and a general value distribution over bins (how many elements within bins). Users are able to specify value-based query conditions by checking this descriptor. This descriptor is generated based on bitmap indices metadata, which we will describe next.

The data schema structure for the HDF5 data format is quite similar. One difference is that NetCDF and HDF5 use different terms. For example, HDF5 data format uses dataset instead of variable, data space instead of dimension and dimension scale instead of coordinate variable. Another difference is that because HDF5 dataset can be organized in a hierarchical structure, in which case the layout metadata may be dispersed in separate header blocks for each group. If this is the case, scattered logic metadata should be collected and grouped together.

In our system, the high-level schema provides a *virtual* relational table view to the user, who can now use SQL to express a variety of subsetting conditions. The reason why we support SQL is because it is the most popular database language, and various graphical front-ends currently available for SQL can allow a user to compose their queries interactively.

3.4 Supporting Subsetting Conditions: Bitmap Indexing

One of the requirements we had listed earlier was: *"support a variety of subsetting queries (efficiently)"*. Though dimension-based queries can be supported by using the metadata associated with NetCDF and HDF5, for value-based queries, one clearly needs indexing. Bitmap indexing, which utilizes the fast bitwise operations supported by the computer hardware, has been proved as an efficient approach for scientific data management [21, 33]. Moreover, it can be applied without any need for reorganization of data (which is not desirable in our case). Thus, our system uses bitmaps to help enable a variety of subsetting queries.

Figure 3 showed an example of bitmap indexing. In this simple example, the dataset contains a total of 8 elements with 4 distinct values $(1, 2, 3, 4)$. The *low-level* bitmap indices contain 4 bitvectors (e_0, e_1, e_2, e_3) and each bitvector corresponds to one value. Each bitvector contains a sequence of 0-bits and 1-bits, and the total number of bits is equal to total number of elements in the dataset. In each bitvector, a bit is set to 1 if the value for the corresponding data element's attribute is equal to the *bitvector value*, i.e., the particular distinct value for which this vector is created. The *high-level* indices can be generated based on either the value intervals or value ranges. From Figure 3, we can see two *high-Level* indices (i_0, i_1) are built based on value intervals. During query processing, a collection of bitvectors are extracted based on the value subsetting conditions. Logic AND or OR operations are performed among them to generate a point id set as the result.

Usually scientific dataset contains floating-point values which have extremely high cardinality. Bitmap indexing also has been proven to be an efficient method for floating-point values [32]. In such case, instead of building bitvector for each distinct value, we can first group a set of values together (binning) and build bitvectors for small bins. This way, the total number of bitvectors can be greatly decreased. From the example we can also see that the number of bits within each level bitmap indices is $n \times m$ (n is total number of elements and m is the total number of bitvectors), which is even greater than data itself. Existing methods use *run-length compression* [3, 31] to address these problems, which are incorporated in our system as well.

3.5 System Overview

We now describe how the major components of the system operate together. In the process, we also address the requirement of *"interoperating with existing GridFTP server binaries"*.

As mentioned in Section 3.1, one of the key features of GridFTP is the API for accessing any new data storage medium, referred to as the DSI. GridFTP also allows a new DSI to be loaded at runtime. Thus, the file-level subsetting functionality we provide is encapsulated as a new DSI, which we refer to as the scientific data query or *SDQuery* DSI.

Any data transfer protocol, including the Globus GridFTP data transfer protocol, can be divided into two phases: a preparatory phase, where a control channel is first built up, and then the operation of the data channel between the client and the server. The data subsetting optimization using our *SDQuery DSI* is applied during the data channel communication, whereas the control channel setup is used, unmodified, from the original GridFTP framework. Figure 4 shows the architecture of *DTP* (Data Transfer Process) between client and server using the *SDQuery DSI*. It should be noted that *SDQuery DSI* is also able to support third-party data transfer, i.e., a client can initiate transfer from one server to another server. However, Figure 4, as well as our discussion here, will focus on transfer from a server to a client only.

There are (up to) three different ways in which our system is used. First, it can be used to load a new NetCDF and HDF5 file in a way that indexing support can be generated, and a high-level data schema, based on which structured queries are to be written and executed, can be supported. Second, before issuing a high-level query, a user may want to request a high-level schema to help understand the dataset. Third, the user may want to retrieve a data subset with a structured query. From the figure, we can see that the first step in our workflow is to use the *request parser* to parse the request and check if it is a data store request, a data schema request, or a data retrieval request. Each kind of request is subsequently processed by the corresponding pipeline.

In the case of a data store request, the *index generation* component is invoked to build up multi-level bitmap indices for all variables included in the current file. The indices are stored as metadata with the original file. The *schema management* component generates a high-level data schema view of the current data file based on the file header and the index metadata. The entire index and data schema generation process runs at the backend so that other GridFTP clients are still able to download data files at the same time.

For a schema retrieval request, the *schema management* component will find all data schema files of the current dataset and the *file sender* will send them back to the client-side. The size of data schema file is typically much smaller than the dataset size, and thus, this operation can be performed efficiently. Based on the data schema, users are able to write SQL queries and further generate the data retrieval request. Figure 5 shows an example of a SQL query and data retrieval request based on the data schema we had shown earlier in Figure 2. This query's intent is to find the data elements within the Gulf of Mexico area, under the depth of 50 meters and where the temperature is larger than 5 centigrade. By looking up the *logical layout descriptor*, we can find all *longitude*, *latitude*, *depth* values and their relationship with the variable *TEMP*. By looking up the *value distribution descriptor*, we can find the value range of the variable *TEMP* and specify value-based query conditions that are likely to provide useful insights. A GridFTP data retrieval request is generated by embedding the SQL query. The

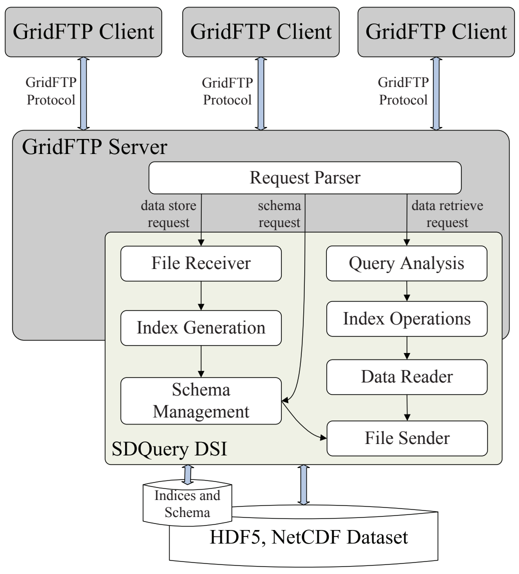
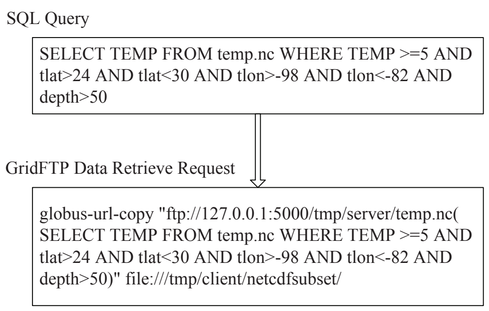


Figure 4: System Architecture



```
SELECT TEMP FROM temp.nc WHERE TEMP >=5 AND
tlat>24 AND tlat<30 AND tlon>-98 AND tlon<-82 AND
depth>50
```

```
globus-url-copy "ftp://127.0.0.1:5000/tmp/server/temp.nc(
SELECT TEMP FROM temp.nc WHERE TEMP >=5 AND
tlat>24 AND tlat<30 AND tlon>-98 AND tlon<-82 AND
depth>50)" file:///tmp/client/netcdfsubset/
```

Figure 5: An Example SQL Query and GridFTP Data Retrieval Request Embedding the Query

globus-url-copy tool provides a command-line client for requesting transfers to, from, or between GridFTP servers, and supports rich data transfer functionality by adding different command-line parameters. Among these parameters, the source URL includes the transfer protocol, the server IP address, the GridFTP port number, the target file location (obtained from the *Physical Storage Descriptor*), and the embedded SQL query. The destination URL contains the path where the data subset file is to be stored at the client-side.

After the server-side receives the data retrieval request, the *query analysis* component takes the request as the input and invokes the following modules: first, *SQL parser*, implemented by making certain modifications to the parser from SQLite [1], generates the parse tree. Next, the *metadata parser* takes the data file name and the variable name(s) as the input, looks up the corresponding metadata files, finds the data schema and the data layout information, and loads them into the memory. Finally, the *query request generator* is responsible for generating a data subsetting request by combining the SQL parse-tree information with the metadata information.

The *indexing operations* component takes the query request as the input, performs bitwise operations using the bitmap indices, and returns all data position identifiers (IDs) that satisfy the current query. Recall that there are three types of subsetting conditions. For the subsetting condition based on the dimension identifiers and the coordinate values, dimension bitvectors that satisfy these two query types are dynamically generated (for query over coordinates, a mapping is first applied to map the coordinate values to the dimension IDs). Within the dimension bitvectors, the bits with the value 1 satisfy the current dimension and/or coordinate query conditions. For query based on the variable values, the bitvectors that satisfy the current value ranges are read from the disk. After that, bitwise (logic AND/OR) operations are performed among the value bitvectors and the dimension bitvectors. Finally the *result bitvector* is returned. All 1-bits indicate the data positions that satisfy the current query. To improve data retrieval efficiency, parallel indexing approach is applied, where indices are built separately for different data sub-blocks. Details of this optimization method can be seen in Section 4.3. The *data reader* component takes the data position IDs generated in the previous step as the input, retrieves the data from the dataset chunk by chunk, and send chunks to the sending queue of the *file sender*. The *file sender* component dequeues the data chunks from the sending queue and sends them to the client-side. Two key optimizations, a performance model-based hybrid data retrieval method and a parallel streaming data transfer method, can be applied during this process to improve the efficiency. The detailed description of these two optimizations can be seen in Sections 4.1 and 4.2, respectively.

[1]http://www.sqlite.org

4. SYSTEM OPTIMIZATIONS

This section describes several optimizations implemented in the system.

4.1 Performance Model-Based Optimized Data Subset Retrieval

Consider processing of a *value-based* query given by the user. We can perform bitwise operations over the bitvectors that were generated earlier, and generate a *point-ID-set*, specifying the records that should be retrieved. This step normally does not consume much time, because the size of bitvectors is much smaller than that of the dataset, and fast bitwise operations can be performed efficiently in memory. However, we next need to read records that comprise the results of the query. As this step can potentially involve a number of distinct and possibly non-contiguous I/O requests, it can get expensive, and the advantage of subsetting can be easily undone.

Clearly, initiating a separate read operation for each record that needs to be read will most likely be prohibitively expensive. One simple optimization that can be applied will be to generate contiguous or almost contiguous segments of records that are needed by the query. This process is called *segmentation*. Thus, segments, instead of individual elements, can be read at any given time. One reason why this approach turns out to be quite effective in practice is that for most scientific datasets, neighboring records tend to have very similar values for any given attribute. Hence, reading based on segments will greatly decrease the I/O access times and improve the data reading efficiency.

In cases where segmentation is not sufficiently effective, we can choose to read either the entire dataset or data blocks (if the memory size is not sufficient to hold the entire dataset) from the disk, and then perform *memory filtering*, i.e., apply filtering conditions on each element in memory. Memory filtering is an alternative to the scheme in which each segment is individually read from the disk, which we also refer to as the *direct access* method. A key optimization built in our system involves automatically choosing between the two methods at the runtime, so as to minimize the data access times. This optimization exploits the fact that by using bitmap indices, we know the fraction of the data to be read after bitwise operations, even before performing any I/O operation.

Intuitively, we can see that if the subsetting percentage is relatively small, directly read the query results from the disk is likely to be more efficient compared to load the entire dataset into the memory. In comparison, if a large fraction of the data needs to be returned as query results, the direct access method will likely incur many distinct disk accesses, which can be time consuming. At the same time, memory filter will perform fewer (and more contiguous) data accesses, lowering disk I/O costs to a level that even after memory-based filtering, the total execution time will be less. However, except for the cases where only a very small amount of data or a very large fraction of data needs to be retrieved, the choice between the two methods depends on the data queries, the dataset itself, and the hardware.

Thus, we have developed a performance model to choose between these two options. Before we explain the details of this model, we note that for both HDF5 and NetCDF data formats, the data reading methods can be divided into three categories: reading the *entire dataset* (or all elements of a variable), reading a data block, and reading a data point. The time cost associated with each of them can be divided into three parts: the *seek time* and actual *retrieval time*, as in the case of any disk operation, and in addition, before every read operation, both HDF5 and NetCDF invoke several functions, the time cost of which is referred to as the *preparation time* of data reading. For reading the entire dataset (variable) or a sufficiently large data block, read operation time is the dominant factor, and in comparison, data seek and read preparation time can be ignored. For reading a small data block or a point, we need to explicitly include the read preparation time and the seek time in our model.

Table 1: Major Parameters of Auto-Tuning Model

Identifier	Description
DS	Size of the entire dataset
BN	Total number of blocks in the dataset
SS	Size of the query results
SN	Total number of segments
T_{db}	Average time to load one data block from disk to memory
T_{de}	Average time to load one data element from disk to memory
T_{ml}	Average time to filter one data element in memory
T_{seg}	Time for generating data segments
T_p	Average time to prepare for data read operation
T_{dl}	Average time to locate the start position in the dataset

The complete list of all parameters used in our model is shown in Table 1. We start our discussion by focusing on the costs associated with the memory filtering method. T_{MF} denotes the cost of producing the query results using the memory filter method, and can be calculated as:

$$T_{MF} = BN \times (T_p + T_{dl} + T_{db}) + FS \times T_{ml}. \quad (1)$$

$$FS = MIN\,(SS, DS - SS)\,. \quad (2)$$

Equation 1 can be divided into two parts. Because in most cases, the actual data size is much larger than the memory, the entire data is first logically divided into fix-sized blocks and each time one data block is loaded and filtered in memory. Hence, the entire disk I/O time is the product of the total number of blocks and the sum of the read preparation time, the data seek time, and the data transfer time of each block. The data transfer time is much larger than the read preparation time and the data seek time.

The second part of Equation 1 involves the term FS, which is the smaller value between the current subset size and the rest of the data size, as shown in Equation 2. With the help of bitmap indexing, we do not need to apply the filtering conditions on each record and choose data subset based on that. Instead we can directly locate the target data elements based on the point IDs within *point-ID-set*. This brings another optimization: when the data subsetting percentage is smaller than 50%, we can directly locate those 1-bits to select the data subset in memory; when the data subsetting percentage is larger than 50%, we can directly locate those 0-bits and select data elements between each 0-bits pair. This way, we ensure that the in-memory filter operation will be applied to at most 50% of the data elements for all different queries. Now, returning to Equation 1, the second term calculates the memory filtering time, which is the product of average filter time per element and FS.

Next, we focus on query processing using the direct access. Initially, we consider the case when segmentation is not used. The time for this approach is denoted as T_{DA1} and calculated as follows:

$$T_{DA1} = SS \times (T_p + T_{dl}) + SS \times T_{de}. \quad (3)$$

In this case, we have to prepare, seek and read each element separately. So the total time is the product of the size of data subset and the sum of the read preparation time, the data seek time, and the data transfer time. Alternatively, we can use segmentation, and the resulting cost will be:

$$T_{DA2} = T_{seg} + SN \times (T_p + T_{dl}) + SS \times T_{de}. \quad (4)$$

Using this method, although it has additional point segmentation cost, the total number of read preparations and seek operations can be much less.

Now, the goal of our model is to calculate the likely query processing costs using the two approaches, and choose the more efficient method. We now elaborate on how (and when) all parameters are obtained. Considering all parameters involved in our model, we can see that the total number of blocks (BN) and the dataset size (DS) are known for each dataset. For each query, after the indexing operations, we can also see the size of the query result (SS), the total number of segments involved (SN) and the segmenting time (T_{seg}). The parameters whose values are not readily available are the average time to prepare for the read operation (T_p), the average seek time (T_{dl}), the average time to load the data from disk to the memory - (T_{db}, per block, for the memory filter method and T_{de}, per element, for the direct access method), and the average time to filter data in memory (T_{ml}).

To improve the accuracy of the method, we obtain values of several of these parameters separately for each method, i.e., the memory filter and the direct access method, and in some cases, even for queries with different range of subsetting levels. In the case of memory filtering method, all data blocks have to be loaded into the memory first, which implies that the read preparation time (T_p), and the data block loading time (T_{db}) are constants. Moreover, for simplicity, the seek time (T_{dl}) is also treated as a constant. The total in-memory data filtering time is proportional to the number of target elements (either to be selected or to be skipped). The average filter time (T_{ml}) is easy to estimate based on results of an initial set of queries.

For the direct access method, the parameters that need to be trained are T_p, T_{dl} and T_{de}. T_p is (almost) identical for different queries, whereas T_{dl} and T_{de} are related to the data subsetting percentage. Specifically, the average seek time (T_{dl}) is inversely proportional to the subsetting percentage, whereas the average data transfer time T_{de} is proportional to the subsetting percentage. This is because the average segment length is, in practice, proportional to the data subsetting percentage. When the segment length becomes larger, the average data transfer speed increases. Based on these observations, we divide the training set of the direct access method into several buckets, based on subsetting percentage, e.g., 20%-30%, 30%-40%, and so on. The parameters T_{dl} and T_{de} are obtained based on execution of several queries from each bucket.

The training process we use is a combination of off-line training and on-line training. When the server is free, we perform the off-line training to estimate the parameters. Otherwise, we apply the on-line training based on users' real queries, and improve our estimate of different parameters. Specifically, during the training process, T_p, T_{dl}, T_{de}, T_{db}, and T_{ml} are continuously updated until each parameter reaches a relatively stable status.

4.2 Parallel Streaming Data Transfer

Parallel streaming data transfer, as supported in our system, involves the following: 1) the data subsetting operations are performed in parallel to improve disk I/O efficiency, 2) the data transfer is performed in parallel to improve network transfer efficiency,

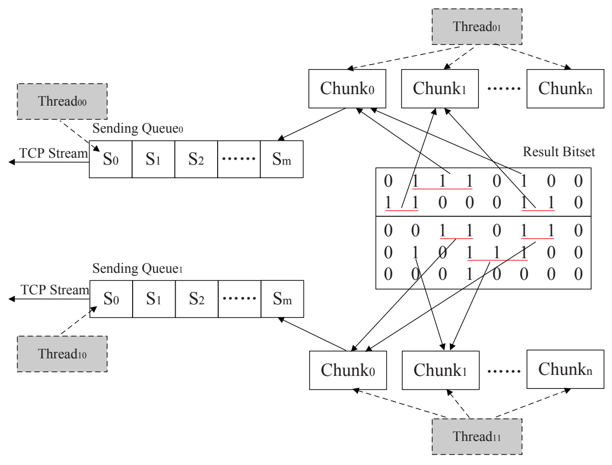


Figure 6: Parallel Streaming Data Transfer

and finally, 3) the data reading and data transfer operations are performed in a pipeline mode, and thus, the data reading time is amortized by the network transfer time. We now elaborate on some of the key aspects of implementation of this approach.

Figure 6 shows an example of parallel data streaming. The number of streams in this simple example is 2. Two threads are used in each stream, with $Thread_{00}$ and $Thread_{01}$ belonging to the first stream, and $Thread_{10}$ and $Thread_{11}$ belonging to the second stream. Within each stream, one thread is responsible for fetching data subset chunk by chunk and inserting data chunks into the *sending queue*, and another thread is responsible for extracting data chunks from the *sending queue* and transferring them to the client-side through network.

Initially, bitmap operations are performed sequentially, since they do not consume too much time. The result of this step is a *point-ID-set*, containing the point IDs (bits with the value 1) that satisfy the current query. As an example, in Figure 6, the result bitset contains 17 out of 40 elements.

The next step is crucial for parallel streaming performance. Here, *points segmenting* and *points partitioning* are used. Points segmenting groups continuous points into segments to decrease the disk read times. This method is used only if the direct access method is chosen. From Figure 6, we can see that with the help of segmenting, the total number of disk I/O accesses is 9 instead of 17. Points partitioning divides the dataset into blocks based on the number of streams, with each stream takes care of one data block. Here, we have two partitioning options: a *traditional* partition method would involve dividing the dataset into blocks with equal size based on the dimensions. This method is straightforward to implement, and has no partitioning overheads, but can incur serious load imbalance, as the subset of interest may not be evenly distributed within each block. An *optimized* partition method involves counting the total number of elements within the result bitset, and then dividing the dataset into blocks with equal number of 1-bits. This option has additional partitioning cost, but can clearly obtain much better load balance. We have used the optimized option because the data partitioning based on the result bitset (using fast bit-based operations in the memory) is much smaller than the disk I/O and the network transfer time. We also use multi-threads to perform both points segmenting and partitioning in parallel to further improve the efficiency, which can be divided into two stages: 1) The result bitset is logically divided into intervals with a fixed size, and then, all 1-bits within each interval are counted and segmented. 2) Merge operations are performed to group intervals into blocks based on the count of 1-bits, and segments are also grouped together between intervals. As an example, in Figure 6, we can see that the first stream will process the 8 elements in the first two rows, and another stream will process the left 9 elements in the following three rows. The entire operations in the first two steps are based on bitmap indices without touching the dataset.

After the point partitioning, one thread within each stream will be invoked to perform data read operations, with performance model based data subsetting method applied. Also, as one thread in each stream is responsible for data reads, another thread ($Thread_{00}$ in the first steam and $Thread_{10}$ in the second stream) can keep monitoring the *Sending Queue*. If the queue is not empty, it will extract the data chunk at the head of the queue and send it through the network using the TCP protocol. Chunks in different streams can be sent in parallel which makes better use of the bandwidth. Moreover, the network transfer process can be started immediately after the first data chunk is ready. This way, the data transfer and data reading overlap with each other, which further improves the efficiency.

4.3 Parallel Indexing

This subsection describes parallel bitmap indexing, which has at least two advantages: First, we are able to improve both index generation and index retrieval efficiency. Second, our performance model-based data subsetting method can be applied at a finer granularity.

During the index generation phase, instead of building and compressing bitmap indices over the entire dataset, we first logically partition one dataset into a collection of data blocks, and then initialize multiple processes and make each process build multi-level bitmap indices over a set of data blocks. This way, the index generation is performed in parallel and achieves a good speedup. A global metadata file, which keeps the relationship between dimension boundaries and bitmap indices of each block, is generated. It can be used to locate target index files during query processing. In the index operation phase, by checking the dimension and/or coordinate based query conditions, we are able to know how many blocks are involved in the current query. By looking up the global metadata, we are able to locate the index files of the corresponding data blocks. Afterwards, we can invoke multiple processes and perform indexing operations over different index files in parallel.

Besides the obvious advantages of this approach, another point to note is that in most cases, data elements within the subset are not evenly distributed among different blocks. For data blocks that do not contain any data subset element, parallel indexing can help us skip these blocks. For data blocks that contain a small percentage of elements, we can use *direct access* method to subset the data. For data blocks that involve a large percentage of data subset elements, we can use the *memory filter* method to subset the data. Hence, the performance model-based data subsetting method can be applied to each data block, instead of the entire dataset, which offers more subsetting flexibility and is able to improve the data reading efficiency.

5. EXPERIMENT RESULTS

In this section, we report results from a number of experiments conducted to evaluate *SDQuery DSI*. We designed the experiments with the following goals: (1) To compare the performance (query processing and data transfer time) of *SDQuery DSI* against GridFTP default *File DSI*, for queries involving a range of subsetting ratios, and show that despite indexing operations and possibly non-contiguous accesses, server-side data subsetting is able to improve data transfer efficiency (the same set of experiments are performed with three different network bandwidths). (2) To show that our performance model-based selection between direct access and memory filtering is effective (i.e., we can almost always choose the more efficient approach at runtime). (3) To measure how the parallel streaming with our partitioning approach is able to improve the data transfer efficiency. (4) To examine how the parallel indexing method improves the efficiency of indexing operations.

Because *SDQuery DSI* supports both NetCDF and HDF5 data formats, our experiments used two large and real datasets, one for each format. For NetCDF, we used the datasets generated by the Parallel Ocean Program (POP) [11]. POP is an ocean circulation model, and the execution we used has a grid resolution of approximately 10 km (horizontally), and vertically, it has a grid spacing of nearly 10 m near the surface, and reaching 250 m in the deep ocean. POP generates 1.4 GB data for each variable per time-slice, and each variable is modeled with three dimensions: longitude, latitude, and depth. The dataset we use here is *TEMP* with 100 timesteps. The size of the dataset is 140 GB. For HDF5, we used the datasets generated by Mediterranean Oceanic Data Base (MODB). MODB is generated from a simulation for a 34-layer space in the Mediterranean Sea. The dataset we use here is *salinity*. A sample data file available for downloading has 34 layers, 63 rows, and 167 columns. Because the real dataset was only of a small size, we extrapolated the original data by extending the time dimension, and created a dataset of size 105 GB for our experiments.

The majority of our experiments were conducted on a local cluster (the *RI* cluster), where every node has 8 cores 2.53 GHz Intel(R) Xeon(R) processors, with 12 GB RAM and 200 GB local disk space. Some of our experiments also used another cluster, from a supercomputing center (the *Glenn* cluster), where every node has 8 cores, 2.6 GHz AMD Opteron(TM) processors, with 64 GB RAM and 1.9 TB local disk space. We use three server-client pairs, to evaluate our approach with different bandwidths. The first situation involves transfers over a local area network (LAN) with 1 Gb/s bandwidth and 0.17 msec round trip time (RTT), the second situation involves an inter-cluster but intra-campus transfer at 200 Mb/s and 24 msec RTT, and the third situation involves a WAN transfer with 20 Mb/s average speed and 60 msec RTT.

5.1 Efficiency Comparison between SDQuery DSI and File DSI

In this experiment, we examine the performance advantages of *SDQuery DSI*, by comparing it against the default GridFTP implementation that simply transfers the entire file to the client-side, referred to as *File DSI*. The *Read & Transfer Time* that we report for *File DSI* includes both the data retrieval time (from the disk) and the network transfer time. *SDQuery DSI* execution time that we report can be divided into two parts: the *Query Processing Time* and the *Subseting and Transfer Time*. The former includes the time to parse the query, perform indexing operations to generate point ID set, and perform points segmenting and partitioning. The latter includes the data subset retrieval and network transfer time. Here, we have used the *Direct Access with segmentation* method for data subsetting. Optimizations based on performance model to speedup data retrieval will be emphasized in the next set of experiments. Two streams were used for both methods in the results we report.

Figure 7 compares the efficiency between the *SDQuery DSI* and the *File DSI* for the POP Dataset. Here we generated 2000 SQL queries based on scientists' real ocean analysis requirement, e.g., different *temperature* scopes within specific ocean areas and certain depths. We also divided queries into 6 categories, which include queries with subsetting percentage of $<$1%, 1%-10%, 10%-25%, 25%-50%, 50%-75%, and $>$75%, respectively. The execution time of the *File DSI* indicates the time to transfer the entire data file, as shown in the rightmost bar of each sub-figure. In Figure 7, each sub-figure corresponds to one network environment. In the left sub-figure, where we use 1 Gb/s network, we can see that when the data subsetting percentage is smaller than 50%, *SDQuery DSI* achieves better efficiency than *File DSI*, with speedups ranging between 1.26 and 9.41. Otherwise, *File DSI* achieves better efficiency. When the query is going to return a large fraction of the data, and the network bandwidth is very high, the reduction in data transfer time is offset by the query processing time. Particularly, the disk I/O now becomes a bigger constraint than the network, and retrieving a subset is not likely to be as efficient as retrieving the entire file. However, if we look at the center sub-figure, where we use 200 Mb/s network, we can see that our method achieves better efficiency than *File DSI* for all six categories, with speedups between 1.15 and 29.07. This is because the total execution time of a transfer request is now dominated by the network time. As shown in the right sub-figure, where the average network speed is 20 Mb/s network, our method achieves even better efficiency than *File DSI*, with speedup between 1.21 and 81.32. Moreover, if we compare the query processing time with data transfer time of *SDQuery DSI*, we can see that in different network environments, the query processing over bitmap indices has much smaller time cost than the disk I/O and network transfer in most cases (The only exception is to transfer a small amount of data (less than 10%) with high-speed (1Gb) network). Overall, considering that most data transfers occur over a wide area network where limited bandwidth is further shared among a number of transfers, we can expect a large improvement from our system. Even with a high bandwidth, our method is still useful if less than 50% of the original file needs to be transferred, as is indeed the case with the applications we described in Section 2.

Figure 8 compares the efficiency between *SDQuery DSI* and *File DSI* for the MODB (HDF5-based) dataset. We also generated 2000 queries for MODB dataset and divided them into 6 categories. The results are very similar. With 1 Gb/s network, our method achieves better efficiency when the subset percentage is smaller than 50% and the speedup ranges from 1.16 to 7.91. Our method achieves better efficiency for all subsetting percentages using lower bandwidths. The speedup using 200 Mb/s and 20 Mb/s can be as high as 31.15 and 74.34, respectively.

5.2 Effectiveness of the Performance Model

This subsection evaluates the effectiveness of the performance model-based *hybrid method*. As explained in Section 4.1, the hybrid method automatically chooses between memory filtering and direct access for any given query. Thus, to evaluate the effectiveness of this method, we compare the performance of the memory filtering and direct access methods, as well as note which one is picked by the hybrid method. For completeness, we include both direct access with segmentation and direct access without segmentation. Parameters of our performance model were obtained using 400 test queries, and another 2000 queries were used for validation.

Figure 9 compares performance of the method over two different datasets (POP and MODB) and two different execution environments (RI cluster and Glenn cluster). The X axis shows different subsetting percentages and the Y axis shows the execution time. To emphasize the difference among the methods, we only show the data subsetting time (i.e., do not include either the query processing or the network transfer time, which are identical for all methods). The left sub-figure shows the subsetting time using the POP dataset on the RI cluster. The *Direct Access (points)* method does not use segmentation, and we can see that it is very inefficient. With segmentation, i.e., *Direct Access (segments)*, we have greatly improved the efficiency. It turns out that the average segment length is 300.36 and the speedup compared with the approach without the segmentation method is between 1.64 and 3.93. The *Memory Filter* method achieves similar subsetting efficiency for all different queries. Compared with *Direct Access (segments)*, it achieves better efficiency when subsetting percentage is larger than 62%.

If we look at the use of the performance model, i.e., the *Hybrid Access* method, we can find that in most cases it makes the right choice between the two methods. The only exception is that from 62% to 70%, memory filtering has better efficiency but our model chooses direct access method. However, as we can see from the figure, the time difference within this subsetting range is quite small. In other words, the hybrid method either matches the best method, or is only very marginally (1%-2% at most) slower.

The middle sub-figure shows the subsetting time for the MODB dataset on the RI cluster. The direct access method is faster if the subsetting percentage is 42% or lower. The data subsetting efficiency using the direct access method on this dataset is worse than what we observed for the POP dataset. The reason is that HDF5 supports more complex storage structure and provides more powerful subsetting functionality, but it also incurs heavier overhead for each subsetting operation. Though our implementation does use advanced HDF5 functions that are able to read multiple points or hyperslabs together within one function call, still the overhead is larger than what we observed for NetCDF. Another reason is that for the MODB dataset, the average segment length is only 72.21. Thus, we incur more frequent I/O accesses.

Again, we can see that our performance model works well. It makes the correct prediction in most cases, with only exception being the subsetting percentage varied from 42% to 50%. However, as we observed earlier also, this is the range where the performance difference between the two methods is negligible. Thus, again we

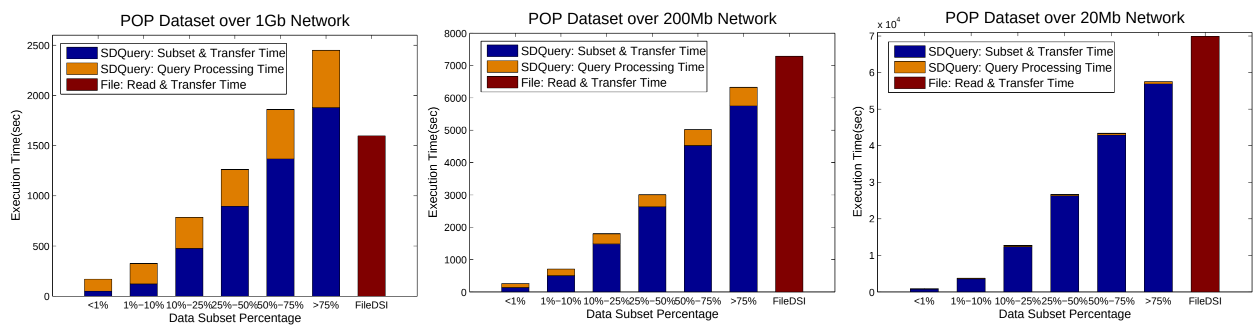


Figure 7: Efficiency Comparison between SDQuery DSI and File DSI for POP Dataset (Three Different Network Bandwidths)

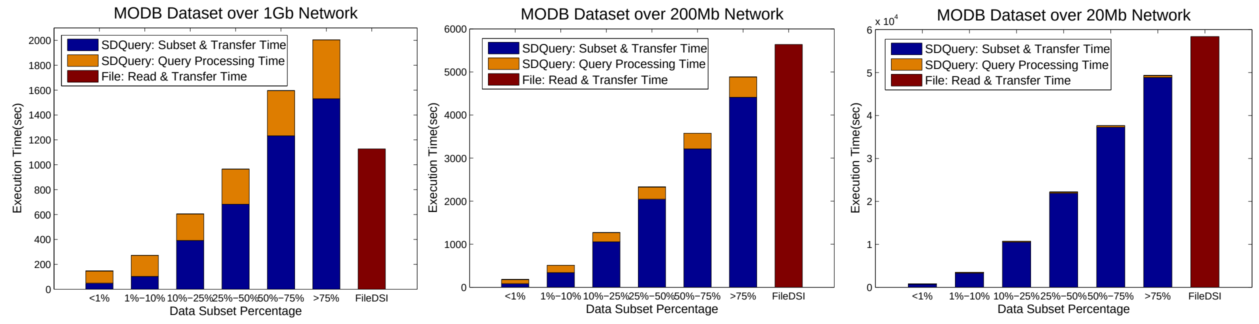


Figure 8: Efficiency Comparison between SDQuery DSI and File DSI for MODB Dataset (Three Different Network Bandwidths)

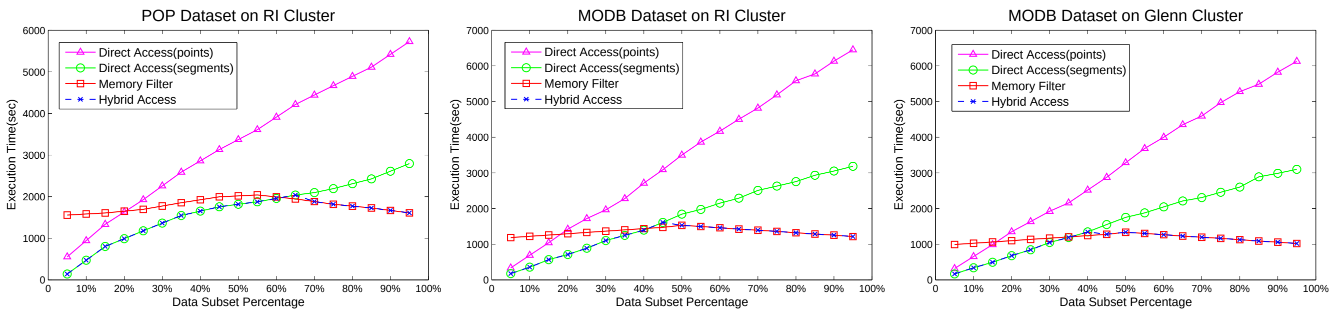


Figure 9: Data Retrieval Based on Performance Model: Different Datasets and Platforms

have shown that our performance model either chooses the best method, or results in performance that is only 1%-2% slower than the best method.

The right sub-figure shows the subsetting time using MODB dataset on the Glenn cluster. The switch point between the direct access and the memory filtering method is with a subsetting percentage of 36%. The memory filtering method becomes more efficient even for a smaller subsetting percentage because the Glenn cluster has faster disk transfer speed, though the seek times are the same. Again, the hybrid method makes the right choice in almost all cases.

To summarize, the relative performance of direct access and memory filtering methods depends not only on the subsetting percentage of the query, but also the data format, the dataset itself, and/or the execution environment. By obtaining parameters from a set of initial or training queries specific for the data format and the environment, we are able to tune our model, and almost always choose the best method for the given query.

5.3 Improving Efficiency with Parallel Streaming

The next experiment was designed to evaluate how data transfer efficiency can be improved with the help of parallel streaming. As we discussed in Section 4.2, parallel streaming not only uses parallel TCP streams (to make better usage of the bandwidth), but also enables parallel data retrieval, and overlap between data retrieval and transfer. The results we report here are from experiments with the MODB dataset only, as the results from the POP dataset are very similar. To highlight the benefits of the streaming method, we also implemented a *Non-overlapping* method. In this version, the data subset is retrieved (and possibly filtered), and the data transfer takes place only after the data subset is ready. Because of the memory limit for the *Non-overlapping* method, we use a 10.5 GB dataset here. The network speed is 200 Mb/s.

Figure 10 shows the performance of our method with different number of streams. We again generated 2000 queries and divided them into 4 categories. The Y axis shows the execution time, which

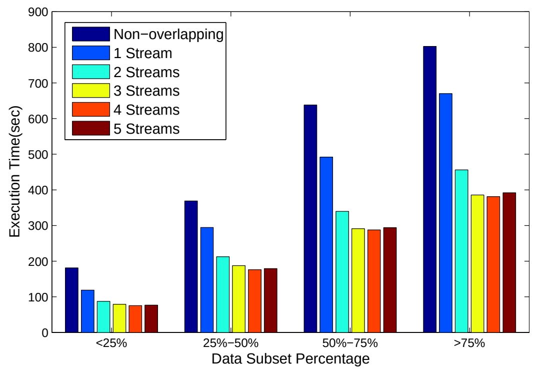


Figure 10: Efficiency Improvements with Parallel Streaming

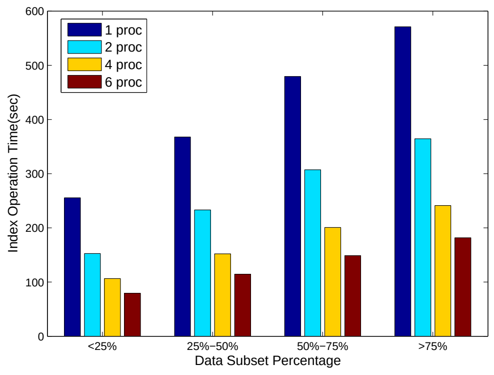


Figure 11: Indexing Time with Different Number of Processes

includes both data subsetting time and network transfer time. From the figure we can see that, for all different categories of queries, although the *1 stream* method does not apply any parallel optimization, it greatly improves the total efficiency because the data read time is effectively overlapped by the data transfer time. The speedup compared with *Non-overlapping* method is from 1.19 to 1.52, and the majority time is spent on data transfer. Moreover, parallel streaming can further improve the efficiency. Compared with 1 stream, the speedup using 2 streams for all categories ranges from 1.36 to 1.47, the speedup using 3 streams ranges from 1.50 to 1.73, the speedup using 4 streams ranges from 1.57 to 1.75, and the speedup using 5 streams ranges from 1.54 to 1.71. Compared with 1 stream, use of 2 streams is able to obtain over 40% more bandwidth. Use of 3 or 4 streams does result in more bandwidth, but the gains become smaller. After reaching a certain number of streams (5 streams in this case), the efficiency is not improved. This is because the bandwidth has been fully utilized (around 90% usage in this case), and increasing the stream number leads to more seek time during data retrieval. The appropriate number of parallel streams depends on both network bandwidth and RTT, and the range is from 2 to 16 in most cases.

5.4 Benefits of Parallel Indexing

This experiment was designed to show the performance advantages of parallel indexing. Although query processing time is much smaller compared with the data read and data transfer time, it can also be optimized to improve the overall efficiency, especially in the condition where the dataset is extremely large. In this experiment, we use the POP dataset of size 140 GB.

Figure 11 shows the scalability of parallel indexing with different number of processes. Here, we first logically divide the dataset into a collection of blocks. Each process takes care of index files that correspond to a separate set of data blocks. This way, parallel indexing not only decreases the index file loading time, but also reduces the time for bitwise indexing operations. From the figure, we can see that there is a good speedup as the number of processes increases. Compared with the use of only 1 process, the speedup on 2 processes varies from 1.55 to 1.69, the speedup on 4 processes varies from 2.38 to 2.43, and the speedup on 6 processes varies from 3.14 to 3.24.

6. RELATED WORK

Scientific data management has been widely studied. Here, we first compare our effort with the work that has been in the context of Globus GridFTP (DSI implementation), and then discuss closely related other efforts.

By default, Globus GridFTP has its own File DSI [8] to support data fetching on POSIX systems. Several other DSIs [12] have also been widely used, including the Storage Resource Broker (SRB) DSI, the High Performance Storage System (HPSS) DSI, and NeST DSI. MAPFS DSI is designed to support parallel data transfer on MAPFS system, which is a parallel and multi-agent file system for clusters [23]. Hans-Christian *et al.* [10] did an initial study on supporting HDF5 data subsetting and visualization using GridFTP. However, their tool only supports dimension-based queries. Compared to our effort, it did not support NetCDF, did not include support for value-based queries, and did not apply any of the optimizations we have included here.

Our work has several similarities with the NoDB approach [1] (previously also presented as automatic data virtualization [29]), where database-like operations are supported without loading data into a database. The distinct aspect of our work is application of this approach on NetCDF and HDF5 formats, and integration with a data movement protocol.

Several other tools have also been developed for scientific data management. OPeNDAP [7] provides data virtualization through a data access protocol and data representation. We had compared an earlier implementation of our approach [25, 28] (before its integration with GridFTP) against OPeNDAP and demonstrated that our approach has better efficiency, because it does not require data transformation to another format. In other efforts related to scientific data management, SciHadoop [5] enhances the map-reduce framework with a data partitioning method suitable for scientific datasets. Fastbit [30] and FastQuery [6] apply bitmap indexing and parallel indexing to support efficient value-based subsetting. Scientific Data Manager (SDM) [19] employs the Metadata Management System (MDMS) and provides a programming model to abstract low-level parallel I/O operations for complex scientific processing. NCO and its parallel implementation SWAMP [27] support data query and data computation over NetCDF datasets. Neither of them supports flexible data subsetting or includes integration with a data transfer protocol. In-situ analysis has been a topic of much investigation in recent years, with ADIOS project providing a mature implementation of this approach [15].

7. CONCLUSIONS

This paper has described *SDQuery DSI*, a GridFTP plug-in which supports flexible server-side data subsetting over HDF5 and NetCDF data formats. We have shown how a schema can be constructed using metadata from HDF5 and NetCDF formats, and structured queries can be issued to specify subsets of interest to the users. Another contribution of the work is in designing the system to be used by existing GridFTP servers without reinstallation. We have also provided several optimizations to help improve the performance. We have extensively evaluated our implementation. We show that subsetting at the server-side is effective, despite some overheads of indexing-related operations, with only exception being where a query outputs a large fraction of the original and the network bandwidth is also very high. We have evaluated each of our optimization methods and have demonstrated their effectiveness.

Acknowledgements

This work was partially supported by NSF grants IIS-0916196, CCF-1319420, and ACI-1339757.

8. REFERENCES

[1] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2012.

[2] W. E. Allcock, I. Foster, and R. Madduri. Reliable Data Transport: A Critical Service for the Grid. In *Proceedings of the Workshop on Building Service Based Grids*, 2004.

[3] G. Antoshenkov. Byte-aligned bitmap compression. In *Data Compression Conference (DCC)*, page 476. IEEE, 1995.

[4] Andrew Baranovski, Keith Beattie, Shishir Bharathi, Joshua Boverhof, John Bresnahan, Ann Chervenak, Ian Foster, Tim Freeman, Dan Gunter, Kate Keahey, Carl Kesselman, Rajkumar Kettimuthu, Nick Leroy, Michael Link, Miron Livny, Ravi Madduri, Gene Oleynik, Laura Pearlman, Robert Schuler, and Brian Tierney. Enabling petascale science: Data management, troubleshooting, and scalable science services. *Journal of Physics: Conference Series*, 125, 2008.

[5] J. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. Scihadoop: Array-based query processing in hadoop. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[6] Jerry Chou, Kesheng Wu, O Rubel, Mark Howison, Ji Qiang, Brian Austin, E Wes Bethel, Rob D Ryne, Arie Shoshani, et al. Parallel index and query for large scale data analysis. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2011.

[7] P. Cornillon, J. Gallagher, and T. Sgouros. Opendap: Accessing data in a distributed, heterogeneous environment. *Data Science Journal*, 2(0):164–174, 2003.

[8] Ian Foster and Carl Kesselman. The globus toolkit. *The grid: blueprint for a new computing infrastructure*, pages 259–278, 1999.

[9] Thomas J Hacker, Brian D Athey, and Brian Noble. The end-to-end performance effects of parallel tcp sockets on a lossy wide-area network. In *16th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 434–443. IEEE, 2002.

[10] Hans-Christian Hege, Andrei Hutanu, Ralf Kähler, André Merzky, Thomas Radke, Edward Seidel, and Brygg Ullmer. Progressive retrieval and hierarchical visualization of large remote data. *Scalable Computing: Practice and Experience*, 6(3), 2001.

[11] PW Jones, PH Worley, Y. Yoshida, JB White III, and J. Levesque. Practical performance portability in the parallel ocean program (pop). *Concurrency and Computation: Practice and Experience*, 17(10):1317–1327, 2005.

[12] Rajkumar Kettimuthu, Michael Link, John Bresnahan, and William Allcock. Globus data storage interface (dsi)–enabling easy access to grid datasets. In *First DIALOGUE Workshop: Applications-Driven Issues in Data Grids*, 2005.

[13] Rajkumar Kettimuthu, Alex Sim, Dan Gunter, Bill Allcock, Peer-Timo Bremer, John Bresnahan, Andrew Cherry, Lisa Childers, Eli Dart, Ian Foster, Kevin Harms, Jason Hick, Jason Lee, Michael Link, Jeff Long, Keith Miller, Vijaya Natarajan, Valerio Pascucci, Ken Raffenetti, David Ressman, Dean Williams, Loren Wilson, and Linda Winkler. Lessons learned from moving earth system grid data sets over a 20 gbps wide-area network. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010)*, Jun 2010.

[14] Ezra Kissel, D. Martin Swany, and Aaron Brown. Improving GridFTP performance using the Phoebus session layer. In *Proceedings of SC*, November 2009.

[15] Scott Klasky, Hasan Abbasi, Jeremy Logan, Manish Parashar, Karsten Schwan, Arie Shoshani, Matthew Wolf, Sean Ahern, Ilkay Altintas, Wes Bethel, et al. In situ data processing for extreme-scale computing. In *Scientific Discovery through Advanced Computing Program (SciDAC'11)*, 2011.

[16] T. Kosar and M. Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[17] Wantao Liu, Brian Tieman, Rajkumar Kettimuthu, and Ian Foster. A data transfer framework for large-scale science experiments. In *19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 717–724. ACM, 2010.

[18] D. Lu, Y. Qiao, P. A. Dinda, and F. E. Bustamante. Modeling and Taming Parallel TCP on Wide Area Networks. In *Proceedings of the 12th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005.

[19] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[20] S Narayanan, TJ Madden, AR Sandy, Raj Kettimuthu, and Michael Link. Gridftp based real-time data movement architecture for x-ray photon correlation spectroscopy at the advanced photon source. In *8th IEEE International Conference on E-Science (e-Science)*, pages 1–8. IEEE, 2012.

[21] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *ACM Sigmod Record*, volume 26, pages 38–49. ACM, 1997.

[22] Lili Qiu, Yin Zhang, and Srinivasan Keshav. On individual and aggregate tcp performance. In *Seventh International Conference on Network Protocols (ICNP)*, pages 203–212. IEEE, 1999.

[23] Alberto Sánchez, María S Pérez, Pierre Gueant, Jesús Montes, and Pilar Herrero. A parallel data storage interface to gridftp. In *Proceedings of the 2006 Confederated international conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE-Volume Part II*, pages 1203–1212. Springer, 2006.

[24] Michael Stonebraker, Jacek Becla, David Dewitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stan Zdonik. Requirements for science data bases and scidb. In *Conference on Innovative Data Systems Research (CIDR)*, January 2009.

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

[26] Yu Su, Gagan Agrawal, and Jonathan Woodring. Indexing and parallel query processing support for visualizing climate datasets. In *41th IEEE/ACM International Conference on Parallel Processing (ICPP)*, pages 249–258. IEEE, 2012.

[27] Daniel L Wang, Charles S Zender, and Stephen F Jenks. Clustered workflow execution of retargeted data analysis scripts. In *8th IEEE International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 449–458. IEEE, 2008.

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

[29] Li Weng, Gagan Agrawal, Umit Catalyurek, Tahsin Kurc, Sivaramakrishnan Narayanan, and Joel Saltz. An approach for automatic data virtualization. In *Proceedings of the Conference on High Performance Distributed Computing (HPDC)*, June 2004.

[30] K. Wu, W. Koegler, J. Chen, and A. Shoshani. Using bitmap index for interactive exploration of large datasets. In *15th International Conference on Scientific and Statistical Database Management*, pages 65–74. IEEE, 2003.

[31] K. Wu, E.J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *14th International Conference on Scientific and Statistical Database Management*, pages 99–108. IEEE, 2002.

[32] K. Wu, K. Stockinger, and A. Shoshani. Breaking the curse of cardinality on bitmap indexes. In *20th International Conference on Scientific and Statistical Database Management*, pages 348–365. Springer, 2008.

[33] Kesheng Wu, W. Koegler, J. Chen, and A. Shoshani. Using bitmap index for interactive exploration of large datasets. In *15th International Conference on Scientific and Statistical Database Management, 2003*, pages 65– 74. IEEE, July 2003.