Big Data Management Framework based on Virtualization and Bitmap Data Summarization

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Yu Su, B.S., M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

Dr. Gagan Agrawal, Advisor
Dr. Han-Wei Shen
Dr. Arnab Nandi
Abstract

In recent years, science has become increasingly data driven. Data collected from instruments and simulations is extremely valuable for a variety of scientific endeavors. The key challenge being faced by these efforts is that the dataset sizes continue to grow rapidly. With growing computational capabilities of parallel machines, temporal and spatial scales of simulations are becoming increasingly fine-grained. However, the 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.

Our overall goal is to provide a virtualization and bitmap based data management framework for “big data” applications. The challenges rise from five aspects. First, the “big data” problem leads to a strong requirement for efficient but light-weight server-side data subsetting and aggregation to decrease the data loading and transfer volume and help scientists find subsets of the data that is of interest to them. Different low-level data formats and access details should also be hidden from the scientists. The indexing technologies can also help further improve the data subsetting efficiency. Second, several wide-area data transfer protocols have improved the data analysis efficiency by supporting efficient data transfer over wide-area networks. However, the basic data transfer unit is still file level,
which can be as big as hundreds of GB or TB in size. Integrating flexible data management methods with wide area data transfer protocols becomes more than necessary. Third, data sampling, which focuses on selecting a small set of samples to represent the entire dataset, is able to greatly decrease the data processing volume and improve the efficiency. However, finding a sample with enough accuracy to preserve scientific data features is difficult, and estimating sampling accuracy is also time-consuming. Fourth, correlation analysis over multiple variables plays a very important role in scientific discovery. However, scanning through multiple variables for correlation calculation is extremely time-consuming, and how to support interactive correlation analysis over flexible data subsets become very challenging. Finally, because of the huge gap between data computing and storage, a big amount of time for data analysis is wasted on IO. In an in-situ environment, before the data is written to the disk, how to generate a smaller profile of the data to represent the original dataset and still support a variety of data analyses based on that is very difficult.

We first developed a light-weight data management tool, which allows flexible server-side subsetting and aggregation on scientific data. To hide the low-level data format details, our tool supports standard SQL queries for data analysis, and is able to automatically translate SQL queries into different low-level data accessing and filtering code. It also parallelizes selection and aggregation queries efficiently using novel approaches. Next, we designed a new indexing strategy and integrated it into the existing system, which further improves the subsetting efficiency for value-based queries. Our approach is based on bitmap indexing, but involves use of multi-level indices and careful partitioning, based on query profiles. We also show how our indexing support can be used for subsetting operations executed in parallel. Next, we developed SDQuery DSI, a GridFTP plug-in which combines our flexible server-side data subsetting work with efficient wide area data transfer
protocol. The GridFTP server is able to dynamically load this tool, and then subset the data before the transfer. Different subsetting cases are supported by our tool. A number of optimizations for improving indexing (parallel indexing), data subsetting (use of performance models) and data transfer (parallel streaming) are also applied.

Moveover, we addressed the data explosion problems by developing a novel sampling approach, and implementing it in a flexible system that supports server-side sampling and data subsetting. We have developed algorithms for using bitmap indices to sample datasets. We have also shown how only a small amount of additional metadata stored with bitmap can help estimate the loss of accuracy with a particular subsampling level. And this sampling method can be flexibly applied to any subset of the original data.

Correlation analysis across large-scale simulation datasets are time-consuming due to the huge data loading, filtering and correlation metrics calculation cost over multiple variables. To solve this problem, we designed a method which supports correlation analysis using bitmap indexing. This method can greatly improve the efficiency because our method has much smaller data loading time and subsetting time, and it also can calculate different correlation metrics more efficiently. We also show this method can further improve the efficiency in a parallel and distributed environment, and combine correlation analysis with sampling.

Finally, we developed an in-situ bitmap index generation and data analysis using bitmaps method. Each time after a dataset is simulated, we directly generate bitmaps in memory using the simulated data. Then we treat bitmaps as a data summarization and show that both online (time-steps selection) and offline analyses (correlation mining) can be supported purely using bitmaps. We also improve the bitmaps generation efficiency utilizing multi-core architectures.
In addition, all proposed methods have been extensively evaluated with multiple applications and different data inputs.
This is dedicated to the ones I love: my parents, my wife and my children.
Acknowledgments

During my five years of Ph.D study, there are so many people to whom I would like to give my sincerest thanks! Without their help and support, it would not have been possible to write this doctoral dissertation. To only some of them, it is possible to give particular mention here.

First of all, I would like to thank my advisor Dr. Gagan Agrawal for his advice, patience, and support on my Ph.D study. During the last five years, Dr. Gagan Agrawal not only guided me on the research of big data and high performance computing, but also help me develop a broad and strong foundation upon which a life pursuing computer science and technology will be built. His commitment not only taught me all the required qualities as a researcher but also inspired me as a person. The joy, passion, and enthusiasm he has for research were contagious and motivational for me in my future career. I would also like to thank the members of my thesis committee, Dr. Han-Wei Shen and Dr. Arnab Nandi. It is through their active interest an incisive and critical analysis that I have been able to complete my research.

Also, I would like to extend my sincerest thanks to my internship mentors, Dr. Jonathan Woodring from Los Alamos National Laboratory, and Dheeraj Singh from Facebook, Inc. They are not only my mentors but also my friends, and they provide me with tremendous help in both my research, my coding skill, and my life. At the same time, I would like to thank all collaborators in my past research, Dr. James Ahrens, Dr. Joanne Wendelberger,
Dr. Kary Myers, Dr. Han-Wei Shen, and Dr. Rajkumar Kettimuthu. Without their generous efforts, it is impossible for the projects in this dissertation to come true.

I would like to acknowledge the financial, academic, technical and all other kinds of support from National Science Foundation, the Ohio State University, and the Computer Science and Engineering Department, especially the head of our department, Dr. Xiaodong Zhang, and all administrative staffs our department, especially Kathryn M. Reeves and Lynn Lyons. I also would like to thank all my colleagues in my lab, Data-Intensive and High Performance Computing Research Group. Both former and present members of our group have given me invaluable help and support on research, career, and life, and they are: David Chiu, Qian Zhu, Fan Wang, Wenjing Ma, Vignesh Ravi, Wei Jiang, Tantan Liu, Tekin Bicer, Jiedan Zhu, Bin Ren, Xin Huo, Linchuan Chen, Yi Wang, Mehmet Can Kurt, Mucahid Kutlu, Jiaqi Liu, Sameh Abdulah, Roe Ebenstein, Gangyi Zhu, Jiankai Sun and Peng Jiang.

Special thanks are given to my wife, Xiaoning Guo, who has always been by my side. I am also thankful to my parents, my children and all the rest of my family in China and as well as all of the friends that I have encountered throughout my life. It is with their love, support, and encouragement, that this dissertation is in existence.
Vita

March 4th, 1984 ......................... Born - Lianyungang, China

2006 ................................. B.S. Software Engineering, Nanjing University, Nanjing, China

2009 ................................. M.S. Software Engineering, Peking University, Beijing, China

2013 ................................. M.S. Computer Science and Engineering, The Ohio State University, Columbus, USA

June, 2011 - September, 2011 .............. Research Intern, Los Alamos National Laboratory, Los Alamos, USA

May, 2012 - August, 2012 .................. Research Intern, Los Alamos National Laboratory, Los Alamos, USA

May, 2013 - August, 2013 .................. Research Intern, Los Alamos National Laboratory, Los Alamos, USA

May, 2014 - August, 2014 .................. Software Engineer Intern, Facebook, Menlo Park, USA

Publications

Research Publications


Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract .....................................................</td>
</tr>
<tr>
<td>Dedication ....................................................</td>
</tr>
<tr>
<td>Acknowledgments ...............................................</td>
</tr>
<tr>
<td>Vita ............................................................</td>
</tr>
<tr>
<td>List of Tables ................................................</td>
</tr>
<tr>
<td>List of Figures ...............................................</td>
</tr>
<tr>
<td>1. Introduction ...............................................</td>
</tr>
<tr>
<td>1.1 Science Big Data Management Needs .......................</td>
</tr>
<tr>
<td>1.1.1 Data Virtualization Issues ..........................</td>
</tr>
<tr>
<td>1.1.2 Need for Sampling .......................................</td>
</tr>
<tr>
<td>1.1.3 Need for Efficient Common Operations ..................</td>
</tr>
<tr>
<td>1.2 Overview of Scientific Data Formats ....................</td>
</tr>
<tr>
<td>1.2.1 NetCDF Data Format .....................................</td>
</tr>
<tr>
<td>1.2.2 HDF5 Data Format .......................................</td>
</tr>
<tr>
<td>1.3 Contributions ..............................................</td>
</tr>
<tr>
<td>1.3.1 Supporting User-Defined Subsetting and Aggregation over Scientific Datasets</td>
</tr>
<tr>
<td>1.3.2 Indexing and Parallel Query Processing Support for Visualizing Scientific Datasets</td>
</tr>
<tr>
<td>1.3.3 SDQuery DSI: Integrating Data Management Support with a Wide Area Data Transfer Protocol</td>
</tr>
<tr>
<td>1.3.4 Taming Massive Distributed Datasets: Data Sampling Using Bitmap Indices</td>
</tr>
</tbody>
</table>
1.3.5 Supporting Correlation Analysis on Scientific Datasets in Parallel and Distributed Settings .......................... 18
1.3.6 In-Situ Bitmaps Generation and Efficient Data Analysis based on Bitmaps .............................................. 20
1.4 Combining Everything ................................................. 22
1.5 Outline ................................................................. 24

2. Supporting User-Defined Subsetting and Aggregation over Parallel NetCDF Datasets .............................................. 26
   2.1 Motivation and System Overview .................................. 26
   2.1.1 Motivation Application ........................................... 26
   2.1.2 Overview of the Functionality .................................... 28
   2.2 Metadata Descriptor and Code Generation Strategy ............... 31
   2.3 Query Execution Methods .......................................... 33
   2.3.1 Pre-filtering and Post-filtering ................................... 33
   2.3.2 Aggregation Queries .............................................. 34
   2.3.3 Parallelization with PnetCDF ..................................... 36
   2.4 Experimental Results ............................................... 38
   2.4.1 Sequential Comparison with OPeNDAP ............................ 39
   2.4.2 Parallelization of Subsetting Queries ............................. 42
   2.4.3 Sequential and Parallel Performance of Aggregation Queries 44
   2.5 Related Work .......................................................... 46
   2.6 Summary .............................................................. 48

3. Indexing and Parallel Query Processing Support for Visualizing Climate Datasets 49
   3.1 Background: Bitmap Indexing ...................................... 49
   3.2 Motivation and System Overview .................................... 51
   3.2.1 Motivating Application .......................................... 51
   3.2.2 Overview of the Functionality .................................... 52
   3.3 Indexing and Parallel Query Processing .............................. 54
   3.3.1 Our Approach .................................................... 55
   3.3.2 Index Partitioning Strategy ....................................... 58
   3.3.3 Parallel Indexing ................................................ 62
   3.3.4 Index Generation and Retrieving Algorithms ..................... 64
   3.4 Experimental Results ................................................ 68
   3.4.1 Comparing Our Sequential Implementation with Filtering Method in ParaView .......................... 68
   3.4.2 Scalability with Different Number of Partitions .................. 71
   3.4.3 Parallel Comparison with FastQuery ............................. 73
   3.5 Related Work .......................................................... 75
3.6 Summary ................................................................. 76

4. SDQuery DSI: Integrating Data Management Support with a Wide Area Data Transfer Protocol ................................................................. 77
   4.1 Background: Globus GridFTP ........................................ 78
   4.2 Motivating Application ................................................ 79
   4.3 System Overview ......................................................... 81
      4.3.1 Overview of Desired Functionality ............................ 82
      4.3.2 Supporting Structured Queries: High-level Data Schema ..... 84
      4.3.3 System Architecture ............................................ 86
   4.4 System Optimizations .................................................... 90
      4.4.1 Performance Model-Based Optimized Data Subset Retrieval ... 90
      4.4.2 Parallel Streaming Data Transfer ............................... 96
      4.4.3 Parallel Indexing .................................................. 98
   4.5 Experimental Results .................................................... 99
      4.5.1 Efficiency Comparison between SDQuery DSI and File DSI ... 101
      4.5.2 Effectiveness of the Performance Model ....................... 104
      4.5.3 Improving Efficiency with Parallel Streaming ................. 106
      4.5.4 Benefits of Parallel Indexing .................................. 108
   4.6 Related Work .......................................................... 109
   4.7 Summary ............................................................... 110

5. Taming Massive Distributed Datasets: Data Sampling Using Bitmap Indices .... 111
   5.1 System Overview ....................................................... 112
   5.2 Sampling Using Bitmap Indices ....................................... 115
      5.2.1 Stratified Random Sampling over Bitvectors .................. 116
      5.2.2 Error Prediction .................................................. 119
      5.2.3 Sampling Only a Subset of Data ................................. 125
      5.2.4 Data Subsetting and Sampling over Multiple Attributes .... 125
      5.2.5 Multi-Resolution Sampling to Support Interactive Post-Analysis . 126
      5.2.6 Parallel Indexing and Sampling ................................. 128
   5.3 Experimental Results ................................................... 129
      5.3.1 Improving Efficiency of Distributed Data Analysis with Sampling 131
      5.3.2 Accuracy Comparison with Different Sampling Methods ....... 133
      5.3.3 Error Prediction Accuracy ....................................... 140
      5.3.4 Efficiency Comparison with Different Sampling Methods ....... 141
      5.3.5 Data Sampling over Data Subsets ............................... 144
      5.3.6 Speedup with Parallel Sampling ................................ 145
   5.4 Related Work .......................................................... 146
   5.5 Summary ............................................................... 148
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of Type 2 and Type 3 Selection Queries</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of Aggregation Queries</td>
</tr>
<tr>
<td>3.1</td>
<td>Major Parameters of Index Partition Strategy</td>
</tr>
<tr>
<td>4.1</td>
<td>Major Parameters of Auto-Tuning Model</td>
</tr>
<tr>
<td>5.1</td>
<td>Error Comparison of Signal-to-Noise Ratio</td>
</tr>
<tr>
<td>5.2</td>
<td>Error Comparison of Kolmogorov-Smirnov</td>
</tr>
</tbody>
</table>
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Combine Everything</td>
<td>23</td>
</tr>
<tr>
<td>2.1 Data Virtualization System Architecture</td>
<td>28</td>
</tr>
<tr>
<td>2.2 Metadata Generation</td>
<td>31</td>
</tr>
<tr>
<td>2.3 Prefiltering Module</td>
<td>35</td>
</tr>
<tr>
<td>2.4 Different Parallelism Levels</td>
<td>38</td>
</tr>
<tr>
<td>2.5 Performance Comparison for Subsetting</td>
<td>40</td>
</tr>
<tr>
<td>2.6 Speedup of Parallel Subsetting</td>
<td>43</td>
</tr>
<tr>
<td>2.7 Performance Comparison for Aggregations</td>
<td>45</td>
</tr>
<tr>
<td>3.1 An Example of Bitmap Indexing</td>
<td>50</td>
</tr>
<tr>
<td>3.2 Architecture of the Query System Using Bitmap Index</td>
<td>53</td>
</tr>
<tr>
<td>3.3 Index Generation Optimization</td>
<td>56</td>
</tr>
<tr>
<td>3.4 Index Retrieval Optimization</td>
<td>57</td>
</tr>
<tr>
<td>3.5 Parallel Index Architecture</td>
<td>62</td>
</tr>
<tr>
<td>3.6 Comparison between Indexing-Based and Filtering Methods (m1: one-level indexing support, m2: two-level with no indexing optimization, m3: second-phase optimization, and m4: both first-phase and second-phase optimization)</td>
<td>69</td>
</tr>
</tbody>
</table>
6.12 Distributed Analysis with Different Bandwidth ......................... 173

6.13 Efficiency and Accuracy of Correlation Analysis with Different Sampling Levels ......................................................... 174

6.14 Correlation Metrics Results over Data Subsets ......................... 177

7.1 In-Situ Bitmaps Generation .................................................. 186

7.2 Time-Steps Selection Using Full Data and Bitmaps .................... 191

7.3 Calculating Earth Movers Distance Using Bitmaps .................... 194

7.4 Calculating Conditional Entropy Using Bitmaps ....................... 196

7.5 Correlation Mining Using Bitmaps ......................................... 200

7.6 Select 25 Time Steps over 100 Using Different Simulations and Systems ................................................................. 205

7.7 Memory Cost Comparison ...................................................... 208

7.8 Efficiency Comparison between Shared Cores and Separated Cores Strategies ................................................................. 209

7.9 Scalability of Bitmaps Method ................................................. 211

7.10 Efficiency Improvement for Correlation Mining ....................... 213

7.11 Efficiency Comparison between Sampling and Indexing ........... 214

7.12 Accuracy Lost of In-Situ Sampling Method ............................ 215
Chapter 1: Introduction

In this chapter, first, we would like to introduce the motivation of our virtualization and bitmap index based data management work. Then, we will give the background knowledge in scientific data management area, including common used scientific data formats (NetCDF and HDF5). Finally, we will give an overview of our current work including server-side data subsetting, aggregation and sampling over scientific dataset, correlation data analyses using bitmap index and in-situ data analyses using bitmaps.

1.1 Science Big Data Management Needs

Many of the “big-data” challenges today are arising from increasing computing ability, as data collected from simulations has become extremely valuable for a variety of scientific endeavors. With growing computational capabilities of parallel machines, scientific simulations are being performed at finer spatial and temporal scales, leading to a data explosion. As a specific example, the Global Cloud-Resolving Model (GCRM) [61] currently has a grid-cell size of 4 km, and already produces 1 petabyte of data for a 10 day simulation. Future plans include simulations with a grid-cell size of 1 km, which will increase the data generation 64 fold.

The key challenge being faced by the data-intensive science efforts is that while the dataset sizes continue to grow rapidly, disk speeds and wide-area transfer bandwidths are
not keeping 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. Our overall research work has been motivated by three aspects: the scientific data virtualization issues, strong needs for efficient and accurate data sampling, the efficiency issues of common data processing operations, which will be described in the following three subsections. Our overall work focuses on building a platform which supports both individual data analyses (subsetting, aggregation, sampling) and correlation analyses over different scientific data formats. We also greatly improve the overall efficiency for different analyses (subsetting, sampling, correlation analyses, in-situ analyses) with the help of bitmap indexing.

1.1.1 Data Virtualization Issues

The “big data” problem described above leads to a strong requirement for reducing the volume of data that needs to be transferred over the wide area network, by providing support for user-defined data subsetting and aggregation at the server-side. Both data subsetting and aggregation, if supported at the server-side, can reduce the amount of data that needs to be transferred over the network and help scientists perform efficient analysis over subsets of interests. This issue can be further divided into three challenges:

First, it is difficult to find a standard but efficient approach to process different scientific data formats. Scientific datasets are typically stored in specialized formats like NetCDF, HDF5, or ADIOS, and the choice of the format depends upon the domain and even the specific application. For each data format, users have to understand its format details and learn how to use its APIs to access the data, which is time consuming. Clearly, one mechanism to reduce data volumes will be to use the existing relational database technology,
which provides user-defined data subsetting and aggregation through SQL and its variants. The multitude of reasons why scientific data management projects do not use relational databases are well documented [2, 77]. In summary, not only relational databases are too “heavy-weight” for read-only but extreme scale scientific data, they also do not support array-based data, or metadata that users will like to see. SciDB, a scientific database effort [38], has been widely used for scientific data analysis recently. Its key feature is support for arrays stored in parallel. However, it cannot operate on data stored in native format (like HDF5 or NetCDF). It is also not feasible to load data to a database, and solutions that operate on data stored in native format are needed. OPeNDAP [35] provides data virtualization through a data access protocol and data representation, and is extensively used in the Earth Systems Grid (ESG) project. It has only a very limited user-defined data subsetting and aggregation at the server-side. Hence, support various database-like operations while keeping the data in its original format is quite challenging.

Second, server-side data subsetting is resource costly and time-consuming. If we look at the current state-of-the-art to query or subset data, most subsetting operations are done through an in-memory interface, where all data are loaded at-once. Then either through filtering, i.e., applying data transformations, or selection interfaces, the data are subset. Clearly, with growing dataset sizes, loading the entire data and then filtering will be really time consuming and resource costly, which becomes a serious limitation. Therefore, we must look towards pushing data subsetting and querying operations in large-scale analysis tools to the I/O level. In the traditional database realm, different indexing technologies have been developed to solve this kind of issue, and bitmap indexing has been applied in scientific data management area for efficient value-based queries. However, the server-side queries usually include both value-based and dimension-based subsetting conditions. How
to effectively support queries containing both value-based and dimension-based subsetting conditions, and how to efficiently parallelize this kind of queries remain unanswered.

Third, because the scientific data is usually located in remote servers, the ability to provide efficient but reliable network data transfer between client and server becomes quite important. Several data transfer protocols such as Globus GridFTP, tried to solve the “big data” problem by providing parallel, streaming, reliable and restartable data transfers. However, the basic unit for data transfer is a single file which can still be huge in size. While enhancing and optimizing data transfer frameworks 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 and integrate the data subsetting with data transfer protocols, is clearly needed. However, currently there is no solution to combine flexible data subsetting with efficient data transfer protocols, and there are several important efficiency issues to be addressed during the integration, especially if bitmap indexing technology is applied here.

### 1.1.2 Need for Sampling

Finer granularity of simulation data offers both an opportunity and a challenge. On one hand, it can allow understanding of underlying phenomena and features in a way that would not be possible with coarser granularity. On the other hand, larger datasets are extremely difficult to store, manage, disseminate, analyze, and visualize. Neither the memory capacity of parallel machines, memory access speeds, nor disk bandwidths are increasing at the same rate as the computing power, contributing to the difficulty in storing, managing, and analyzing these datasets. Simulation data is often disseminated widely, through portals like the Earth System Grid (ESG) [15], and downloaded by researchers all over the world.
Such dissemination efforts are hampered by dataset size growth, as wide area data transfer bandwidths are growing at a much slower pace. Finally, while visualizing datasets, human perception is inherently limited relative to dataset sizes.

Facing such challenge, flexible server-side data sampling becomes a useful method to quickly reduce the data size and improve data analyses efficiency. However, there are three challenges for server-side data sampling:

1. Creating subsampled (lower-resolution) datasets from a high resolution simulation dataset, on demand and efficiently, while maintaining the characteristics of the original dataset.

2. Assessing the loss of quality (with respect to the key statistical measures) incurred with a particular level of resolution, on the given dataset, without having to take a pass through the entire high resolution dataset.

3. Providing the above functionality in a flexible system, which can support sampling at the server-side in response to requests from the client-side, and combine sampling with data subsetting.

1.1.3 Need for Efficient Common Operations

There is often a need to support common data processing operations over datasets generated by the scientific simulations. Such operations can be classified into two categories: *individual variable analysis* and *correlation analysis*. Individual variable analysis involves analysis over each variable or attribute independently, and can take the form of data measurement (e.g., mean, variance, histogram, QQPlot), data subsetting, data aggregation, data mining or visualization. Much of the existing work, especially in data visualization, has focused on individual variable analysis. More recently, several efforts [17, 115] have focused
on studying the relationship among multiple variables and making interesting scientific
discoveries based on such analysis. For example, correlation query is responsible for cal-
culating correlations among multiple variables within the specific data subsets, correlation
mining is used to automatically suggest data subsets with strong correlations to users, and
time step selection is responsible for finding the time steps with big variations between
each other. All of these methods are very useful in scientific analyses. However, the “big
data” problem of data movements being the constraint becomes even more severe for cor-
relation analysis, and currently no method is able to support efficient correlation analysis
over flexible subsets of the data.

If we look at the general correlation analysis process, each time it needs to load and
scan multiple large scale data blocks for calculation, which has a large memory cost and
is extremely time-consuming. Parallelizing such analysis is also hard, because of the pos-
sibility of large-scale data movement. Moreover, besides addressing the algorithmic chal-
genes, there is a need for a system that can offer a high-level interface for such analysis.
For example, in many cases, scientists may only be interested in performing correlation
analysis over (*value-based* and/or *dimension-based*) subsets of the data, and a structured
query interface is needed for such analysis. Because scientific datasets are stored in for-
mats like NetCDF or HDF5, and not in a database, support for such subsetting (especially,
value-based subsetting) is not (efficiently) available.

Moreover, in-situ data analysis also plays an important role in improving common data
operations efficiency. With the growing computing capacity, different simulations are able
to generate a big amount of data. However, because of the limitation of disk IO and net-
work speed, it takes a long time for these data to transfer from memory to disk. Driven
by this consideration, we can see that the only option for data-driven analysis of an engineering simulation on emerging (cost and power effective) systems involves: 1) Aggressive reduction of data soon after it is generated (so as to reduce the memory requirements for the next step), 2) Analyses performed in real-time over the reduced or summarized data, and possible further data reduction, 3) Long-term storage and/or movement of only the most critical and summarized data for future analyses, and 4) Aggressive analyses, visualization, and exploration, but using only the summarized data. The first two steps are referred to as in situ data reduction and data analysis, respectively.

Such in situ analysis, however, involves many open challenges. On one hand, memory and data movements considerations dictate that any summary structure be very compact, analyses be extremely memory efficient, and only a small fraction of the data generated be stored in the persistent memory or moved to a different device. On the other hand, it is also important that data analysis does not lose out on salient features – otherwise, the entire advantage of simulating the phenomenon at a high spatial and/or temporal granularity can be easily lost. Scientists often perform data reduction based on ad-hoc approaches, typically relying on their own insights into the simulation. However, in the process, they risk losing out on new insights into the phenomenon that can only be extracted by a more systematic analysis of data.

1.2 Overview of Scientific Data Formats

Scientific data is stored in different low-level formats, and APIs written in C, C++ or Java are designed to help access the data. In our work, we focused on NetCDF and HDF5 data formats, which are the two most popular data formats widely used in scientific data
management area to support efficient file reading and writing operations. This section gives an overview of these two data formats.

1.2.1 NetCDF Data Format

NetCDF (network Common Data Form) is a set of data formats and programming interfaces that provide an efficient way to store and retrieve array-oriented scientific data across multiple platforms. Physically, each NetCDF file is divided into two parts: header and data. The header contains the metadata, which describes the layout of the file. One important feature of NetCDF is that it is self-describing. With the information contained in the header, no external information is needed to determine the file contents. The data part stores actual values, and comprises a list of multidimensional arrays.

NetCDF data model contains three basic elements: dimensions, attributes, and variables. Each dimension has a name and a length, which is used to define the shape of variables. The size of the most significant dimension can also be defined as UNLIMITED, to support variables of growing size. Each attribute has a name and a corresponding value. Attributes are of two types: global attributes and variable attributes. The former are used to describe the properties of the dataset, whereas the latter are used to describe the properties of each variable. Each variable is a multidimensional array. The header of each NetCDF file describes each variable by its name, data type, dimensions, and attributes, whereas the data part stores the values. Variables can be divided into three different types: non-record variables, record variables and coordinate variables. Non-record variables are the ones with a fixed size, whereas the record variables are arrays with their most-significant dimension being UNLIMITED, i.e. their size can grow. A coordinate variable is a 1-D array with the
same name as its dimension. It is a variable, but it can be used as a dimension of other variables.

### 1.2.2 HDF5 Data Format

HDF5 is a widely used data model, library, and file format for storing and managing data. It supports a variety of data types, and is designed for flexible and efficient I/O and for high volume and complex data. HDF5 is portable and extensible, and HDF5 files are organized in a hierarchical structure, with two primary structures: groups and datasets. A group contains instances of zero or more groups or datasets, and a dataset is essentially a multidimensional array of data elements, where both of them can have associated metadata.

Additionally, HDF5 is designed to support any data type, so, besides a limited number of atomic datatypes, compound datatype is also supported in HDF5. A compound datatype may have an arbitrary number of data members in any order, and the data members may be of any datatype, including compound.

The HDF5 high-level APIs provide a set of functions built on top of the basic HDF5 Library. The purpose of the high-level API is two-fold: 1) to define functions that perform more complex operations in a single call, as compared to the basic HDF5 interface, and 2) to build sets of functions that provide standard entity definitions (like images or tables).

HDF5 *dimension scale* is one of important features here. An HDF5 dimension scale is an auxiliary dataset that is associated with a dimension of a primary dataset. It can serve as a coordinate system support for the primary dataset by constructing a mapping between values of dimension index and values of the dimensional scale dataset. A common example is a 2-dimensional array with spatial information, such as latitude and longitude, associated with it.
1.3 Contributions

This section introduces the contributions of our research, including six components.

1.3.1 Supporting User-Defined Subsetting and Aggregation over Scientific Datasets

Facing the data formatting challenges, we have developed a light-weight data management approach to enable support for complex and large low-level scientific datasets. Our approach provides a simple virtual view of the data, and allows users to specify subsets and aggregations of interest with such a virtual view. While our approach is general, this work reports an implementation of this approach specific to Parallel NetCDF. As stated above, NetCDF is used in a number of domains, which makes our tools quite widely usable. Our tools support SQL select and aggregate queries specified over the virtual (relational) view of the data. Besides supporting selection over dimensions, which are directly supported by NetCDF API also, we also support queries involving coordinate variables and those involving variable values. For this, we generate code for pre-filtering and post-filtering in our system. We also effectively parallelize selection and aggregation queries using novel algorithms.

Overall, the contributions of this work are as follows:

- First, we proposed a data virtualization method which supports standard SQL queries over low-level scientific data formats. The low-level data formats and data access details are totally hidden from users. A metadata is designed to help users understand the data and perform queries.

- Second, we supported multiple data selection types (dimension-based, coordinate-based and value-based) queries and aggregations.
• Third, we designed a parallelization algorithm which supports efficient data selection and aggregation in a parallel environment. We designed different partition methods for different types of queries to minimize the communication cost among data nodes.

1.3.2 Indexing and Parallel Query Processing Support for Visualizing Scientific Datasets

The “big data” issues arise in many areas, like data analytics, data mining, and visualization. Particularly in the context of visualization, large-scale, parallel visualization and analysis tools, such as VTK [91], ParaView [7] and VisIt [31], have been continuously enhanced by the scientific visualization community over the past 10 years. More recently, research efforts have turned towards large-scale, scientific data staging from the supercomputer to storage, such as ADIOS [74], GLEAN [111], and HDF5 [43]. In preparation of exascale supercomputing, the scientific visualization research community has also focused on in situ analysis, the coupling of analysis and simulation code to reduce the amount of computational time spent in I/O, found in tools such as ParaView Co-Processing [42] and VisIt Libsim [126].

One particular area of research has been in scientific data management, and techniques like FastBit [132] have come out of scientific visualization, analysis, and data management research. Querying, along with other methods like data mining, are able to help a scientist manage large-scale data, by subsetting large-scale data and isolating particular regions of interest in a large-scale dataset. Though there is clear case for use of advanced indexing methods, such as FastBit, the current practice in visualization is very different. If we look at the current state-of-the-art to query or subset data in large-scale visualization and analysis tools, like ParaView and VisIt, most subsetting operations are done through an in-memory interface, where all data are loaded at-once. Then either through filtering, i.e., applying
data transformations, or *selection* interfaces, the data are subset. The memory footprint for this operation is the total size of the data plus the size of the data after subsetting operations.

Clearly, with growing dataset sizes, this is a serious limitation. Therefore, we must look towards pushing data subsetting and querying operations in large-scale analysis tools to the I/O level, prior to loading all the data in memory, to cope with the memory limitations. This work focuses on developing indexing methods and supporting parallel query processing for (climate) datasets stored in scientific data formats. We have developed a novel indexing scheme, which can efficiently support queries on both dimensions and variable values. We also parallelize our method, carefully examining different ways in which the index and the query processing can be partitioned. Our implementation is specific to NetCDF, though the overall approach can be applied to data stored in any other array-based data format as well.

Overall, the contributions of this work are as follows:

- First, we showed that bitmap indexing can greatly improve the data analysis efficiency and decrease the memory usage, especially in data visualization area which generates a big amount of extra data for visualization purpose.

- Second, we proposed a two-phase optimizations (a partition strategy in index generation phase, logic bitwise operations in index retrieval phase) to improve the efficiency for data queries containing both value-based and dimension-based subsetting conditions.

- Third, we designed a parallel data query architecture to improve the data query efficiency for multi-variable dataset using parallel bitmap indexing.
1.3.3 SDQuery DSI: Integrating Data Management Support with a Wide Area Data Transfer Protocol

In many 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 [45] 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 [36] (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 [10, 63, 65, 73, 76, 60, 13] 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 work, 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 work 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, coordinates and values) are supported by our tool, which is made possible using existing metadata as well as bitmap indexing. Several optimizations (performance model, parallel streaming, parallel indexing) are also designed to improve the efficiency.

Overall, the contributions of this work are as follows:

• First, we combined flexible data subsetting with efficient data transfer by developing **SDQuery DSI** in GridFTP.

• Second, we used a performance model to automatically select between indexing-based retrieval of data segments and full retrieval followed by in-memory filtering to improve data reading efficiency.

• Third, we supported a *parallel streaming* technique, where different disk blocks are read concurrently and piped to different TCP/IP streams.
• Fourth, we incorporated parallel indexing to perform indexing operations for each sub-block concurrently.

1.3.4 Taming Massive Distributed Datasets: Data Sampling Using Bitmap Indices

Sampling for “big data” applications has been extensively studied. Broadly, different statistical sampling methods [33, 75, 113, 128] have been proposed to find a representative subset of the entire dataset. Some popular techniques include *simple random sampling*, where we select a certain percent of elements randomly out of original dataset, and *stratified random sampling*, where we first divide the dataset into strata and then perform random sampling within each stratum. The latter method maintains certain spatial properties of the original dataset. To compare the accuracy between the sampled dataset and the original dataset, different error metrics [67, 114, 55] have also been used.

However, as we argue below, the existing work does not meet all the requirements, especially in the context of growing dataset sizes and the need for data dissemination and analysis in a distributed environment.

*Sampling Accuracy:* Two factors are extremely important while creating samples of scientific datasets so as to facilitate accurate analysis. The first is *value distribution*, i.e., the value distribution of the sampled dataset should be as close to the original dataset as possible. The second is *spatial distribution*, i.e., the data accuracy should be maintained not only for the entire dataset but also for various spatial sub-blocks. Most of the sampling methods [85, 114] developed in the context of scientific data management are focused on the second factor, but ignore the first one. On the other hand, value distribution based sampling is well studied and has been proven to be a good method in the database area [51, 88].
These methods, however, are not developed for scientific datasets, and do not even consider spatial distribution. Consideration of both value distribution and spatial locality is necessary for scientific datasets, and unfortunately, none of the existing work has included both.

*Error Calculation without High Overheads:* After sampling, it is also important to know how accurately the current sample is able to represent the original dataset. Different error metrics, such as mean, variance, histogram\(^1\) and Q-Q plot\(^2\) are used as diagnostics of the accuracy. With increasing dataset sizes and the distributed nature of analysis, there are several challenges in applying these methods. In particular, when the goal is to find the smallest sample that can achieve a satisfactory accuracy, the traditional sampling process involves the following (possibly iterative) process: 1) sample generation, and 2) error metrics calculation. If the error is too high, repeat with a larger sample, starting from step 1. The entire process can be extremely time consuming, especially if one needs to iterate multiple times. In particular, with the current methods, there is no way to know in advance what may be the smallest sample size at which acceptable accuracy levels can be achieved.

*Flexible Data Analysis over Any Subset:* In many cases, users are only interested in data analysis or visualization over a subset of the data. For example, only certain timestamps may be of interest, and/or only a particular spatial subarea needs to be analyzed. Even if server-side subsetting is available, the resulting dataset size may be very large. Thus, the sampling method should be such that it can be applied to any specified subset. Unfortunately, existing sampling methods cannot support such flexible data subset sampling.

\(^1\)http://en.wikipedia.org/wiki/Histogram
\(^2\)http://en.wikipedia.org/wiki/Q-Q_plot
Data Sampling without Data Reorganization: Certain sampling methods, such as KDTree-based stratified sampling[128], have been shown to be effective for scientific datasets. However, before sampling can be performed, data reorganization is necessary. This imposes huge memory and disk I/O costs. Moreover, it is not possible to maintain multiple copies of a massive dataset, and sampling is not the only operation to be performed at server-side. After reorganization, other data features that are necessary for other tasks could be lost. Thus, we need sampling methods which operate while maintaining the data in the original format.

In this work, we address the above limitations of existing work by developing a novel sampling approach. We observe that to allow subsetting over scientific datasets, data repositories are likely to use an indexing technique [97]. Among these techniques, we see that bitmap indexing can not only effectively support subsetting over scientific datasets, but can also help create samples that preserve both value and spatial distributions over scientific datasets. We have developed algorithms for using bitmap indices to sample datasets. We have also shown how only a small amount of additional metadata stored with bitvectors can help assess loss of accuracy with a particular subsampling level, i.e., we do not need to take a pass over the entire sampled dataset to calculate accuracy based on these metrics.

Overall, the contributions of this work are as follows:

- First, we designed a bitmap index based sampling method, which preserves value distribution as well as spatial distribution of the original dataset and hence can achieve a much better sampling accuracy than other sampling methods.

- Second, we proposed an error prediction method based on bitmap index. With only small amount of metadata, our method can help assess loss of accuracy with a particular subsampling level efficiently.
• Third, we showed that our sampling method can be flexibly applied to a subset of the original dataset, which may be specified using a value-based and/or a dimension-based subsetting predicate.

• Fourth, for our sampling method, no data reorganization is needed, once bitmap indices have been generated.

1.3.5 Supporting Correlation Analysis on Scientific Datasets in Parallel and Distributed Settings

In our work, we propose a set of algorithms and a system for correlation analysis in parallel and distributed settings, starting from a high-level API, and with support for incremental and interactive analysis. From an algorithmic side, we present a series of methods for correlation analysis using bitmap indexing [32, 129, 130]. Bitmap indices, one of the popular indexing methods, preserve both the value distribution and the spatial locality of the data, and thus can be treated as a summary or profile of the original dataset, though much smaller in size. Moreover, because bitmap indices can help support basic database-like operations, e.g., data subsetting [32, 97], they do not have to be built exclusively for correlation analysis. Because of the data reduction associated with bitmap indices, the novel correlation calculation algorithms we have developed can incur much smaller network data transfer and memory accesses costs compared with the traditional method. We have designed two different indexing methods, which are dynamic and static indexing, to improve the efficiency. It turns out that bitmap indices or bitvectors also can help reduce the amount of communication during the computation of correlations in a parallel environment. We have developed two different partitioning methods, dimension-based partitioning and value-based partitioning, to perform parallel correlation analysis with bitvectors. These
methods are also extended to a distributed setting, where datasets corresponding to different variables may be stored at geographically distributed locations. Finally, with the help of bitmap indexing, we are able to generate accurate samples, which are then used to perform correlation analysis more efficiently. This allows us to trade some of the accuracy for improved response time. We have developed algorithms to use bitvectors for sampling and support correlation analysis based on these samples.

The algorithms we have developed have been incorporated in a flexible correlation analysis system, which also has several other desirable properties. First, correlation analysis is offered from a high-level API, where users can conveniently express dimension-based and value-based subsetting conditions (using an SQL-like syntax). These subsetting conditions are also supported on scientific datasets using bitmap indices. Moreover, we support incremental analysis. Users can add additional constraints on the top of the subset used for the last round of analysis. Finally, users can also perform an undo operation, which will allow them to build on top of not the most recent result, but an earlier result (and possibly further specialize on those subsets). With the help of bitmap indexing, we only need to keep track of bitvectors during the interactive querying process to support these features.

Overall, the contributions of this work are as follows:

- First, we showed that the correlation analysis using bitmap indexing can achieve a great speedup compared with the method based on original dataset.

- Second, we proposed two partitioned methods for parallel correlation analysis and compared the efficiency between each other.
• Third, we combined bitmap index based sampling with correlation analysis and showed that the efficiency of correlation analysis can be further improved with the help of sampling.

• Fourth, we incorporated our work in a flexible correlation analysis which supports interactive correlation analyses over subsets of the data.

1.3.6 In-Situ Bitmaps Generation and Efficient Data Analysis based on Bitmaps

In our work, we propose a novel method that utilizes bitmap index (bitmaps) as a representation of the data and showed that many kinds of analyses can be supported by bitmaps without requiring the original dataset. Each time after the data is simulated in memory, we generate bitmaps. Subsequently, a number of different analysis steps are performed on bitmaps. Then, instead of writing the data, we only write the bitmaps (much smaller in size) back to disk, which greatly improves the I/O efficiency.

Besides identifying bitmaps as an appropriate summary structure for scientific data, we have also demonstrated how to use bitmaps for both online (in-situ) analysis (e.g., time step selection which focuses on selecting a subset of time-steps containing more ‘important’ information out of simulated time-steps [116, 106, 69, 115]) and offline analysis (e.g., correlation mining which automatically finds data subsets with high correlations [17]) efficiently.

The following observations can be made with respect to bitmap index and their suitability as a summary structure for multi-dimension arrays. There is no binning (and thus no loss of precision) with respect to dimensional attributes, unlike almost any other method, such
as a histogram. This turns out to be a very important advantage, especially for any application where spatial precision is critical. The value distribution is also maintained (though this is also true for histograms). At the same time, because each point is represented by a single bit, compression is used, continuity of dimensional attributes is exploited, and due to the hardware support for bit operations, we can achieve space and time efficiency. Note that like other methods discussed above, there is binning with respect to value-based attributes, but it seems unavoidable when space efficiency is important.

The other advantages of bitmaps are as follows:

- Bitmaps are much smaller in size compared with the original dataset. In most of the cases, the size of bitmaps is less than 30% in size compared with the original dataset, which improves both data I/O and memory usage.

- Many kinds of analyses can be executed purely using bitmaps without touching the original dataset. In our previous work, we demonstrated that approximate data aggregation, data spatial join, correlation query, incomplete data analysis and subgroup discovery can be supported using bitmaps without touching the original dataset [3, 4, 98, 122, 123]. In this work, we will further show that time-steps selection and correlation mining can be supported using bitmaps.

- Bitmaps can be generated efficiently in an in-situ setting with acceleration using multiple cores. More importantly, we observe that newer architectures have a large number of cores, which can be used to generate bitmaps without very high overheads. Moreover, the cost of bitmaps generation is easily offset by reduction in data movement times.

Overall, the contributions of this work are as follows:
• First, we showed that in an in-situ environment, bitmap index can be generated in parallel efficiently utilizing multi-core architecture. We also designed an in-place bitmaps compression algorithm to save the memory cost of bitmaps generation.

• Second, we showed that bitmaps can be treated as a data summarization and support both online (time-steps selection) and offline analyses (correlation mining) without the original dataset.

• Third, we compared the efficiency of in-situ analyses with and without bitmaps and showed that although our method has extra bitmaps generation cost, we are still able to achieve better efficiency than full data method because our method has smaller time-steps selection and data IO time.

• Fourth, we compared in-situ sampling method with our method and showed that our method can achieve better accuracy and discussed the scenarios where our method can even achieve better speedup compared with in-situ sampling method.

1.4 Combining Everything

In this section, we combine all our work together and propose a data management framework to support more efficient scientific data analysis. Our data management framework contains two modules: SQL-based Data Virtualization module and Bitmap-based Data Summarization module, as shown in Figure 1.1.

SQL-based Data Virtualization module supports high-level SQL-like queries over different kinds of low-level data formats such as NetCDF and HDF5. From the scientists’ perspective, all they need to know is how to use SQL queries to specify their data sub-setting, aggregation, sampling or even correlation analysis requirements. And our module
can automatically transfer the high-level SQL queries into low-level data access languages written in C or C++. The program can automatically fetch the data subsets, perform different calculations and return the final results directly to the scientists. We also designed a metadata description to help the scientists understand the data and specify their queries.

*Bitmap-based Data Summarization* module treats bitmap index as a data summarization and supports different kinds of analysis only using bitmaps. Indexing technology, especially bitmap indexing have been widely used in database area to improve the data query efficiency. The major contribution of our work is that we find bitmap index keeps both value distribution and spatial locality of the scientific dataset. Hence, it can be treated as a summarization of the data with much smaller size. In the following chapters, we will demonstrate that many different kinds of analyses can be supported only using bitmaps.

We use these modules to support both data analysis and data sampling. Here we divide the data analysis into three categories: individual analysis, correlation analysis and in-situ analysis. The *SQL-based Data Virtualization* module supports individual analysis (data
subsetting and aggregation), correlation analysis (correlation query over data subsets), and data sampling (specify sampling percentage) using high-level SQL language. The Bitmap-based Data Summarization module supports correlation analysis, in-situ analysis and data sampling using bitmaps. Compared with the methods using the full data, our method is able to improve the data analysis efficiency, decrease the memory cost and IO or Network transfer volume. Both SQL-based Data Virtualization and Bitmap-based Data Summarization module support parallel data analysis and sampling.

1.5 Outline

The rest of the dissertation is organized as follows. In Chapter 2, we introduce our light-weight data management tool, which is to support flexible server-side subsetting and aggregation on scientific data. In Chapter 3, we extend our data management method with a novel indexing strategy. Our approach is based on bitmaps, but involves use of two-level bitmap indices and careful partitioning, based on query profiles. We also show how our indexing support can be used for subsetting operations executed in parallel. We present SDQuery DSI in Chapter 4, a GridFTP plug-in which supports flexible server-side data subsetting over HDF5 and NetCDF data formats. Several optimizations for improving bitmap indexing (parallel indexing), data subsetting (use of performance models) and data transfer (parallel streaming) are also applied. In Chapter 5, we introduce our flexible data sampling and subsetting approach. We have developed algorithms for using bitmap index to sample datasets. We also show how only a small amount of additional metadata stored with bitvectors can help assess loss of accuracy with a particular subsampling level. In Chapter 6, we introduce the correlation analysis using bitmap index approach. We show that different correlation metrics can be calculated efficiently using bitmaps. We propose
the in-situ bitmaps generation and data analyses using bitmaps method in Chapter 7. We show that bitmaps can be generated efficiently on different multi-core systems. It can also support different kinds of online and offline analyses. Then we discuss the potential future in Chapter 8 and conclude in Chapter 9.
Chapter 2: Supporting User-Defined Subsetting and Aggregation over Parallel NetCDF Datasets

In this chapter, we address the first “big data” issue, which is to support flexible server-side data management over different scientific data formats. Our approach provides a simple virtual view of the data, and allows users to specify subsets and aggregations of interest with such a virtual view. Besides supporting selection over dimensions, which are directly supported by NetCDF API also, we also support queries involving coordinate variables and those involving variable values. For this, we generate code for pre-filtering and post-filtering in our system. We also effectively parallelize selection and aggregation queries using novel algorithms.

2.1 Motivation and System Overview

In this section, we give an overview of our system.

2.1.1 Motivation Application

NetCDF dataset is stored in binary format. The advantage is that it supports compact storage and efficient processing, while the disadvantage is it makes the data specification and data processing much harder. Consider scientists who are interested in data subsetting. They need to have a detailed understanding of the data layout before being able to extract
the subset. In addition, they also have to get familiar with the NetCDF software libraries in order to write an application that can extract the subset. For each different subsetting task, the scientists may need to write a separate program.

The goal of our work is to alleviate these problems. The first premise of our work is that a *virtual relational table view* is supported over NetCDF dataset, and standard SQL queries with Select, From, and Where clauses on such a view, provides a very convenient yet powerful way to specify the data subset. The second premise is that data aggregation and group-by, using the same abstraction, can be applied to extract data summaries, which can minimize the data extraction and/or transfer costs. The third premise is that sequential subsetting of data can be prohibitively expensive with growing datasets, and therefore, parallelism should be used to accelerate data subsetting and aggregation queries.

A NetCDF dataset consists of a set of multi-dimensional arrays. These arrays typically involve spatial and/or temporal dimensions and coordinate variables. Scientists are usually interested in querying a subset of the data based on the either dimensions or coordinate variables. Hence, we can divide the queries into three categories: (1) Queries based on dimensions - The scientists can perform queries based on the dimensions of each variable, (2) Queries based on coordinate variables - Besides the dimension information, the scientists are also able to query data subsets based on spatial or temporal values, which are usually stored as coordinate variables, and (3) Queries based on variable value(s) - The scientists could also query the target variable value within a specific range. The goal of our system is to be able to support any combination of above three types of queries.
2.1.2 Overview of the Functionality

Figure 2.1 shows a high-level overview of our system. The basic function of our system is to automatically translate various SQL queries into data access code. In order to achieve this, a metadata descriptor containing semantic information of the NetCDF dataset should be generated. This metadata descriptor should be able to describe both the physical location and the logical layout of each NetCDF file. Based on the metadata, a code generation strategy is applied to generate data access code automatically. Detailed information about the metadata and the code generation strategy will be introduced in Section 2.2.
During the runtime of the system, several key modules work together to support efficient parallel queries for users. The SQL parser is responsible for parsing SQL queries and generating a parse tree. After users submit an SQL query, a scanner will take the stream of SQL characters as input and produce a sequence of tokens. Then, a parser will be used to process the stream of tokens and build a tree representation of corresponding SQL expressions. By evaluating this parse tree, key parameters within queries can be extracted and used in the data access code. Our parser is implemented with certain modifications of the parser from SQLite\(^3\), which is a light-weight open-source database engine. The metadata parser is responsible for generating metadata information for current query. Each time after the parse tree is generated, the metadata parser will take the dataset name and variable name list as input, look up the metadata collection, find corresponding physical and logic metadata, and load them into memory.

The Pre-filter module is responsible for generating a data subsetting request based on the parse tree and metadata information. It evaluates the parse tree, extract key elements such as variables, dimensions, and/or attributes from the tree and wrap them with the metadata information. Based on that, a query request, which contains specific parameters needed by the data access code is generated. In additional to this, the Pre-filter module also provides a mapping component to support relational constraint expressions. The detailed information about this module will be introduced later in Section 2.3.

The query partitioning module divides the query request further into several sub-requests. Each sub-request will be sent to a separate process to fetch a data block. Different partitioning ways tend to have a significant impact on the performance. In this work, an optimized partitioning method is applied to achieve the best efficiency. We first divide data partition

\(^3\)http://www.sqlite.org
into four levels: files, variables, blocks and sub-blocks. Because of the flexibility of SQL, one query can involve multiple data files in a dataset, multiple variables in a data file and multiple blocks in a variable. Our partition strategy will generate sub-requests based on both number of files, variables, blocks involved in current query and available processes in current running environment. In addition, the size of each data block can be huge. In order to achieve a better concurrency, we further divide one data block into several sub-blocks. Here, we consider two different situations. For a query without aggregation, the partitioning is based on maintaining continuity in data accesses. We partition the data block based on dimensions or coordinate variables which can lead to continuous access of each sub-block in file system. For a query that involves aggregation, the partitioning is performed with the goal of minimizing communication among different processes. We choose those dimensions or coordinate variables used in \textit{group by} clauses, which can make sure in most case, each aggregation group can be processed within one process. Detailed information about data aggregations will be introduced later in Section 2.3.

The data access code generated by the system first takes data sub-requests as the input, performs queries using the PnetCDF API and obtains the data results. Then, it may apply local data aggregation or post-filtering to the result, depending on different kinds of queries. The post-processing module is responsible for data collection and global data aggregation, as needed. The entire architecture is based on the MPI Master-Slave mode. Here, the master process is responsible for query analysis, which is, parsing SQL and generating data sub-query requests. The slave processes execute queries once they receive the request(s) from master process. Detailed information about parallel subsetting and aggregation will be introduced later in Section 2.3.
2.2 Metadata Descriptor and Code Generation Strategy

This section gives an overview of the metadata descriptor and code generation strategy.

In our system, a metadata descriptor that we define should be generated for each NetCDF dataset. This metadata descriptor is designed so that statically, it is able to expose the low-level layout of the data to the code generation module. At runtime, it can be served as an input to the metadata parser, helping finding both physical and logic metadata regarding to current query. In order to achieve these goals, the metadata descriptor includes: (1) Dataset Layout Descriptor, which contains description about variables, dimensions, attributes and relationship among them within each NetCDF file, and (2) Dataset Storage Descriptor, which describes physical locations where each data variable is resident.

Figure 2.2: Metadata Generation
Figure 2.2 shows the process of generating the metadata descriptor. Dataset Layout Descriptor contains a collection of data layout files. Each data layout file contains a logic description of a NetCDF data file. The logic layout information can be generated conveniently because it is included in the header part of each NetCDF file. In addition to the header part, the coordinate variable values also need to be extracted to support the pre-filtering module. With this information, the queries based on coordinate variables can be performed efficiently.

Dataset Storage Descriptor contains a collection of (variable, path) pairs. It is generated automatically using the Dataset Layout Descriptor and additional file system information. Since each data layout file maps strictly to one NetCDF data file, an address mapping is used here to improve the querying efficiency. By searching Dataset Storage Descriptor with the target variable, users are able to find not only target data layout file location, but also the corresponding NetCDF data file location.

Based on the metadata descriptor and the SQL query, a data access code is generated automatically. At compile-time, we parse the metadata and generate functions for data sub-setting and data aggregation. At runtime, these functions take a query request (containing key information from both the SQL query and metadata) as input and generate the data results. The advantage of this design is that expensive code generation associated with the metadata does not need to be carried out at runtime. The data access code consists of three parts: data subsetting, data aggregation and post-filtering. Data subsetting functions use the PnetCDF API to open and extract the data subset in parallel.
2.3 Query Execution Methods

This section describes our query execution methods, including pre-filtering and post-filtering for selection queries, processing of aggregation queries, and parallelization of both types of queries.

2.3.1 Pre-filtering and Post-filtering

From Section 2.1, we know that there can be three different types of subsetting queries: queries based on dimensions, coordinate variables and variable values. The standard NetCDF API only supports the first type of queries, which are those based on dimensions. To issue such a query, one needs to know the exact index range, which, in turn, requires a detailed understanding of the data layout. For most datasets, the most common type of query involves subsetting on spatial and/or temporal attributes, which is a query based on coordinate variable(s). Currently, for such queries, users have to first query the coordinate variables, manually determine each index range based on these values, and then they can perform a subsetting query with the index range. This requires complex programming with detailed knowledge of the data format and NetCDF APIs, and even the query processing time can be unnecessarily high.

In our system, we use pre-filtering combined with post-filtering to address this problem. Figure 2.3 shows a simple example of how components within pre-filtering module work. The input of this module is a parsed query. One of the components of this module, the Data Path Finder extracts the variables in the SELECT clause and dataset name in the FROM clause from the parsed query, looks up the Dataset Storage Information, and finds the data path of each variable. Another components, the Coordinate Variable Mapper, is able to identify the coordinate variables in the SELECT and the WHERE clause. Specifically, it
automatically builds a mapping between the coordinate variable and its index. From the figure, we can see that the coordinate variables \texttt{time} and \texttt{layers} have been updated to their dimensions after this step, while the dimension \texttt{cells} remains the same. The \textit{Layout Finder} looks up layout information for each variable in the Dataset Layout Descriptor. It finds the variable identifier and dimension identifier, builds connections between each variable and its corresponding dimensions, and adds a \textit{boundary} to each dimension. From the figure we can see that the variable \texttt{geopotential} is represented by its identifier after this step, and \texttt{time}, \texttt{cells}, \texttt{interfaces} are classified as its dimensions. The dimensions \texttt{time} and \texttt{cells} are already specified in the original query, while \texttt{interfaces} is extracted from the metadata. Each dimension is added with its boundary. The \textit{Query Request Generator} is responsible for generating different query requests. In this case, the generator just generates two separate data requests based on the number of variables, though it can handle more complicated queries as well.

For those queries based on variable values, the sub-query can only be applied after we get the data. In our solution, a \textit{post-filtering} component is generated automatically to handle this situation. This component will iterate on the data result and extract the data elements that meet the query constraints.

### 2.3.2 Aggregation Queries

Currently, NetCDF software libraries do not support data aggregation. However, in many user scenarios, this can be desirable functionality, especially while analyzing the data. For example, the scientists may want to calculate the average atmospheric pressure in the past 10 years within a certain range of the earth. Without data aggregation support, they have to write their own code for performing such an aggregation. Moreover, the data
volume before aggregation is much bigger than that after aggregation, so in a wide-area environment, current approach leads to much higher data transfer costs.

Our system addresses this problem by providing two-phase data aggregations, i.e. local aggregation and global aggregation. Our implementation works as follows. The master process will first take queries as input and identify if there is data aggregation involved. For aggregation queries, the query partitioning component will divide the query request into several sub-requests based on the dimensions used in aggregation and group by functions. Each slave process performs a sub-query and stores the result into a one-dimensional array.

Figure 2.3: Prefiltering Module
After that, a local data aggregation component will take the query data as input and generate aggregation result.

Algorithm 1 shows the pseudo-code of local data aggregation. To ensure consistency for arrays with different number of dimensions, any multi-dimensional array is processed as an one-dimensional array. Mapping is needed when classifying one-dimensional data result into specific groups. The input request encapsulates necessary information for data query and aggregation, such as ordered dimensions used in the group-by functions, aggregation type, and the dimension length. DataResults contains the query result. Lines 5-10 in Algorithm 1 maps each element in DataResults into indices of group-by dimensions in a multi-dimensional array. Lines 11-12 computes the target group ID based on these group-by indices. Line 14-18 performs specific type of aggregation on each group. In this algorithm, we only show code for sum aggregation. The code for other aggregation types such as count and avg is similar as sum.

After local aggregation, each slave process will send the partial aggregation result to the master process. The master process will perform a global data aggregation. Additional merge information is needed for global data aggregation. This information is generated by the query partition component before dispatching the query tasks.

### 2.3.3 Parallelization with PnetCDF

We exploit the parallelism at four levels: files, variables, target blocks and target sub-blocks. As shown in Figure 2.4, each time when a query request is generated, target data files and variables will first be found by looking at the data storage descriptor. In this example, all variables except var4 are involved in current query. For each variable, one or more target blocks need to be extracted from the dataset. To achieve a better query
Algorithm 1: performAggregation(request, dataResult)

1: allocate memory for groupSet based on request.groupSize
2: for each value result_i in dataResult do
3:   if result_i satisfies post-filtering conditions then
4:     for each dimension dim_j in request.groupDims do
5:       k ← request.dimNum - 1, temp ← i
6:       while k > dim_j do
7:         temp ← temp/request.dimLength_k
8:         k ← k - 1
9:     end while
10:    mappedID ← temp mod request.dimLength_k
11:    dimLength ← (j > 0)?getLength(dim_j-1) : 1
12:    groupID ← (groupID + mappedID) * dimLength
13: end for
14: if request.aggregateType = ”sum” then
15:   groupSet_{groupID} ← groupSet_{groupID} + result_i
16: else
17:   code to process other aggregation type
18: end if
19: end if
20: end for

Concurrency, we further divide each target block into sub-blocks. The number of sub-blocks depends on two factors. One is the number of processes provided by the running environment, and the other is number of files, variables and target blocks involved in current query. For example, target block in var0 will be divided into two sub-blocks, and target block in var1 will be divided into three sub-blocks. Each process will be assigned a sub-block query request and perform the query in parallel. With PnetCDF support, our parallel implementation allows a group of processes to collectively create one query instance over different sub-blocks of same data file, which will further improves the query efficiency. When dataset becomes larger, even the size of one sub-block can be huge. In this case, we setup a default chunk size, and each sub-block will be divided into chunks. Then each process will fetch data chunks serially. One thing to mention is, when we divide the blocks
into sub-blocks or divide sub-blocks into chunks, we need to follow the same partition rules, which is, to achieve good locality of data for subsetting queries, to minimize data transfer cost for aggregation queries.

![Diagram](image)

Figure 2.4: Different Parallelism Levels

### 2.4 Experimental Results

In this section, we report results from a number of experiments conducted to evaluate our system. We designed experiments with the following goals: 1) to compare the functionality and performance of our system with OPeNDAP, a scientific data management system widely used in environmental and oceanographic applications [35], 2) to evaluate the parallel scalability of our system, for which we measure the speedups obtained with different number of nodes and for different types of queries, and 3) to show how data aggregation queries reduce the data transfer cost.
The dataset used in our experiments was generated from GCRM, a global atmospheric circulation model. The specific dataset which was available for download was the one generated by a simulation with the grid-cell spacing of approximately 223 km\(^4\). The dataset includes large simulated atmospheric variables such as pressure, wind and others. Each variable is modeled by 40962 cells, 12 timeslots and other specific dimensions. While GCRM as well as various other simulations simulated data in thousands of even millions of timeslots which generated several orders of magnitude larger datasets, we did not have access to any such dataset. Therefore, in our experiments, we created larger datasets by generating random data with more timeslots. The dataset is built based on a geodesic grid\(^5\).

### 2.4.1 Sequential Comparison with OPeNDAP

OPeNDAP provides data virtualization through a data access protocol and data translation mechanism. However, there are several constraints for OPeNDAP. In order to hide the data formats and data processing details, OPeNDAP needs to translate the original data format into a standard OPeNDAP data format, which has high overheads. Moreover, relational constraint expressions (queries based on coordinate variables or variable values) are not supported on arrays and grids in OPeNDAP, and similarly, there is no support for data aggregations or group-by functions.

In the experiments we have conducted, we have compared our system with the DAP-enabled NetCDF component. OPeNDAP supports data retrieval in various formats, including ASCII, DODS binary objects, and so on. One component of OPeNDAP, the DAP-enabled NetCDF, specifically supports data translation between OPeNDAP standard model

\(^4\)http://climate.pnl.gov/data/

\(^5\)http://en.wikipedia.org/wiki/Geodesic_grid/
and NetCDF data model. Our system can execute data queries in parallel, whereas OPeNDAP can only execute queries sequentially. To make a fair comparison, we experimented with a sequential version of our system. The comparison between OPeNDAP and our sequential system was performed on a machine with Intel(R) Core(TM) i7-2600K 3.40 GHz CPU and 8 GB of memory.

From Section 2.1 we know that there are three different types of queries: queries based on dimensions (type 1), coordinate variables (type 2) and variable values (type 3). Since OPeNDAP, including the DAP-enabled NetCDF component, only supports queries based on dimensions for arrays and grids (i.e. the type 1 queries), we designed separate experiments for type 1 queries and type 2, 3 queries.

Our first experiment considered only type 1 queries, and compared OPeNDAP with our sequential system. We generated 50 type 1 queries, which included subsetting based on different dimensions. In Figure 2.5(a), the results were divided into 5 groups based on
different percent(<20%, 20%-40%, 40%-60%, 60%-80%, >80%) of data subset results. The size of each data variable is 4 GB. The first bar in each group shows the execution time of NetCDF data query function, without any additional data processing time. The second bar shows the total execution time of our sequential system. For type 1 query, since there is no coordinate mapping or post-filtering cost, the extra processing time is only due to SQL parsing and metadata query, which are both very small. From this figure, we can see that the total sequential processing time for type 1 query is not distinguishable from the baseline or the NetCDF data query function. The third bar shows the total execution time of OPeNDAP. We can see that OPeNDAP is consistently worse, and moreover, scales poorly as more data has to be output.

### Table 2.1: Example of Type 2 and Type 3 Selection Queries

<table>
<thead>
<tr>
<th>ID</th>
<th>SQL Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL1</td>
<td>SELECT pressure FROM GCRM WHERE time&lt;=(781850) AND pressure&gt;(0.8) AND pressure&lt;=(1.2);</td>
</tr>
<tr>
<td>SQL2</td>
<td>SELECT pressure FROM GCRM WHERE pressure&gt;(0.5) AND cells&lt;20481 AND time&lt;=(781710) AND layers&lt;(250);</td>
</tr>
<tr>
<td>SQL3</td>
<td>SELECT pressure FROM GCRM WHERE (time&gt;=(791000) OR time&lt;(781720)) AND (layers&lt;(250) OR layers&gt;(380));</td>
</tr>
<tr>
<td>SQL4</td>
<td>SELECT pressure, geopotential FROM GCRM WHERE time&lt;=(781790);</td>
</tr>
</tbody>
</table>

The next experiment involved type 2 and type 3 queries. Since DAP-enabled NetCDF does not support relational constraint expressions for *arrays* and *grids*, user client has to download the whole data from the server first and then write its own filter to generate data subset results. We implemented a client side filter for OPeNDAP, and then did a performance comparison between OPeNDAP and our system. In this experiment, we generated
50 queries which includes a combination of coordinate variables and variable values. Table 2.1 showed several examples of type 2 and type 3 queries. Each data variable is 4 GB. Figure 2.5(b) shows the execution time over different queries. The first bar in each group is the baseline of current experiment, which contains two subparts. The first part is execution time of NetCDF data query function, which is the same as previous experiment. The second part is post-filtering time. NetCDF API does not support subsetting over variable values. As a result, post-filtering is needed to take care of query over variable values. This time is proportional to the size of data results. The second bar is the execution time of our system. For type 2 and type 3 queries, more preprocessing time is used in metadata analysis such as coordinate variable mapping, but still we can see our system very small overhead when compared with the actual NetCDF query time. The third bar shows the execution time of OPeNDAP. Without pre-filter support, OPeNDAP has to scan and perform filtering on the entire dataset. Thus, both data query time and data filtering time is much larger than our system.

2.4.2 Parallelization of Subsetting Queries

As we have stated earlier, besides better performance and expressibility of our system over OPeNDAP, one of our advantages is the support for parallelization. In this subsection, we evaluate the performance improvements through parallelization of data subsetting operations. Our parallel implementation, with PnetCDF and MPI, is executed on an IBM Xeon Cluster, where every compute node has 8 cores Intel(R) Xeon(R) CPU 2.53GHz and 12 GB of memory.

The first experiment evaluates the performance of parallel subsetting with different data sizes. In this experiment, the input queries simply scan the whole data variable. The total
volume of the data which needs to be scanned is varied from 4 GB to 32 GB. The number of nodes used for parallel subsetting is varied from 1 to 16. The results are presented in Figure 2.6(a). We can see that our method showed a good speedup as number of nodes increases. The speedup on 4 processes is varied from 2.17 to 2.87. The speedup on 8 processes is varied from 4.06 to 5.54. The speedup on 16 processes is varied from 7.23 to 9.33.

The second experiment evaluates the performance of parallel subsetting with different queries. We generate 100 queries to cover different subquery conditions. In Figure 2.6(b), we divide the query results into 5 groups based on different percent of data result. The total volume of the data that needs to be scanned is 32 GB. The number of nodes used for parallel subsetting is varied from 1 to 16. We can see that with Query Partition module support, which guarantees each process can query a continuous data sub-block, the parallelization time stays proportional to the amount of data that is retrieved and processed. The speedup
on 4 processes varies from 2.20 to 2.92. The speedup on 8 processes varies from 3.95 to 4.21. The speedup on 16 processes varies from 7.25 to 7.74.

### 2.4.3 Sequential and Parallel Performance of Aggregation Queries

We now compare the performance of OPeNDAP with our parallel system for data aggregations. Recall that OPeNDAP does not support data aggregation. Thus, if users want to perform data aggregations, all the data has to be first downloaded from the server side, and a manually written code is needed to perform the aggregation at the client side. However, our system supports parallel server side data aggregation which has much smaller time cost for both data transfer and data aggregation.

<table>
<thead>
<tr>
<th>Type</th>
<th>SQL Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG1</td>
<td>SELECT avg(pressure) FROM GCRM;</td>
</tr>
<tr>
<td>AG2</td>
<td>SELECT count(pressure) FROM GCRM GROUP BY time HAVING time&lt;=791360;</td>
</tr>
<tr>
<td>AG3</td>
<td>SELECT sum(geopotential), interfaces FROM GCRM GROUP BY interfaces;</td>
</tr>
</tbody>
</table>

Table 2.2 shows aggregation types and examples used in our evaluation. We generated 60 aggregation queries and divided them into three types: AG1: query only includes aggregations, AG2: query includes both aggregations and `groupby` with `having` subsetting, and AG3: query includes both aggregations and `groupby` without `having` subsetting. To emphasize the efficiency improvement of parallel data aggregation, queries in this experiment don’t include any `where` expressions. Each data variable is 16 GB. The number of nodes is varied from 1 to 16.
One advantage of our parallel server side aggregation is the reduction in data transfer volume, which, in the likely case where the server and the client need to communicate over the wide area network, can itself result in large performance improvements. Figure 2.7(a) compares total data volume which needs to be transferred over the network between OPeNDAP and parallel system. We can see that with OPeNDAP, no matter what kind of aggregations, the whole dataset has to be transferred over the network before data aggregation. In our system, AG1 has the smallest data transfer volume since each node only generates one value as result. With having expression to subset aggregation groups, AG2 has smaller data transfer cost than AG3. In AG2, there is small data transfer difference among 4 nodes, 8 nodes and 16 nodes. This is because some queries had smaller number of groups than nodes. In this case, one data group can be processed by multiple nodes, which will bring extra data transfer cost. In AG3, the group number in current experiment is much larger than the nodes number, so the total data transfer volume is equal to number of data groups.
In general, for all three aggregation types, our method showed a much smaller data transfer volume than OPeNDAP.

In addition to the performance improvement in data transfer, our method also has much smaller data aggregation time. Figure 2.7(b) compares aggregation time between OPeNDAP and parallel system. We can see that our parallel aggregation method showed a good speedup compared with single aggregation. And it showed a good scalability as number of nodes increases. The speedup of 4 nodes is varied from 2.61 to 3.08. The speedup of 8 nodes is varied from 4.31 to 5.52. The speedup of 16 nodes is varied from 6.65 to 9.54. From the previous experiment, we can see that as number of nodes increases, the total data volume transferred over the network almost stays the same (remain the same for AG1 and AG3, increase only a few for AG2), which indicates that the overall efficiency improves as number of nodes increases.

2.5 Related Work

Scientific data management has been widely studied. Here, we initially focus on tools or approaches that can be used with scientific data formats like NetCDF, and then briefly overview most significant other work.

OPeNDAP [35] provides data virtualization through a data access protocol and data representation. We have extensively compared the performance and functionality of our tool against OPeNDAP. NCO and its parallel implementation SWAMP [117] support data subquery and data computation over NetCDF datasets. However, users have to use a very specialized scripting language to express their queries. This method is also very specific to NetCDF. SciHadoop [21] and SciMate [124] use the map-reduce framework and a data partitioning strategy to support parallel aggregations over scientific datasets. Our work uses a
higher level API for expressing queries, i.e. SQL. SAGA [120] uses a self-defined libraries to support different kinds of data aggregations over array-based data. FastQuery [32] applies parallel bitmap indexing to scientific datasets, which improves efficiency for query over variable values. Our work is more focused on supporting different types of queries over scientific dataset with a standard high-level interface. In the future, we will extend and incorporate indexing methods in our framework.

Each of the major data dissemination efforts today is building their own support for coping with dataset sizes and transfer requests. The Earth System Grid (ESG) has received funding from the previous two SciDAC competitions, and has built significant infrastructure for distributing climate data [16]. More recently, the CMS experiment at CERN has built PhEDEx\(^6\), LHC computing grid infrastructure has built ATLAS DDM[19], and cancer research community has built caGRID\(^7\). Some of the common topics addressed by these projects include file management, file replication, security, subscription, automatic updates to latest data, and bulk data movement. Storage Resource Broker (SRB) has been used in context of the nanoCMOS project [94]. Scientific Data Manager (SDM) employs the Metadata Management System (MDMS) [83] and provides a programming model to abstract low-level parallel I/O operations for complex scientific processing. More broadly related to data management in wide-area environments, other services include replica management services [30], metadata services [41], parallelism services [119, 81]. While such services can also improve efficiency of data transfers, the work is orthogonal to the functionality we have introduced.

\(^6\)cmseven.cern.ch/phedex
\(^7\)http://cagrid.org
Scientific data management has received much attention lately. Several groups have articulated the requirements in this area [2, 77]. Recently, there has been a significant interest in extending (relational) database technology to support the need of (extreme-scale) scientific data [105], leading to multiple Extreme Scale Databases Workshops (XLDB). The requirements arising have been summarized in a position paper by Stonebraker [95], and now, an initial version of SciDB is available. While our approach focuses on providing database like support to the users (i.e. user-defined subsetting and aggregations), the key difference in our approach is that the data is kept in its native form (e.g. flat-files, HDF5, or NetCDF), completely eliminating the need for loading the dataset in a database system. Our approach also provides a light-weight solution, which can be made available to different application domains more easily. The work presented in this chapter can be viewed as an implementation of the more general data virtualization approach introduced in our earlier work [125].

2.6 Summary

This chapter describes how we have implemented a light-weight data management approach for scientific datasets stored in NetCDF. Our tools support SQL select and aggregate queries specified over a virtual view of the data. We have extensively evaluated our implementation and compared its performance and functionality against OPeNDAP. We demonstrate that even for the queries that are directly supported in OPeNDAP, the sequential performance of our system is better. In addition, our system is capable of supporting a larger variety of queries, scaling performance by parallelizing queries, and reducing data transfers through server-side data aggregation.
Chapter 3: Indexing and Parallel Query Processing Support for Visualizing Climate Datasets

In the previous chapter, we have shown a light-weight data virtualization method to support data subsetting and aggregation over parallel NetCDF datasets. For dimension or coordinates-based query, we use prefiltering to load the exact data subset into memory. However, for value-based subsetting, we have to load the entire data blocks into memory and then perform postfiltering on each element, which can be both time consuming and memory costly. This issue becomes even worse when we combine the data subsetting with data visualization (e.g., Paraview). In this work, we further improves the data subsetting efficiency and memory usage by developing a novel bitmap indexing method, which efficiently supports queries on both dimensions and values. We also parallelize our method, carefully examining different ways in which the index and the query processing can be partitioned.

3.1 Background: Bitmap Indexing

Broadly, indexing provides an efficient way to support value-based queries and has been extensively researched and used in the context of relational databases. Bitmap indexing, which utilizes the fast bitwise operations supported by the computer hardware, has been
shown to be an efficient approach, and has been widely used in scientific data management [84, 132]. In particular, recent work has shown that bitmap indexing can help support efficient querying of scientific datasets stored in native formats [32, 97]. Recently several works [62, 66] demonstrate that in-situ bitmap indexing can greatly improve the index generation speed.

Figure 3.1 shows an example of a bitmap index. In this simple example, the dataset contains a total of 8 elements with 4 distinct values. The low-level bitmap indices contain 4 bitvectors, where each bitvector corresponds to one value. The number of bits within each bitvector is the same as 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 3.1, we can see two high-level indices are built based on value intervals.

This simple example only contains integer values. Bitmap indexing also has been shown to be an efficient method for floating-point values [131]. For such datasets, instead
of building a bitvector for each distinct value, we can first group a set of values together 
(*binning*) and build bitvectors for these bins. This way, the total number of bitvectors is 
kept at a manageable level.

From the example we can also see that the number of bits within each level of bitmap 
indices is $n \times m$, where $n$ is the total number of elements and $m$ is the total number 
of bitvectors. This can result in sizes even greater than the size of the original dataset, 
causing high time and space overheads for index creation, storage, and query processing. 
To solve this problem, *run-length compression* algorithms such as Byte-aligned Bitmap 
Code (BBC) [11] and Word-Aligned Hybrid (WAH) [130] have been developed to reduce 
the bitmap size. The main idea of these approaches is that for long sequences of 0s and 
1s within each bitvector, an encoding is used to count the number of continuous 0s or 1s. 
Such encoded counts are stored, requiring less space. Another property of the run-length 
compression methods is that it supports fast bitwise operations without decompressing the 
data.

### 3.2 Motivation and System Overview

This section provides an overview of our system. We will discuss the specific moti-
vating application, and then give an overview of a data virtualization system where our 
indexing approach is implemented and evaluated.

#### 3.2.1 Motivating Application

Our work is driven by the challenges arising in visualizing very large datasets, espe-
cially, visualizing large climate datasets stored in NetCDF, using ParaView. ParaView is an
open-source, multi-platform data analysis and visualization application\textsuperscript{8}. It can help users build visualization applications and analyze the data.

In most cases, scientists are only interested in a subset of data, such as temperature within a specific area or a given value range. However, the current method for subsetting scientific data in ParaView is based on the load plus filter approach. Irrespective of how large the data block is, ParaView has to first load the entire data block and only then it can apply a series of filters to subset the data. For example, a threshold filter is used to perform subsetting on data values, and an extract subset filter is used to perform data subsetting over spatial information. Multiple filters keep multiple data copies in the main memory, and new grids may be created when applying different filters together. Clearly this load plus filter mode adds a very high time and memory cost, and makes it infeasible to scale processing over very large datasets.

Instead of load plus filter mode, we want to find a method which can extract a subset of data from the storage system, similar to the way databases extract relevant data.

3.2.2 Overview of the Functionality

Recall that our goal is to provide a database like subsetting functionality on top of scientific (array-based) data formats. In order to achieve this goal, we build a virtual view over the NetCDF data format and support standard SQL queries with Select, From, and Where clauses over the dataset. NetCDF datasets can be accessed by NetCDF software libraries, which are usually written in C or Java. An automatic translation between standard SQL queries and NetCDF data access code in C or Java is supported by our system.

Furthermore, a NetCDF dataset consists of a set of multi-dimensional variables, which typically involve spatial and/or temporal dimensions or coordinate variables. Hence, we

\textsuperscript{8}http://www.paraview.org
can divide the queries into three categories: (1) Queries based on dimensions: scientists can perform queries based on the dimensions of each variable, (2) Queries based on coordinate variables: besides the dimension information, the scientists are also able to query data subsets based on spatial or temporal values, which are usually stored as coordinate variables, and (3) Queries based on variable value(s) - scientists can also query the target variable value within a specific range. Our system is able to process any above query type.

Figure 3.2 shows a high-level overview of our system. There are two main modules in the system, which are the Query Analysis Module and the Query Execution Module. We now describe these modules.
Query Analysis Module takes an SQL query and corresponding metadata as input and generates a specific format query request as output. The functionality of SQL Parser and Metadata Parser is same as we described in Figure 2.1 of Chapter 2. SQL Parser is responsible for parsing the SQL query and generating a parse tree, and Metadata Parser is responsible for generating data layout and data schema information for the current query. The Query Request Generator is responsible for generating a data subsetting request based on the parse tree and the metadata information. Based on this, a query request, which contains specific parameters needed by the data access code, is generated.

The second major module, the Query Execution Module, takes the query request as input, executes a data sub-query and visualizes the result. This step involves extensive use of multi-level bitmap indexing, which is the main focus of this work. Specifically, in this step, the module first checks if indices for current data files exist. If the indices do not exist, an Index Generation Function is invoked to generate indices for the data files. If the indices have been generated, an Index Retrieving Function will take the query request as input, perform indexing operations based on the dimensions, coordinates or variables subsetting conditions. Finally a data subset which satisfies both variable value and dimension/coordinates conditions is generated as the output and sent to VTK Pipeline for visualization.

### 3.3 Indexing and Parallel Query Processing

This section describes the indexing support and parallel query processing we perform. Though our implementation is in the context of the system we outlined earlier in Figure 3.2, the methods are applicable to subsetting array-based (scientific) datasets.
Now, let us consider the challenge of supporting a bitmap like indexing in our system, with the goal of enabling dimension, coordinate, and/or value based queries. Two previous solutions have addressed this need [96, 132]. Before describing these approaches and their limitations, we first define two variables. For any given dataset and query, let $\text{DimCount}$ denote the number of elements extracted by dimension subsetting conditions, and let $\text{VarCount}$ be the number of elements extracted by the value subsetting conditions.

In our earlier work [96, 121, 6], we extract the data subset based on dimensions, and then perform a post-filtering based on values. Post-filtering here implies simply scanning the data records linearly. This method is more biased towards dimension subsetting queries, i.e., if $\text{DimCount}$ is much smaller than $\text{VarCount}$. However, if $\text{DimCount}$ is much larger than $\text{VarCount}$, there is a large post-filtering cost. Another solution [132] that has been proposed is the opposite. It uses bitmap indexing to find data subset based on values, and then performs post-filtering based on dimensions. As we can expect, this approach is more biased towards value subsetting queries. It is inefficient if $\text{VarCount}$ is much larger than $\text{DimCount}$.

In view of this discussion, we want to find a solution which utilizes bitmap indexing to support query over variable values while at the same time maintaining efficient subsetting over dimensions (coordinates). We use a two-phase optimization (index generation phase and index retrieval phase) to solve this problem.

### 3.3.1 Our Approach

Our proposed approach is as follows. During the index generation phase, since compression over the entire data block can cost large I/O and bitwise operations, we partition one data block into a collection of sub-blocks. Then, we generate compressed multi-level
bitmap indices for each sub-block. This way, the sub-blocks that are not included in the dimension subsetting conditions are eliminated. We also develop an intelligent partitioning strategy to partition the data based on likely queries. Moreover, most scientific data applications today are running in a parallel environment. Hence, our approach also provides a parallel indexing framework to improve the query efficiency. Subsections 3.3.2 and 6.2.5 will describe the index partition strategy and parallel bitmap indexing, respectively, in details.

Figure 3.3 shows the process of index generation. From the figure, we can see that a multi-dimensional data block is first partitioned into a collection of sub-blocks based on a partition strategy. A global index file, which stores the dimension boundaries of each sub-block, is generated. It can be used to locate target index files during query processing. After that, a two-level bitmap indexing is built up for each sub-block. Second-level indices are built based on each distinct value, and first-level indices are built based on value intervals.
Each interval is formed by combining a group of second-level bitvectors. *Local metadata* is generated during the two-level indexing generation process and is used to locate target multi-level bitvectors during query processing. Detailed algorithm for multi-level indexing and local metadata generation will be described in Subsection 3.3.4.

The indexing approach proposed above needs to be used efficiently during the index retrieval (query processing) phase. Particularly, we improve the query efficiency by supporting bitwise operations between dimension subsetting conditions and value subsetting conditions. As we mentioned before, the previous solutions require a post-filtering either over the values or the dimensions. Such post-filtering is performed by iterating over elements one-by-one, resulting in low performance. Instead, our system utilizes fast bitwise
operations to improve the efficiency. For each query, we dynamically convert the dimension subsetting conditions into bitvectors, and logic operations are performed between the dimension bitvectors and the value bitvectors to generate the final result. The entire subsetting process is based on fast bitwise operations, which further improves the efficiency.

Figure 3.4 shows an example of how index retrieval component generates results based on the query request. From Figure 3.4, we can see that the query request is first divided into two subparts: $Dims$, which indicates dimension subsetting conditions and $Values$, which indicates value subsetting conditions. Our $Index Locator$, takes the $Dims$ and $Global Index$ file as input and finds the target index files that satisfy the dimension subsetting conditions. Through this way, those index files which do not qualify current query will be ignored by the following indexing operations. For each selected index file, the first-level and second-level bitvectors that satisfy the value subsetting conditions will be extracted. Logic operations will then be performed among them to generate value result bitvectors. For each query, the $Dim Bitvector Generator$ dynamically generates dimension bitvectors based on dimension subsetting conditions. Each 1 value in dimension bitvectors satisfies the dimension subsetting conditions. Then, logical $AND$ operations are performed between the dimension bitvectors and the value result bitvectors to generate a final point set. In this way, we can support all different query types while at the same time maintaining high efficiency.

### 3.3.2 Index Partitioning Strategy

As we stated before, the main problem for run-length compression algorithms is the ability to directly support subsetting over dimensions is lost after compression. One optimization we perform, as discussed above, is to first partition the data into sub-blocks and build indices based on each sub-block.
This, however, leads to the question of how blocks should be partitioned into sub-blocks. According to our experiment with 1.4 GB data block and 64 partitions, for specific kinds of queries, a “good” partitioning method can achieve a factor of 1.88 speedup over a “bad” partitioning method. Further, we observe that scientists also have a certain pattern or preference in their queries. Some scientists are likely to perform sub-queries over all dimensions, while some may only be interested in queries over a subset of dimensions. For example, some scientists only observe the ocean temperature changes in a given spatial area over different time-steps, while some others only compare the temperature among multiple areas within one time-slot. The basic idea of our index partitioning strategy is to partition the data block based on scientists’ query preferences, as observed from a certain amount of likely queries.

We use the following model. Table 3.1 lists all parameters in the model.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>Participation rate of data elements</td>
</tr>
<tr>
<td>$E$</td>
<td>Number of elements in index operation</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of sub-blocks in index operation</td>
</tr>
<tr>
<td>$S$</td>
<td>Size of each sub-block</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of dimensions</td>
</tr>
<tr>
<td>$D = D_1 \times D_2 \times \ldots \times D_n$</td>
<td>Total number of elements in a variable, $D_i$ is the length of $i$th dimension</td>
</tr>
<tr>
<td>$P = {p_1, p_2, \ldots, p_n}$</td>
<td>Total number of sub-blocks, $p_i$ is number of sub-blocks in $i$th dimension</td>
</tr>
<tr>
<td>$Q = {q_1, q_2, \ldots, q_n}$</td>
<td>Number of elements specified in a query, $q_i$ is the query length of $i$th dimension</td>
</tr>
<tr>
<td>$C$</td>
<td>A constant value</td>
</tr>
</tbody>
</table>
We define $\alpha$, which is equal to the number of elements involved in indexing divided by total data size, as the participation rate. The $\alpha$ value varies from 0 to 1. In the best case, all elements involved in indexing operations are exactly the same as those resulting from dimension subsetting conditions specified in the query. In the worst situation, on the other hand, $\alpha$ is always equal to 1, which means all elements have to be involved in indexing operation. Now, we can state the problem as:

$$\text{Minimize } \alpha = E = \frac{N \times S}{D}$$

where $N = \prod_{i=1}^{n} \left( \frac{q_i \times p_i}{D_i} + C' \right)$, $S = \frac{D}{P} = \prod_{i=1}^{n} \left( \frac{D_i}{p_i} \right)$

(3.1)

In Equation 3.1, $\alpha$ is proportional to $E$ and inversely proportional to $D$. Since $D$ is a constant value for a specific dataset, $\alpha$ is only impacted by $E$. $E$, in turn, equals $N$ multiplied by $S$, where $N$ indicates the total number of sub-blocks involved in the current query. It is equal to product of the number of sub-blocks specified by each dimension. Number of sub-blocks specified by each dimension is equal to total number of sub-blocks of the dimension $p_i$ multiplied by the dimension subsetting rate, which is $q_i/D_i$. Identifier $C'$ only has two values, 0 and 1. It is equal to 0 if and only if all elements within the selected sub-blocks are exactly the same as those specified by the dimension subsetting conditions. In our model, we further assume $C'$ is equal to 0 if and only if the entire dimension range is involved in the subsetting. The variable $S$ indicates each block’s size, which is equal to data size divided by number of partitions.

Based on Equation 3.1, three principles can be used to guide the data partitioning process.

**Rule 1:** The identifier $E$ is proportional to the dimension subsetting percentage specified in queries, and inversely proportional to the number of partitions. Formally:

$$E = \prod_{i=1}^{n} \left( \left( \frac{q_i \times p_i}{D_i} + C' \right) \times \frac{D_i}{p_i} \right) = \prod_{i=1}^{n} \left( q_i + C' \times \frac{D_i}{p_i} \right)$$

(3.2)
In Equation 3.2, we multiply $N$ with $S$ and generate a detailed formula description of $E$. In this formula, $C'$ is equal to 1 and $D_i$ is constant. From the formula we can see the value of $E$ decreases as $q_i$ decreases and $p_i$ increases. In theory, therefore, our first rule suggests that we should choose as many data partitions as possible. But in a real environment, this can also generate a huge number of small index files, which creates I/O problem. Furthermore, the compression rate becomes worse when index file becomes so small. As a result, the number of partitions chosen also depends on the data size and available resources in a parallel environment.

**Rule 2:** Suppose we have chosen the number of partitions, $P$, as per the constraints above. We still have the problem of deciding how these partitions should be chosen. If the target queries only include one or a subset of dimensions, the partitioning should also be based on these dimensions. Formally:

$$E' = \left( \prod_{i=1}^{k-1} D_i \right) \times \left( q_k + \frac{D_k}{p_k} \right) \times \left( \prod_{i=k+1}^{n} D_i \right)$$

(3.3)

Equation 3.3 shows the calculation of $E$ when only one dimension, $k$, is involved in the queries. For all the other dimensions, by default the entire ranges of them have to be selected, which makes $C'$ equal to 0 and $q_i$ equal to $D_i$, since, for each dataset, $D_i$ is a constant value. Identifier $E'$ is only impacted by $q_k$ and $p_k$. The larger the number of partitions we choose over the $k^{th}$ dimension, the smaller $E'$ is. This expression can also be generalized to the situation where a subset of dimensions are involved.

**Rule 3:** If the total number of partitions $P$ is a constant, in the general cases where the subsetting rate over each dimension has a similar probability, the data blocks should be partitioned with equal preference over each dimension. The equation below shows the
calculation for this situation.

\[ E'' = D \times \prod_{i=1}^{n} \left( \frac{q_i}{D_i} + \frac{1}{p_i} \right) \rightarrow \prod_{i=1}^{n} \left( C'' + \frac{1}{p_i} \right) \] (3.4)

We first changed the form of \( E \) by extracting data size \( D \) from the multiplications. \( q_i/D_i \) indicates the subsetting rate over the \( i^{th} \) dimension. If all dimensions have similar subsetting probability, we can assume that \( q_i/D_i \) is constant for dimensions in each query and use \( C'' \) to replace it. After that, we can get a new formula which only includes one variable \( p_i \). We know that the product of all \( p_i \) is equal to \( P \), which is a constant. Hence, according to Lagrange multipliers, \( E'' \) can achieve a minimal value if and only if the values chosen for \( p_i \) are the same.

3.3.3 Parallel Indexing

Nowadays scientific data is increasing in a rapid speed and parallel data processing has been a necessary approach for data analysis and data visualization. With the indexing partition strategy which generates a collection of standalone indices, our index generation
method can be easily applied to a parallel environment. Different parallel levels can be first defined for each NetCDF dataset, and we can choose a parallel level based on the dataset and available number of processes. If the number of processes is sufficiently large, we can build up one index file based on each sub-block. Then, each process can be responsible for one index file. This way, different processes can finish each sub-query independently.

Figure 3.5 shows the architecture of our parallel indexing. Here we define three parallel levels: files, blocks (variables), and sub-blocks. As shown in the figure, one NetCDF dataset includes multiple data files, one data file includes multiple data blocks, and one data block can be partitioned into multiple sub-blocks. Before index generation, we perform index partitioning. According to the rule 3 we listed earlier, by default each variable is partitioned with equal preference over each dimension. Users are also able to specify their preferred dimensions which are most likely to appear in their queries. In such cases, the second rule can also be applied. Different variables can also be partitioned using different rules.

In this figure, we assume all variables are two-dimensional. We can see that according to the partition description specified, $Var0$, $Var2$ and $Var4$ are partitioned based on both $X$ and $Y$ dimensions. Since query over the $X$ dimension is preferred for $Var1$, $Var1$ is partitioned only based on the $X$ dimension. Since query over the $Y$ dimension is preferred for $Var3$, it is partitioned only based on the $Y$ dimension. After the partitioning, each process will build multi-level bitmap indices over each sub-block. During query processing, we can select those sub-blocks which satisfy the dimension subsetting conditions. A global metadata file is maintained to help system locate target data sub-blocks during the query
processing. Hence, through this parallel indexing framework, even for queries which include multiple data files and variables, we can achieve a good efficiency for all different query types.

Two partition methods are supported in our system, physical partitioning and logical partitioning. Physical partitioning refers to the case when we generate new data sub-blocks through data partitioning, while logical partition implies that we only logically divide the data into sub-blocks and build bitmap indexing based on that without any new data sub-block generation. The advantage of physical partitioning method is that we have much faster data query time after indexing. In comparison, the advantage of the logical partitioning method is that we save time and space for new data generation. For scientific datasets where the number of read operations is much larger than the write operations, physical partitioning is preferred to achieve a better data query efficiency if the environment allows so.

### 3.3.4 Index Generation and Retrieving Algorithms

This subsection describes the detailed algorithms for multi-level bitmap indices generation and retrieval.

One challenge during the index generation process is that the cardinality of most variables in NetCDF dataset is high. Most variables in the NetCDF dataset contain float value, in which case, the number of bitvectors is almost equal to the number of elements in the data block. Simply building up bitvectors based on each value will have very high storage costs. A precision control over data values is necessary to decrease the cardinality. During index generation, we first compute the number of distinct values for the sub-block under the current precision level. Then, we choose a lower precision level if the current cardinality of
data is still too high. We repeat this process until the cardinality is smaller than a threshold value.

### Algorithm 2: Generate_Index

```plaintext
1: DataArray ← FetchData(FileName, VarName, Sid);
2: ValNum ← ControlPrecision(DataArray);
3: allocate space for L2BitVectors based on ValNum;
4: L2ValueIDMap ← GenerateValIDMap(DataArray, L2BitVectors);
5: for each value data[i] in DataArray do
   6:   ValueIDMap ← L2ValueIDMap.find(data[i]);
   7:   VectorID ← ValueIDMap.V id;
   8:   L2BitVectors[VectorID][i] ← 1;
   9: end for
10: BinNum ← ValNum/BIN_SIZE + 1;
11: allocate space for L1BitVectors based on BinNum;
12: for each value i in BinNum do
   13:   for each value j in BIN_SIZE do
   14:     L2V id ← i * BIN_SIZE + j;
   15:     if L2V id < ValNum then
   16:       L1BitVectors[i] ← OR(L1BitVectors[i], L2BitVectors[L2V id]);
   17: end if
   18: end for
19: L1BinboundArray[i].low ← ValueIDMap[i * BIN_SIZE].val;
20: L1BinboundArray[i].up ← ValueIDMap[i * BIN_SIZE + j - 1].val;
21: end for
22: CL1BitVectors ← CompressL1Vectors(L1BitVectors);
23: CL2BitVectors ← CompressL2Vectors(L2BitVectors);
24: WriteCBitVectors(CL1BitVectors, CL2BitVectors);
25: WriteMetadata(L1BinboundArray, L2ValueIDMap);
```

Algorithm 5 shows the method for index generation. Line 1 fetches the target data sub-block based on file name, variable name and sub-block ID. Lines 2 to 9 show the process of the second-level indices generation. In the line 2, the function \textit{ControlPrecision} chooses the right precision level for current the sub-block and returns the number of distinct values in the data array under the current precision level. Each value corresponds to one second-level bitvector. In the line 3, \textit{L2BitVectors} is initialized with size \textit{ValNum} and used to store
all second-level bitvectors. In line 4, the function GenerateValIDMap generates a collection of (key,pos) pairs. Each key indicates a distinct value of the data array, and pos indicates the id of a bitvector in L2BitVectors. We can classify each element in data array into its bitvector by looking at this map. Lines 5 to 9 involve an iteration over the data array. For each value, we look up the L2ValueIDMap and map the current value to target second-level vector. Then, we set the corresponding bit in that bitvector to 1. After this step, all second-level bitvectors have been generated, with the right bit set to 1. Lines 10 to 21 generate the first-level indices based on second-level indices. After all indices are generated, lines 22 to 24 compress both the first-level and the second-level indices and write the compressed indices to the index files. Line 25 writes L1BinBoundArray and L2ValIDMap into index metadata files. Both L1BinBoundArray and L2ValIDMap are used as metadata to help the system locate indices for a query.

During the query processing, first-level indices are first used to find a general value range. For the boundary values which first-level indices cannot cover, the second-level indices are looked up to finish the query. An OR operation is performed among all value bitvectors. Dimension subsetting conditions are dynamically converted into a bitvector, and then an AND operation is performed between dimension bitvector and the value result bitvector to generate the result point set.

Algorithm 3 summarizes the method for index retrieval within a data sub-block. The input ValueInfo stores variable subsetting conditions such as TEMP > 0, and DimInfo stores dimension subsetting conditions such as t.on > 10. The goal of this function is to generate a bitvector which satisfies all conditions specified in ValueInfo and DimInfo. In the line 1, the function QueryL1IndexMetadata reads the index metadata, and finds first-level index ids which satisfy current value subsetting conditions. In the line 2, the function
Algorithm 3: Retrieve Index($ValueInfo, DimInfo$)

1: $L1IndexIDs \leftarrow \text{QueryL1IndexMetadata}(ValueInfo)$;
2: $L1CompressedVectors \leftarrow \text{QueryL1IndexFile}(ValueInfo, L1IndexIDs)$;
3: for each cvector in $L1CompressedVectors$ do
4: \hspace{1em} $ResultVector \leftarrow \text{OR}(ResultVector, cvector)$;
5: end for
6: $isL2Needed \leftarrow \text{CheckL1IndexBounds}(ValueInfo, L1IndexIDs)$;
7: if $isL2Needed = \text{TRUE}$ then
8: \hspace{1em} $L2IndexIDs \leftarrow \text{QueryL2IndexMetadata}(ValueInfo)$;
9: \hspace{1em} $L2CompressedVectors \leftarrow \text{QueryL2IndexFile}(ValueInfo, L2IndexIDs)$;
10: \hspace{1em} for each cvector in $L2CompressedVectors$ do
11: \hspace{2em} $ResultVector \leftarrow \text{OR}(ResultVector, cvector)$;
12: end for
13: end if
14: if $DimInfo \neq \text{NULL}$ then
15: \hspace{1em} $DimBitVector \leftarrow \text{GenerateDimBitVector}(DimInfo)$;
16: \hspace{1em} $ResultVector \leftarrow \text{AND}(ResultVector, DimBitVector)$;
17: end if

$\text{QueryL1IndexFile}$ looks up the first-level index file and finds all compressed bitvectors based on indexing ids. Lines 3 to 5 perform OR operation for all $L1CompressedVectors$ and store the result into $ResultVector$. Line 6 checks if first-level indices are able to cover all value subsetting conditions. For most of the queries, the second-level indices are needed if the value subsetting conditions locate in the middle of bins. Lines 8 to 9 find the second-level compressed bitvectors for the rest of the values. Lines 10 to 12 perform logic OR operation for all $L2CompressedVectors$ and store the result into $ResultVector$. The input $DimInfo$ stores the dimension subsetting information. Line 15 generates a bitvector based on $DimInfo$. For each bit within the bitvector, if the id satisfies the dimension subsetting conditions, it is set to 1, otherwise, it is set to 0. Line 16 performs the AND operation between value bitvector and dimension bitvector. Finally a result bitvector is returned, and those bits set to 1 satisfy the query conditions.
3.4 Experimental Results

In this section, we report results from a number of experiments conducted to evaluate our indexing approach. We designed experiments with the following goals: 1) to compare the performance of our implementation, which uses our novel indexing approach, with the filtering method currently used in ParaView, 2) to show the scalability of our parallel query processing systems as the number of processes is increased, and 3) to compare the performance of our parallel indexing method with FastQuery [32], a recently proposed method.

The dataset used in the experiment is generated by Los Alamos National Laboratory’s Parallel Ocean Program (POP)[56]. POP is an ocean circulation model, and the execution we used has a grid resolution of approximately 10 km (horizontally). Vertically, it has a grid spacing on the order of 10 m near the surface, and reaches 250 m in the deep ocean. The data is stored in NetCDF format. A 33.6 GB dataset which includes 4 variables is used in our experiments. Each data variable is split into a collection of data files based on the time-steps. Within each file, the variables are modeled with three dimensions, longitude, latitude and depth. In our experiment, each variable has 6 time-steps and the data size of each time-step is 1.4 GB. Our experiment is executed on an IBM Xeon Cluster, where every compute node has 8 cores Intel(R) Xeon(R) CPU 2.53GHz and 12 GB of memory.

3.4.1 Comparing Our Sequential Implementation with Filtering Method in ParaView

The current method for subsetting scientific data such as HDF5 and NetCDF in ParaView is based on load plus filter mode. ParaView has to first load the entire dataset and then it can apply a series of filters to generate sub-visualization results. In this experiment,
we want to show that our indexing based query processing has both better query efficiency and smaller memory cost.

![Figure 3.6: Comparison between Indexing-Based and Filtering Methods (\textit{m1}: one-level indexing support, \textit{m2}: two-level with no indexing optimization, \textit{m3}: second-phase optimization, and \textit{m4}: both first-phase and second-phase optimization)](image)

Figures 3.6(a) and 3.6(b) show the execution time and memory cost, respectively, between the filtering method and different indexing methods. \textit{Index \textit{m1}} indicates subsetting with only one-level bitmap indexing support. \textit{Index \textit{m2}, \textit{m3}, and \textit{m4}} indicate bitmap based subsetting with two-level indexing support, where \textit{m2} has no indexing optimization, \textit{m3} has the second-phase optimization, and \textit{m4} has both first-phase and second phase optimizations. The input is 400 queries which include subsetting over dimensions, coordinates, and variables. The results are divided into 5 groups based on different percent (<20%, 20%-40%, 40%-60%, 60%-80%, >80%) of data returned as results.

From Figure 3.6(a) we can see that the time cost of the filtering method remains almost the same across different query groups. This is because the entire data block always needs
to be loaded into memory, and only then a post-filtering operation is applied to check if each element within the data block satisfies query conditions (both dimensions and values). For the indexing method, the execution time includes both indices retrieval time and data query time. We can see that the execution time with indexing support is proportional to the query selectivity, i.e. the percentage of data returned as the result of the query. As shown in the figure, compared with filtering method, \( m1 \) shows a better performance only when subset percentage is less than 20%. The problem is that when the cardinality of the data block is high, one-level bitmap method involves a huge I/O and logical operation time. Two-level indexing significantly improves the efficiency by binning a range of values into one bitvector. From the figure we can see that \( m2, m3 \) and \( m4 \) show a much better performance than \( m1 \). The difference between \( m2 \) and \( m3 \) is that \( m2 \) uses a post-filtering over the indexing results, while \( m3 \) uses the second-phase optimization to dynamically generate dimension bitvectors. The post-filtering method has to check if each element in the indexing results satisfies the dimension subsetting conditions, which is time-consuming. From the figure, we can see \( m3 \) has a better performance compared with \( m2 \), and its relative performance becomes better as data subsetting percentage increases. The difference between \( m3 \) and \( m4 \) is that \( m3 \) builds indices based on the entire data block, whereas \( m4 \) builds indices based on a group of sub-blocks. When the data subsetting percentage is less that 20%, the speedup of \( m4 \) over \( m3 \) is a factor of 2.97. However, this relative speedup decreases as the data subset percentage increases. This is because more sub-blocks have to be involved in the indexing operations, and the number of total indexing operations is larger than that of \( m3 \). When the data subsetting percentage is more than 80%, \( m4 \) is slower than \( m3 \). However,
as we mentioned before, another significant advantage of $m4$ is that it can be easily parallelized. Compared with the filtering method, the performance of $m4$ is better when the data subsetting percentage is less than 70%. The speedup ranges between 1 and 11.17.

Besides the execution time, from Figure 3.6(b) we can also see that memory cost for the indexing method is much smaller than that of the filtering method. Memory is a significant consideration as dataset sizes continue to grow. The memory required for the filtering method is the sum of the data block size and the size of the results after data subsetting. (In the Figure, the dashed line indicates the size of one data block). Thus, when the data subsetting percentage is larger than 80%, the memory cost for filtering is over 10 GB. For the indexing methods, the memory cost includes the size of the results and the size of the indexing structures. In our experiments, the size of first-level indices is 40.7% of data block size, and size of second-level indices is only 4.2% of data block size. Compared with $m1$ which only use one-level indexing, $m2$, $m3$ and $m4$ clearly save memory when the percentage of data subset becomes larger. The memory requirements for $m2$, $m3$ and $m4$ is similar. Although $m4$ generates a larger number of index files, after compression, the size of each index file becomes smaller.

### 3.4.2 Scalability with Different Number of Partitions

This subsection describes how parallel indexing and query processing speeds up the subsetting operations. We compared the execution time and the participation rate $\alpha$ with different number of processes. The total data size is 8.4 GB. The number of processes is varied from 6 to 96. We use 100 queries and the results are divided into 5 groups based on the fraction of data reported as results. The variable $TEMP$ with 6 time-steps is used in the queries.
The first experiment evaluates the performance of parallel indexing with different number of processes. In this experiment, to emphasize the scalability of parallel indexing, we chose the same number of data partitions as processes, with each process takes care of one partition. The results are presented in Figure 3.7(a). We can see that our method shows a good scalability as number of processes increases. Compared with 6 processors, the relative speedup with 24 processors varies from 2.15 to 2.87, the relative speedup on 48 processors varies from 3.98 to 6.09, and the relative speedup on 96 processors varies from 8.37 to 12.20.

Besides good scalability, another important goal of parallel query processing is that through data partitioning, the data elements that do not satisfy the dimension subsetting conditions are not involved in the indexing operations. As stated in Section 3.3, the participation rate $\alpha$ is an important indicator to reflect parallel indexing results. Figure 3.7(b) compares $\alpha$ rate with different number of processes. The upper dashed line shows the
worst-case situation in which there is no data partitioning and $\alpha$ is always 1, i.e., all elements in the data block have to be included in the bitwise indexing operations. The lower dashed line shows the ideal situation, where the number of elements involved in the indexing operation is exactly the same as the elements that match the dimension subsetting conditions. We can see that the $\alpha$ rate increases as data subsetting percentage increases, as we would expect. And $\alpha$ rate is reduced as the number of partitions is increased, especially for the cases where queries return a small fraction of the data. For example, when the data subsetting percentage is smaller than 40%, $\alpha$ rate for 96 blocks is 50% less compared to the case when there is no partitioning.

3.4.3 Parallel Comparison with FastQuery

This subsection compares our parallel indexing method with FastQuery[32]. FastQuery also provides a parallel indexing solution using bitmap technique as the basis. This method builds a relational table view over scientific dataset and supports parallel indexing based on the partitioning of this table. Their method is efficient if only rows and columns are involved in data subsetting. However, one important feature of scientific data is that they are multi-dimensional. Mapping a multi-dimensional dataset to a one dimensional relational table can be very time consuming. Furthermore, most of data subsetting is based on multiple dimensions or coordinates. If the data is always partitioned as one dimension, it can reduce the locality of bitwise indexing operations and generate a large number of data segments for data retrieval.

Compared with FastQuery, our method has a key advantage, which is that our partitioning strategy divides data blocks in an intelligent fashion, leading to better performance.
We have experimentally compared our approach with FastQuery. We implemented the operations in FastQuery according to the description in their paper [32].

In this experiment, we divided queries into 4 categories, which are: queries include subsetting over values and the first dimension, subsetting over values and the second dimension, subsetting over values and the third dimension, and a combination of different dimensions. We generated 100 queries for each category. The number of partitions over 8.2 GB dataset is 48. Figures 3.8(a) and 3.8(b) show the $\alpha$ rate and total I/O times between FastQuery and our method. We can see that both $\alpha$ rate and I/O times are similar if queries only include the first dimension. Although in FastQuery, data is mapped to a one-dimensional table, subsetting over the first dimension still maintains continuity of data elements. However, for queries over the second and third dimensions, mapping multi-dimensional data into one dimensional table reduces data locality, and results in a large number of data segments that need to be independently retrieved. Thus, the average I/O access time for FastQuery is around 4000 times more than our method. Figure 3.9 shows
the execution time between these two methods. Again the performance is similar when queries only include the first dimension. In other cases, our parallel indexing achieves a speedup varying from 1.41 to 2.12.

3.5 Related Work

Indexing and parallel indexing technologies have been widely studied in the data management area. However, most of the methods are derived from B-tree indexing [34], which is mainly designed for transaction type applications. In scientific data management, most approaches are built around bitmap. Fastbit [132], which uses WAH compression algorithm [130], supports bitmap indexing over scientific dataset. However, with run-length compression algorithms such as WAH, the ability to directly locate dimension subset is lost. Several methods have been applied to solve this problem. Instead of run-length compression, multi-hashing compression algorithm [12] is designed to compress bitmap indices while at the same time supports subsetting based on table ids. This algorithm can achieve 1 to 3 orders of magnitude comparing to WAH. But the disadvantage is that it loses 10%
precision due to hashing conflicts. FastQuery [32] solves this problem by building a re-
lational table view over scientific dataset, and supporting parallel bitmap indexing based
on a collection of sub-tables. We have extensively compared our parallel indexing method
against FastQuery. Again the main difference is that our method supports parallel indexing
based on a flexible multi-dimensional partition strategy.

3.6 Summary

This chapter describes a new indexing approach suitable for supporting a variety of
subsetting queries over scientific datasets (arising from scenarios where datasets need to be
visualized with tools like ParaView). Besides supporting select queries over dimensions,
we also support queries over coordinates and queries over values, by using and optimizing
multi-level bitmap indexing. Moreover, we have given a parallel indexing architecture and
an intelligent index partitioning strategy to improve query processing efficiency.

We have extensively evaluated our implementation. We first compared a sequential
implementation with the filtering method in ParaView, and demonstrated that in almost
all cases our solution has better performance and smaller memory requirements. We also
showed the scalability of our indexing method in a parallel environment. Finally, we com-
pared our method with another parallel indexing method, FastQuery [32]. We demonstrate
that, on the average, the performance of our method is better than FastQuery by a factor of
1.77.
Chapter 4: SDQuery DSI: Integrating Data Management Support with a Wide Area Data Transfer Protocol

The previous two chapters describe a light-weight server-side data management framework to improve data analysis efficiency. Bitmap indexing is also applied to improve the time cost of value-based subsetting. In this work, we further integrate the flexible data subsetting work with the existing efficient wide-area data transfer protocols. Specifically we focused on Globus GridFTP [45]. GridFTP 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 [35] (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 [10, 63, 65, 73, 76, 60, 13] 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.
4.1 Background: 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 4.1 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 capabilities. 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.

4.2 Motivating Application

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 solutions 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 [80]. With support for subsetting, future data rates can be adequately handled.

### 4.3 System Overview

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

As we had summarized in Section 4.2, 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. This can be achieved because GridFTP allows a new DSI to be loaded at runtime. 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 [48, 89] 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.
4.3.2 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 4.2 shows the data schema example of a NetCDF file, which is generated by the Parallel Ocean Program (POP) [56]. 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 4.2, 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 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.

### 4.3.3 System Architecture

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”.

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 4.3 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 4.3, 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 that 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 4.4 shows an example of a SQL query and data retrieval request based on the data schema we had shown earlier in Figure 4.2. 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 know 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 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 generate a data subsetting request by combining the SQL grammar information with the metadata information as output.
Figure 4.4: An Example Query and GridFTP Data Retrieval Request Embedding the Query

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 4.4.3. 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 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 4.4.1 and 4.4.2, respectively.

4.4 System Optimizations

This section describes several optimizations implemented in the system.

4.4.1 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, which we refer to as the *preparation stage* 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 data 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>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>Size of the entire dataset</td>
</tr>
<tr>
<td>BN</td>
<td>Total number of blocks in the dataset</td>
</tr>
<tr>
<td>SS</td>
<td>Size of the query results</td>
</tr>
<tr>
<td>SN</td>
<td>Total number of segments</td>
</tr>
<tr>
<td>$T_{db}$</td>
<td>Average time to load one data block from disk to memory</td>
</tr>
<tr>
<td>$T_{de}$</td>
<td>Average time to load one data element from disk to memory</td>
</tr>
<tr>
<td>$T_{md}$</td>
<td>Average time to filter one data element in memory</td>
</tr>
<tr>
<td>$T_{seg}$</td>
<td>Time for generating data segments</td>
</tr>
<tr>
<td>$T_{p}$</td>
<td>Average time to prepare for data read operation</td>
</tr>
<tr>
<td>$T_{dl}$</td>
<td>Average time to locate the start position in the dataset</td>
</tr>
</tbody>
</table>

The complete list of all parameters used in our model is shown in Table 4.1. 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:

\[ T_{MF} = BN \times (T_p + T_{dl} + T_{db}) + FS \times T_{ml}. \]  \hspace{1cm} (4.1)

\[ FS = \text{MIN} (SS, DS - SS). \]  \hspace{1cm} (4.2)

Equation 4.1 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 read preparation time, data seek time, and the data transfer time of each block. The data transfer time is much larger than the read preparation time and data seek time.

The second part of Equation 4.1 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 4.2. 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 the result bitset. 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 4.1, 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:

$$T_{DA1} = SS \times (T_p + T_{dl}) + SS \times T_{de}.$$  \hspace{1cm} (4.3)

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, data seek time, and the data transfer time. Alternatively, we can use segmentation, and the resulting cost will be:

$$T_{DA2} = T_{seg} + SN \times (T_p + T_{dl}) + SS \times T_{de}.$$  \hspace{1cm} (4.4)

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 ($T_p$), average seek time ($T_{dl}$), the time for loading 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 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.

### 4.4.2 Parallel Streaming Data Transfer

Parallel streaming data transfer, as supported in our system, involves the following: 1) 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, and finally, 3) data reading and data transfer operations are performed in a pipeline mode, and thus, 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 4.5 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 *result bitset*, containing the point IDs (bits with the value 1) that satisfy the current query. As an example, in Figure 4.5, 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 4.5, 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 is counted and segmented. 2) Merge operations are performed to group intervals into blocks based on the count of 1-bits number, and segments are also grouped together between intervals. As an example, in Figure 4.5, 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 (Thread₀₀ in the first steam and Thread₁₀ 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.

4.4.3 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 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. Then we are able to 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.

### 4.5 Experimental 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, and finally, (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 dataset, one for each format. For NetCDF, we used the datasets generated by the Parallel Ocean Program (POP) [56]. 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*, and with 100 time-steps, 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 different server-client pairs, to evaluate our approach with different bandwidths. The first situation involves transfers over a local area network (LAN) with an speed of 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.

4.5.1 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 Subset & 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 used the Direct Access with segmentation method to retrieve data subset. Optimizations using performance model-based data retrieval will be emphasized in the next experiment. Two streams were used for both methods in the results we report.
Figure 4.6: Efficiency Comparison between SDQuery DSI and File DSI for POP Dataset

Figure 4.7: Efficiency Comparison between SDQuery DSI and File DSI for MODB Dataset

Figure 4.6 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 4.6, 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 to 29.07. This is because the total execution time of a transfer request is now dominated by the network time. As showed 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 to 81.32. Moreover, if we compare the query processing time with data transfer time of SDQuery DSI, we can see that for different network environments, the query processing over bitmap indices has much smaller time cost than the disk IO 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 4.2.

Figure 4.7 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.

### 4.5.2 Effectiveness of the Performance Model

This subsection evaluates the effectiveness of the performance model based *hybrid method*. As explained in Section 4.4.1, 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 4.8: Data Retrieval based on Performance Model: Different Datasets and Platforms

Figure 4.8 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 to 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 is better but we choose 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, but 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 of 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 can say that our performance model either chooses the best method, or results in performance that is only 1-2% slower that 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 seeking 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.

4.5.3 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 4.4.2, 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 No Stream 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 No Stream method, we use a 10.5 GB dataset here. The network speed is 200 Mb/s.

Figure 4.9 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 includes both data subsetting time and network transfer time (we ignore the query processing time here). 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 reading time is effectively overlapped by the data transfer time. The speedup 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 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 number of parallel streams depends on both network bandwidth and RTT, and the range is from 2 to 16 in most cases.

4.5.4 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 reading 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 4.10 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.

4.6 Related Work

In this section, we compare our effort with the work that has been in the context of Globus GridFTP (DSI implementation).

By default, Globus GridFTP has its own File DSI [45] to support data fetching on POSIX systems. Several other DSIs [59] 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 [90]. Hans-Christian et al. [49] 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.
4.7 Summary

This chapter 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 rein- stallation. 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.
Chapter 5: Taming Massive Distributed Datasets: Data Sampling
Using Bitmap Indices

In previous chapters we have provided a light-weight server-side data subsetting method based on bitmap indexing to improve data analysis efficiency. However, in an environment where scientific analysis is mainly focused on the global data or even the data size after subsetting is still huge, flexible server-side data sampling becomes a useful method to quickly decrease the data size and improve data analysis efficiency. There are at least three problems when we tried to apply data sampling in scientific data management area:

1. Creating sub-sampled (lower-resolution) datasets from a high resolution simulation dataset, on demand and efficiently, while maintaining the characteristics of the original dataset.

2. Assessing the loss of quality (with respect to the key statistical measures) incurred with a particular level of resolution, on the given dataset, without having to take a pass through the entire high resolution dataset.

3. Providing the above functionality in a flexible system, which can support sampling at the server-side in response to requests from the client-side, and combine sampling with data subsetting.
In this work, we address the above limitations by developing a novel sampling approach. We observe that to allow subsetting over scientific datasets, data repositories are likely to use an indexing technique [97]. Among these techniques, we see that bitmap indexing can not only effectively support subsetting over scientific datasets, but can also help create samples that preserve both value and spatial distributions over scientific datasets. We have developed algorithms for using bitmap indices to sample datasets. We have also shown how only a small amount of additional metadata stored with bitvectors can help assess loss of accuracy with a particular subsampling level, i.e., we do not need to take a pass over the entire sampled dataset to calculate accuracy based on these metrics. Some of the other properties of this novel approach include: 1) value distribution as well as spatial distribution of the original dataset are preserved, 2) sampling can be flexibly applied to a subset of the original dataset, which may be specified using a value-based and/or a dimension-based subsetting predicate, and 3) no data reorganization is needed, once bitmap indices have been generated.

5.1 System Overview

This section gives an overview of the system we have developed to support flexible server-side sampling (and subsetting) of large datasets. Technical details of the sampling method will be given in the next Section.

Figure 5.1 shows a high-level overview of our system. In Chapter 2 and Chapter 3 we designed a system to support flexible data subsetting (including both value-based and dimension-based predicates) using a standard SQL-like interface [96, 97]. The advantage of this approach is that a simplified virtual or high-level view of the dataset is presented to users. Thus, users downloading the data do not need to be familiar with the details of
the data format. Instead, they can specify subsetting (and now sampling) requests with the high-level view.

There are two main modules in the system, the *Query Analysis Module* and *Query Execution Module*. The *Query Analysis Module* takes an SQL query and corresponding metadata as input and generates a query request (in a specific format internal to the system) as the output, which is the same module as described in Figure 3.2 of Chapter 3.

The second major module, *Query Execution Module*, takes the query request as input, performs data subsetting and sampling based on bitmap indices, and sends the data result back to the client. The *Bitmap Indexing* performs different indexing operations based on the query request and generates a collection of bitvectors which satisfies the current query as output. After that, we check to see if sampling is needed for the current query. If data sampling is not required here, the *Data Reader* will query the data subset based on the
indexing information and return the result to the client. Otherwise, the *Data Sampling* sub-module will generate data samples based on bitmap indices.

There are two main components in the *Data Sampling* sub-module: *Error Prediction* and *Index-based Sampling*. Our approach includes a novel error prediction mechanism based on bitmap indices. With the help of this mechanism, we are able to pre-calculate approximation errors before actually sampling the data. Moreover, the error estimation can be performed based on indices instead of scanning through the entire sample. While the latter also reduces the error calculation time, our pre-calculation method allows a user to choose a sampling level which maintains a desired level of accuracy. Moreover, this alleviates the need for extracting a sample, calculating the error, and then resampling (likely with a different subsampling level), which can be very expensive in practice.

After error estimation, the *Index-based Sampling* component performs data sampling directly over bitmap indices and generates a set of data record identifiers as the result. Then the *Data Reader* will take the data record identifiers as the input, extract the data records, and return the results.

Besides error pre-calculation, which can improve the overall sampling efficiency significantly, and the overall effectiveness of our method, there are at least three other advantages for our system.

*Small Preprocessing Costs:* If the data repository already uses bitmap indices, or will like to use bitmap indices to efficiently obtain subsets of the original dataset, we can directly apply our sampling method without any preprocessing. For those applications without bitmap indexing support, the computational complexity of index generation is only $O(n \log(m))$ where $n$ is the number of total elements and $m$ is the number of bitvectors [132]. With
the help of binning, \( m \) can be much smaller than \( n \), so \( \log(m) \) can be considered a constant number. Thus, our method is much faster compared with sampling methods with \( O(n \log(n)) \) preprocessing time, such as the KDTree-based method [128]. Another advantage of our method is that we do not need any modifications or reorganization of the original dataset. All sampling operations are performed using data in the original format and the bitmap indices.

*Tradeoff between Accuracy and Sampling/Memory Costs:* The bitmap indexing allows flexible multi-level indices over a given dataset. The low-level bitmap indices are able to reflect data features at a fine granularity, whereas the high-level indices improve the efficiency by binning a group of low-level bitmap indices together. By choosing to perform sampling using high-level or low-level bins, and even choosing the bin size at one or both levels, one can achieve the desired tradeoff between accuracy of sampling and time/memory costs of the sampling process.

*Combining Sampling and Subsetting:* Because our system is built on top of a data subsetting system, users can combine sampling with subsetting. Moreover, such queries can be executed efficiently because of the properties of bitmap indices. We will elaborate on this later.

### 5.2 Sampling Using Bitmap Indices

This section first introduces our data sampling method using bitmap indices. Then we describe four enhancements of our sampling method, which are error prediction, sampling over flexible data subset, sampling to support multi-attributes data analysis, and multi-resolution data sampling to support interactive post-analysis. Finally we propose a parallel
5.2.1 Stratified Random Sampling over Bitvectors

Consider data storage in a large-scale scientific repository. If we are using bitvectors to be able to retrieve subsets of the original dataset [32, 97], the question we want to focus on is “can the same bitvector be used to obtain accurate and representative samples, while also assessing the loss of accuracy with a particular sampling level”. It turns out that bitvectors can not only be used in this fashion, but they also provide several advantages over existing and popularly used sampling techniques.

We now describe the bitvector based sampling method we have developed. The basic idea in our method is to perform random stratified sampling over each bitvector, which corresponds to a particular value or, more likely, a bin of values. Specifically, we extract the same percent of samples out of each bitvector. By sampling over bins with equal probability, we are able to keep value distribution in the sampled dataset close to that of the original dataset. In fact, as we will show below, this approach preserves entropy of the original dataset, a highly desired property of samples in many applications.

Within each bitvector, we first divide the bitvector into sectors of a certain size, and choose the same percent of samples out of each sector. This way, we can also preserve the value distribution within each spatial region. Furthermore, when multi-level bitvectors are created this method can be applied to either the low-level or the high-level index. This choice allows a tradeoff between efficiency and accuracy.

We now explain the steps of our method in more detail, using an example in Figure 5.2. There are three main steps:
Sampling over Low Level Indices:

\[ e_0 = 1 \]
\[ 00010100 | 00110001 | 01010110 | 00010001 \]
\[ se_0 = 1 \]
\[ 00000100 | 00100001 | 00000110 | 00010000 \]

\[ e_1 = 2 \]
\[ 00100000 | 10000010 | 00100000 | 10000010 \]
\[ se_1 = 2 \]
\[ 00000000 | 10000000 | 00100000 | 00000010 \]

\[ e_2 = 3 \]
\[ 10001001 | 00001000 | 10001001 | 00100100 \]
\[ se_2 = 3 \]
\[ 00001000 | 00000000 | 10000001 | 00100000 \]

\[ e_3 = 4 \]
\[ 01000010 | 01001000 | 00000000 | 01001000 \]
\[ se_3 = 4 \]
\[ 01000000 | 00001000 | 00000000 | 00001000 \]

Sampling over High Level Indices:

\[ i_0 [1, 2] \]
\[ 00110100 | 10110011 | 01110110 | 10010011 \]
\[ si_0 [1, 2] \]
\[ 00100000 | 10000010 | 00010100 | 10010000 \]

\[ i_1 [3, 4] \]
\[ 11001011 | 01001100 | 10001001 | 01101100 \]
\[ si_1 [3, 4] \]
\[ 10001001 | 00001000 | 10001000 | 01000100 \]

Figure 5.2: Our Sampling Method: Stratified Random Sampling over Bitmap Indices
**Building bitmap indices:** In this example, the small dataset contains 32 elements, so each bitvector has 32 bits. The number of distinct values is 4. The low-level bitmap indices contain 4 bitvectors: \( e_0(=1) \), \( e_1(=2) \), \( e_2(=3) \), \( e_3(=4) \), and the high-level bitmap indices include 2 bitvectors: \( i_0([1, 2]) \), \( i_1([3, 4]) \). In this simple example, all values are integers, though bitmap indices can be (and have been) used for floating-point values by generating bins with value ranges.

**Dividing bitvectors into sectors:** In order to preserve distribution of values in each spatial region, bitmap indices should be logically divided into spatial sectors. In the figure, we can see that for both the low-level and the high-level bitmap indices, every bitvector is divided into 4 sectors, and there are 8 bits within each sector.

**Random sampling over each sector:** After creating sectors, random sampling can be performed within each sector, and for each bitvector, to generate data samples. Within each bitvector, random sampling is only applied to 1-bits. To preserve value distribution within each region, we need to make sure sample percentages over each sector are the same. One advantage of using bitmap indexing is that its implementations help us locate all 1-bits efficiently. In Figure 5.2, we are generating 50% samples out of the original dataset. We can see that \( se_0, se_1, se_2, se_3 \) are identifiers of data records that are in the sample generated using the low-level bitvectors, whereas \( si_0, si_1 \) are the data records for the sample using the high-level bitvectors. For both low-level and high-level bitmap indices, within each sector, only half of the 1-bits are picked. For example, after sampling, the number of 1-bits in the sample bitvector \( se_0 \) is 6, which is only half of that in original bitvector \( e_0 \).

From the figure, we can also see that although low-level bitmap indices have more bitvectors, each bitvector has fewer 1-bits. On the other hand, the number of bitvectors in the high-level bitmap indices is smaller, but more 1-bits exist in each bitvector. Hence, both
methods generate sampled datasets of the same size. Low-level bitmap indexing is able to achieve better accuracy because it reflects the value distribution at a finer granularity. However, it also has an additional time cost, because of higher indices loading time and bitvector striding time.

Finally, we point out the property of this method with respect to preserving entropy. Information theory and entropy have been extensively used while sampling data (or even selecting angles, streamlines, or other features) in graphics and visualization, as also summarized by Xu et al. [133].

Formally, if $X$ is a random variable with a series of possible outcomes $x$, where $x \in \{x_1, x_2, \ldots, x_n\}$, and if the probability for the random variable to have the outcome $x_i$ is $p(x_i)$, then Shannon’s entropy is defined as

$$ H(X) = \sum_i p(x_i) \times \log(1/p(x_i)). $$

Assuming no binning is performed, and sector sizes are large enough that precisely the same fraction of values can be chosen, we can see that the sampled dataset using bitvectors will have the same distribution of values, or the same entropy.

### 5.2.2 Error Prediction

After sampling, it is also important to know how accurate the sampled dataset is compared with the original dataset. Traditional sampling methods can only calculate error metrics after samples are generated, and if the error is too high, the entire sampling process has to be repeated with another sample percentage. As we will show now, with bitvectors we are able to pre-calculate error metrics based on bins. Thus, we can perform error predictions analysis to find a sample percentage which will give desired accuracy levels, and then can perform data sampling only once. This is a significant advantage, since the error
calculation method only takes at most \(O(m)\) time, where \(m\) is the number of bitvectors. In comparison, sample generation normally takes \(O(n)\) time, where \(n\) is the number of data records in the original dataset, and \(n >> m\).

As we stated earlier, while evaluating quality of a sampled dataset, different error metrics, like mean, variance, histogram and Q-Q plot are used. In particular, for our discussion we consider error metrics of two types: 1) mean, variance, histogram, and Q-Q plot for each variable, and 2) mean and variance for each sector.

We need to calculate and store some additional information during bitmap index generation. Figure 5.3 shows the metadata generation over bitmap indices. For dataset or variable level error calculation, the only additional information we need is the total number of 1-bits within each bitvector. From the figure, we can see that \(\text{count}(0)\), \(\text{count}(1)\), \(\text{count}(2)\) and \(\text{count}(3)\) record the total number of 1-bits for each bitvector. The results are stored in the first column of the 2-dimensional count matrix \(c\). The metadata we need for sector-level mean and variance calculation is the number of 1-bits within each sector. From

---

**Figure 5.3: Metadata Generation for Error Prediction**
the figure, we can see that for bitvector \( e_0(=1) \), \( count(0, 0) \), \( count(0, 1) \), \( count(0, 2) \) and \( count(0, 3) \) record the number of 1-bits within each sector. The result is stored in columns \( c_0 \), \( c_1 \), \( c_2 \) and \( c_3 \) of the count matrix.

Now we elaborate on calculation of specific metrics. Our approach can also be referred to as error pre-calculation, which is in contrast to error post-calculation normally done with the traditional sampling methods.

**Mean, Variance, Sector Means, and Sector Variances:** We now show how to pre-calculate mean and variance of the sampled dataset based on bins and the count matrix. The input is the representative value (value) of each bin, which we determined at the time of index generation, and the total number of elements (count) within each bin, which we can find from the count matrix. Besides that, each time we also set a sample percentage to decide the size of the sample result, denoted as \( SamplePercent \). Equation 5.1 computes the number of samples selected from each bitvector \( scount_i \) based on \( count_i \) and \( SamplePercent \):

\[
scount_i = count_i \times SamplePercent. \tag{5.1}
\]

Our method fetches the same percent samples out of each bitvector, which is equal to \( SamplePercent \). Hence, by multiplying \( count_i \) with \( SamplePercent \), we are able to compute the approximate number of samples within each bitvector. Now, Equation 5.2 calculates the mean value of the sampled dataset:

\[
Mean = \frac{\sum_{i=1}^{m} (scount_i \times value_i)}{\sum_{i=1}^{m} scount_i}. \tag{5.2}
\]

Within each bitvector, we know both the representative value \( value_i \) and sample size \( scount_i \). By multiplying these two factors together, we can get the sum value of samples in the current bitvector. Based on that, we can calculate the total value by adding the sum value of each bitvector together. We are also able to count the total number of sample elements.
by adding $scount_i$ of each bitvector together. Based on the sum value and total sample elements count, we can get the *mean* value.

Equation 5.3 calculates the *variance* of the sampled dataset. We first compute the value differences within each bitvector based on *mean* and *value*$_i$, then add all value differences together and finally divide by the total number of sample elements:

$$Variance = \frac{\sum_{i=1}^{m} (scount_i \times (Mean-value_i)^2)}{\sum_{i=1}^{m} (scount_i)}.$$  \hspace{1cm} (5.3)

The method of calculating *sector means* and *sector variances* is similar. We simply need to apply the Equations 5.2 and 5.3 for each sector.

We can see that our approach, error pre-calculation, can calculate *mean* and *variance* within $O(m)$ where $m$ is the total number of bitvectors. Note that in contrast, the error post-calculation method will have to scan the entire sampled dataset twice to compute the mean and the variance. The time complexity is $O(s)$, where $s$ is the sample size.

*Histogram:* The input is still *value*, *count* and *SamplePercent*. Based on Equation 5.1, we can obtain the number of sampled elements for each bitvector ($scount_i$). Now,

$$Prob_i = \frac{scount_i}{\sum_{i=1}^{m} (scount_i)}.$$  \hspace{1cm} (5.4)

Equation 5.4 calculates each value $Prob_i$ in the histogram by simply dividing the sample size of each bitvector $scount_i$ by the total sample size. This way, we obtain the element probability of each bitvector. By calculating probabilities over all bitvectors, we are able to generate a histogram.

This method can compute the histogram within $O(m)$, where $m$ is the number of bitvectors. In comparison, error post-calculation has to first perform a Radix Sort$^9$ over the entire sampled dataset. After that, it needs to count the number of elements within each bucket.

$^9$http://en.wikipedia.org/wiki/Radix_sort
and then divide this number by the total sample size. The time complexity is $O(s)$ where $s$ is the sample size.

**Q-Q Plot:** We first recap the definition of a Q-Q plot. Viewing the original dataset and the sampled dataset as two distributions, we compare them by plotting their quantiles against each other.

Algorithm 4 shows how to calculate a Q-Q plot using bitvectors. The input is $s$, which indicates the total number of sample elements; $m$, the total number of bitvectors; $q$, the total number of quantiles; $count$, the number of elements within each bitvector; and $value$, the *representative value* of each bitvector (calculation described below). In line 1, we define a variable $curCount$ to record the total number of elements that are smaller than the value of the current bitvector. The variable $pos$ indicates each quantile position identifier in the sampled dataset. It can be computed based on total sample size($s$), multiplying it with the quantile percentage, as shown in line 8. Lines 3 to 12 compute the quantile value based on each quantile position. We iterate from the bitvector with the smallest value to the bitvector with the largest value. If the current quantile position $pos$ is larger than $curCount$, we update the $curCount$ and go to the next bitvector, as shown in line 4 and line 10. If $pos$ becomes smaller than $curCount$, it means the current quantile is located within the current bitvector. Then we can record the representative value of the current bitvector as the quantile value and go to the next quantile, as captured by lines 5 through 8. We keep performing this calculation until we find the value of all the desired quantile positions.

Our method is able to calculate the Q-Q plot with $O(q)$ in the best case and $O(q + m)$ in the worst case, where $q$ is the total number of selected quantiles. In comparison, the error post-calculation method has to first perform a quick sort over the entire sampled dataset to calculate the Q-Q plot. After that, certain quantiles need to be selected out of the sorted
Algorithm 4: Compute QQPlot($s, m, q, count, value$)

1: $curCount ← 0, pos ← 0$
2: $i ← 0, j ← 0$
3: while $i < m && j < q$ do
4:     $curCount ← curCount + count_i$
5:     if $curCount > pos$ then
6:         QQPlotArray$_j$ $←$ value$_i$
7:         $j ← j + 1$
8:     pos $←$ $s * j/100$
9:     else
10:     $i ← i + 1$
11: end if
12: end while

dataset as Q-Q plot values. For example, we can fetch the data elements located at 1%, 2%, . . . ,100% positions out of the sorted sample dataset as the result. The time complexity is $O(s \times \log(s))$ where $s$ is the sample size.

Now, we describe how we calculate $value$, the representative value of a bitvector, when we have multi-level bitmap indices. For low-level bitmap indices, we can simply use the mean or the median value as the representative value of each bin. For high-level bitmap indices, each bitvector indicates a relatively larger value range. In our work, we use three indicators to predict errors for high-level bitmap indices. In high-level bitmap indices, each bin indicates a value range which has both a lower-bound and an upper-bound. By using lower-bound and upper-bound values during the error prediction process, we are able to calculate a boundary on the actual error metric results. Besides, each high-level bin is built by combining a group of low-level bins together. Hence, we are able to calculate the value distribution of each high-level bin by looking at corresponding low-level bins and finding an estimated value to represent each high-level bin. This way, we are able to find the actual error boundaries and also generate a relatively accurate error prediction. In some cases,
when the data range of the dataset is large, the bin size of low-level bitmap indices can be big. We can also apply this three indicators method to low-level bitmap indices.

### 5.2.3 Sampling Only a Subset of Data

When a data repository is disseminating data, a particular user might only be interested in a certain subset of data, based on spatio-temporal ranges (dimension subsetting) and/or specific values for attributes (value-based subsetting). However, as the dataset size for the subset may still be too large, sampling may still be needed.

Traditional sampling methods cannot efficiently support data sampling over a user-specified subset of data that includes value-based subsetting. For example, simple random sampling, stratified random sampling and KDTree stratified random sampling methods can all handle dimension-based subsetting, but when value-based subsetting is involved, they have to first generate data samples over the entire dataset and then perform post-filtering, which is clearly not efficient.

Suppose we need to sample datasets at a certain level, in conjunction with a subsetting condition, which includes both dimension-based and value-based subsetting conditions. We will proceed as follows. We first focus on the value subsetting conditions and search the (possibly) multi-level bitmap indices to find corresponding bitvectors. Only these bitvectors need to be loaded. Next we perform dimension subsetting over the retrieved bitvectors. Finally, we apply the stratified sampling only over this bitset.

### 5.2.4 Data Subsetting and Sampling over Multiple Attributes

In a typical scientific dataset, certain attributes can be stand-alone, i.e., can be analyzed separately. On the other hand, certain attributes can be closely connected with each other, and it is better to study them together. Suppose we consider the output from the cosmology
data described in Section 5.3 below. Each record in the dataset corresponds to one particle and includes multiple attributes. For example, the attribute mass indicates the field value related to the current particle, and VX, VY, VZ indicate the particle velocity in each of the three spatial dimensions. mass can be analyzed separately, as it does not have a strong connection with the other attributes. For VX, VY, VZ, however, scientists prefer to analyze them together to find the relationships among them.

The techniques we have described so far build indices over each attribute separately, which does not fit the second scenario very well. We now describe an extension to support sampling to ensure a preserved distribution over multiple attributes.

Suppose we need to sample with respect to two attributes, X and Y. The entire process can be divided into 3 steps: (1) Divide the value range of each attribute into one-attribute bins, say, \((X_1, X_2, \ldots, X_{m_1})\) and \((Y_1, Y_2, \ldots, Y_{m_2})\). (2) Form multiple attributes bins (or mbins) \((X_1, Y_1), (X_1, Y_2), \ldots, (X_{m_1}, Y_{m_2})\) based on the one-attribute bins generated in the previous step. For each mbin, generate a bitvector and initially set all bits to 0. (3) Scan through the dataset. For each record, find its X and Y value, classify it into the corresponding mbin and set the corresponding bit to 1. Repeat this process until all records are mapped to related mbins.

### 5.2.5 Multi-Resolution Sampling to Support Interactive Post-Analysis

In some scenarios where users are interested in interactive data analysis or visualization, multi-resolution sampling is necessary to provide different granularity of the data to users. For example, after examining the current level of samples, users may want to either explore the upper-level of samples for more information or go to the lower-level of samples for
more efficient analysis. In such case, how to support multi-level samples to answer the interactive post-analysis or visualization becomes very important.

Figure 5.4 shows the process of generating multi-resolution samples using our index sampling method. Initially there are $m$ bitvectors for the current dataset ($e_0, e_1, ..., e_m$). For each bitvector $e_i$, we apply the stratified random sampling method to generate corresponding first-level sample bitvector ($se_i(50\%)$ in this example). After that, we continue to apply stratified sampling over the first-level sample bitvectors to generate the second-level sample bitvectors ($se_i(25\%)$ in this example). We keep generating lower level bitvectors based on current level until all resolution levels are generated. For each level sample bitvectors, we perform logic OR operations among them to generate one bitset as the $i$th level sample bitset. Compared with other multi-resolution model, one advantage of our method is that with the help of bitmap indexing, we do not have to store the actual sample data in each level. When the data size is huge, store multi-level of samples will be extremely resource consuming. Instead, we are able to store the compressed bitset to represent each resolution.
level, which greatly saves the spaces. During data analysis, each time when users want to examine one sample level, we first provide the error prediction results of current level. If users are interested in the data sample, we find the actual sample based on the IDs specified within the current bitset. Moreover, another feature of our method is that each upper-level bitset contains all bits of lower-levels. If the data analysis is moved from one-level to another, we do not have to reload all the sample data. Instead, we are able to just add or remove the data elements missed between two levels. And this can be achieved by simply one XOR operation between two sample bitsets to generate IDs of the missing elements, which is quite efficient.

5.2.6 Parallel Indexing and Sampling

Nowadays scientific data is increasing in a rapid speed and parallel data sampling is even necessary when the data sizes become extremely large. This subsection proposes a MPI-based parallel sampling framework which generates sample dataset out of original dataset based on distributed bitmap indices. Different parallel levels can be first defined for each scientific dataset, and we can choose a parallel level based on the size of the
dataset and available processes. If the number of processes is sufficiently large, we can build up multiple distributed index files over sub-blocks of the data. Then, each process is responsible for performing data sampling over one index file (correspond to one data block).

Figure 5.5 shows our parallel indexing and sampling framework. Here we define three parallel levels: data files, variables and data blocks. As shown in the figure, one scientific dataset includes multiple data files, one data file includes multiple variables, and one variable can be logically partitioned into multiple blocks. To support parallel indexing and sampling, during the index generation phase, instead of building and compressing bitmap indices over the entire variable, we first logically partition each variable into a collection of blocks, and then initialize multiple processes and make each process build bitmap indexing over a set of blocks. This way, the index generation is performed in parallel and is able to achieve a good speedup. A global metadata file, which keeps the relationship between dimension boundaries and bitmap indices of each block, is generated. During the index sampling phase, each process will be assigned with a certain number of index files, depending on the number of index files and available processes. If the number of processes is sufficiently large, each index file will be assigned with one process to generate data samples of current data block. This way, the sampling operations over the entire dataset can be totally performed in parallel, which greatly improves the efficiency.

5.3 Experimental Results

In this section, we report results from a number of experiments conducted to evaluate our sampling approach. We designed experiments with the following goals: (1) To show how data sampling is able to improve data analysis efficiency in a distributed environment
(where data source and resources for data analysis are geographically separated), (2) To examine the accuracy of our bitmap indices sampling method and compare it with a number of other sampling methods, (3) To evaluate the accuracy of error pre-calculation, by comparing predicted errors with the actual errors, (4) To compare the efficiency of our method against other sampling methods, in particular in view of error pre-calculation, (5) To show how sampling over data subsets improves the efficiency, and (6) To show how parallel indexing and sampling improves the efficiency.

We used two different scientific datasets. The ocean dataset is generated by the Parallel Ocean Program (POP) [56], which is an ocean circulation model. The execution we used has a grid resolution of approximately 10 km (horizontally), and vertically it has a grid spacing close to 10 m near the surface, increasing up to 250 m in the deep ocean. POP generates 1.4 GB output for each variable per time-slice, and each variable is modeled with three dimensions: longitude, latitude, and depth. The data is stored in the NetCDF format. The cosmology dataset is generated by the Road-Runner Universe MC$^3$, which is a large N-body cosmology simulation of dark matter physics. An MC$^3$ time step of 4000$^3$ (64 billion) particles with 36 bytes per particle takes 2.3 TB per time-slice. The particles generated per time-slice are split into a collection of data files based on the spatial information. Each particle within the file corresponds to one record, which is formed by 8 attributes (X, Y, Z, VX, VY, VZ, MASS, TAG). The data is stored in binary format.

In our experiments, the data repository and the server-side data sampling are on the Darwin Cluster at Los Alamos National Laboratory. Darwin consists of 120 compute nodes with 48 core (12-core by 4 socket) 2GHz AMD Opteron 6168 and 64 GB memory. The client-side data analysis is performed on one compute node which has 8 cores Intel(R) Xeon(R) CPU 2.53GHz and 32 GB memory.
5.3.1 Improving Efficiency of Distributed Data Analysis with Sampling

In this experiment, we consider the following scenario. The entire dataset is located on a remote server, and any analysis must be done after the data is downloaded to the client-side. We consider two distinct applications: data visualization and data mining. In the data visualization scenario, we visualize the sampled dataset using Paraview [8], a widely used data analysis and visualization application. In the data mining scenario, we take data samples as input and perform K-means clustering using MATE [54], a map-reduce like system. With these two applications, we compare the efficiency of data analysis (including data downloading time), when using the original dataset against the cases where different subsampling levels are used. In particular, we divide the data processing time into three parts: 1) Server-side data sampling time, 2) Data transfer time between the server and the client, and 3) Client-side data analysis time. The second factor above varies with the wide-area data transfer bandwidths one might have. For our experiments, we used two different networks, one with 10 MB/s bandwidth and the other with 100 MB/s bandwidth.

Figure 5.6(a) compares the efficiency of the data visualization using different subsampling levels: 100%, which means that we are using the original dataset without sampling, 12.5%, 1%, and 0.1%. The dataset without sampling is 11.2 GB in size and is from the POP application. From the figure, we can see that although our method incurs extra sampling costs compared to the case when the original dataset is analyzed, both the data transfer and analysis time is much lower (as expected), and more than compensates for the sampling time. Specifically, we find that compared to visualization over the original dataset, if network bandwidth is 10 MB/s, the speedup with 12.5% sampling rate, 1% sampling rate, and 0.1% sampling rate is 4.82, 15.91, and 47.59, respectively. If network bandwidth is
100 MB/s, the corresponding speedups are 2.61, 6.72, and 19.02, respectively. Of course, another consideration with sampling is the accuracy of the analysis, which we will focus on in the next subsection.

Figure 5.6(b) compares the efficiency of K-means clustering (data mining) execution, using the original dataset and the three sampling levels (12.5%, 1%, and 0.1%). The dataset is from cosmology, and is 16 GB in size. The number of K-means cluster centers is 10 and the number of iterations is 50. The number of threads is 4. From the figure, we can see that, similar to data visualization, with the help of sampling, the speedup with 10 MB/s network bandwidth ranges from 5.25 to 84.24, and the speedup with 100 MB/s network bandwidth ranges from 3.26 to 39.8. Again, accuracy is another consideration, which we will analyze next.
Figure 5.7: Error (Means, Histogram, and Q-Q Plot) Comparison Using Cumulative Frequency Plots: TEMP from POP Dataset

Figure 5.8: Error (Means, Histogram, and Q-Q Plot) Comparison Using Cumulative Frequency Plots: VX from Cosmology Dataset

5.3.2 Accuracy Comparison with Different Sampling Methods

As we stated above, besides efficiency, accuracy is a very important consideration for a sampling method. Using visualization and clustering as representative data analysis applications, we not only evaluate the absolute accuracy of our method, but also compare the accuracy against three other methods.
The sampling methods we compare against are as follows. Simple random sampling involves randomly selecting a data subset out of the original dataset without focusing on any features. Stratified random sampling [33] performs random sampling within each stratum. Normally, the way these strata are formed can preserve spatial distribution of samples, but not the value distribution. KDTree-based sampling [128] has been proven to be a good method for visualization, and has also been applied to the cosmology dataset. It divides data into strata by building a k-dimensional tree over the dataset. The tree construction method is primarily based on spatial dimension(s) but can also consider data values as one dimension. Random sampling is performed within each stratum to generate a data sample. Because both data values and spatial distribution are considered in forming the strata, KDTree-based sampling has led to better accuracy than stratified random sampling.

In our method, which we will refer to as index sampling, we chose two bitmap indexing levels. The method we will denote as small bin corresponds to the use of low-level bitmap indices, which indicates fine-grained value distribution. The method we will denote as big bin corresponds to the use of high-level bitmap indices. Here, we groups 10 small bins into a big bin, and thus, value distributions are preserved only at a coarser level. The datasets and the variables used here are the same as the previous experiment: TEMP from the POP dataset and (VX, VY, VZ) from the cosmology dataset. The sample percentage is 0.1% of the original dataset.

It turns out that the appropriate error metrics for visualization and clustering are very distinct. Now we discuss the accuracy of the two applications separately.
Accuracy for Visualization

Characterizing the impact of sampling on visualization is hard, since human perception plays a role in how a dataset is viewed. Based on the existing literature from visualization [128], we used two types of error metrics: average metrics and absolute metrics. For average metrics, we used the following three indicators: means of the value over 200 separate sectors, histogram using 200 value intervals, Q-Q plot with 200 quantiles, and Signal-to-Noise Ratio (SNR). To make the results more obvious, we calculated the sector means, histogram, and Q-Q plot value of both the original dataset and each sample dataset, and computed the absolute value differences between the original dataset and the sample dataset. To represent these charts, we use a Cumulative Frequency Plot (CFP). In our plots (Figure 5.7 for example), a point \((x, y)\) indicates that the fraction \(y\) of all calculated absolute value differences are less than \(x\). Since the error metric value differences should be as small as possible, it implies that a method with the curve to the left has a better accuracy than the method with the curve to the right. Although average metrics are able to reflect the general accuracy of sample data, most localized error effects may be lost in the metrics due to averaging effects. On the other hand, absolute metrics, which defines the maximum errors, are able to help us analyze the worst case and localized the effect. Here we calculate the Kolmogorov-Smirnov (KS) value between original dataset and sample dataset as the indicator of absolute error. For the bitmap index sampling method, the total number of small bins of \(TEMP\) is 442, and the total number of small bins of \(VX\) is 670. Each 10 small bins are grouped into a big bin.

The left subfigures of Figures 5.7 and 5.8 show the absolute value differences of sector means using the five sampling methods (including two versions of our approach). The simple random sampling shows the worst accuracy. The stratified random sampling, which
considers spatial distribution, achieves better accuracy than simple random sampling. However, as it does not consider value distribution, the results are still worse than KDTree-based sampling and index sampling. If we compare KDTree-based sampling with index sampling, we can see that for POP data, index sampling (both small bin and big bin) achieves better accuracy than KDTree-based sampling. For cosmology data, KDTree-based sampling shows better accuracy than index sampling (big bin). However, index sampling (small bin) method still achieves the best accuracy.

The middle subfigures of Figures 5.7 and 5.8 show the absolute value differences for histogram entries, comparing the five sampling methods. KDTree-based sampling considers value distribution by treating variable value as one dimension during the KDTree sorting process. This method is more focused on spatial partitions and only considers value distribution at a very coarse level. Thus, as we can also see from the figures, for the cosmology dataset, the histogram results with KDTree-based sampling are not as good as our method. For the POP dataset, KDTree-based sampling and index sampling with big bin achieve a similar accuracy. Index sampling with small bin achieves a better accuracy than all the other methods.

The right subfigures of Figures 5.7 and 5.8 show the absolute value differences of Q-Q plot values among the five sampling methods. If we compare KDTree-based sampling with index sampling, we can see that for the POP dataset, KDTree-based sampling achieves the best accuracy, but for the cosmology dataset, index sampling (both small bin and big bin) shows better accuracy. On the whole, the Q-Q plot value differences between KDTree-based sampling and index sampling are small.

Table 5.1 shows the Signal-to-Noise Ratio (SNR) value of TEMP and VX. The signal is the baseline variance divided by the mean squared error, i.e. the noise. SNR approaches
infinity when there is no noise and negative infinity when there is no signal, i.e., the transformation doesn’t represent the “original” data as it approaches negative infinity. SNR performs a point-to-point comparison. To make it a useful indicator between original data and sample data, SNR is calculate based on the stratified mean values instead of points. From the table we can see that, for both TEMP and VX, both simple random sampling and stratified random sampling can not achieve good accuracy, as the SNR value is much smaller than the other three methods. If we compare KDTree-based sampling with index sampling, we can see that for TEMP, both index sampling (big bin) and index sampling (small bin) achieve better accuracy than KDTree-based sampling. For VX, KDTree-based sampling shows better accuracy than index sampling (big bin). However, index sampling (small bin) still achieves the best accuracy.

Table 5.2 shows the Kolmogorov-Smirnov (KS) value of TEMP and VX. This metric is derived from the two-sample K-S test, which is used to evaluate if two samples come from the same distribution. It measures the maximum difference in cumulative probabilities between two samples. From the table we can see that simple random sampling and stratified random sampling still can not achieve a good accuracy. For TEMP, KDTree-based sampling method achieve a similar accuracy as index sampling (big bin) method, and index sampling (small bin) achieves the best accuracy. For VX, both index sampling (big bin) and index sampling (small bin) achieve better accuracy than KDTree-based sampling method.

In sum, for both average error metrics and absolute error metrics, our method is able to achieve better accuracy than the other three methods in most cases.

**Accuracy for Clustering**

The error metric here is the difference between cluster centers, using the original and the sampled dataset. Specifically, we first calculate cluster center values for the original
Table 5.1: Error Comparison of Signal-to-Noise Ratio

<table>
<thead>
<tr>
<th>Sampling Methods</th>
<th>TEMP SNR</th>
<th>VX SNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Random Sampling</td>
<td>41.12</td>
<td>7.69</td>
</tr>
<tr>
<td>Stratified Random Sampling</td>
<td>48.69</td>
<td>10.18</td>
</tr>
<tr>
<td>KDTree-based Sampling</td>
<td>54.01</td>
<td>31.14</td>
</tr>
<tr>
<td>Index Sampling (Big Bin)</td>
<td>56.43</td>
<td>23.87</td>
</tr>
<tr>
<td>Index Sampling (Small Bin)</td>
<td>56.67</td>
<td>31.73</td>
</tr>
</tbody>
</table>

Table 5.2: Error Comparison of Kolmogorov-Smirnov

<table>
<thead>
<tr>
<th>Sampling Methods</th>
<th>TEMP K-S</th>
<th>VX K-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Random Sampling</td>
<td>0.0012</td>
<td>0.005</td>
</tr>
<tr>
<td>Stratified Random Sampling</td>
<td>0.00058</td>
<td>0.0031</td>
</tr>
<tr>
<td>KDTree-based Sampling</td>
<td>0.000098</td>
<td>0.00045</td>
</tr>
<tr>
<td>Index Sampling (Big Bin)</td>
<td>0.00014</td>
<td>0.00028</td>
</tr>
<tr>
<td>Index Sampling (Small Bin)</td>
<td>0.000013</td>
<td>0.000088</td>
</tr>
</tbody>
</table>

dataset, then calculate cluster center values for the sampled dataset, and finally compute the Euclidean distance between cluster centers in the original dataset and the sampled dataset. The dataset we used here is the cosmology data and the indices are built over the three attributes \( VX, VY \) and \( VZ \), i.e., the multiple attribute sampling method summarized in Section 5.2.4 is used here. The total number of multiple bins for \( VX, VY, VZ \) is 2000.

Figure 5.9 shows the accuracy using four sampling methods. The X axis shows different sampling percentages (25%, 12.5%, 1%, 0.1%), and the Y axis shows the average cluster center value differences. KDTree-based sampling considers sorting based on spatial information first and then values. In this case, this method sorts the data based on \( X, Y, Z \)
and then VX, VY and VZ. It achieves better accuracy compared with simple random sampling and stratified random sampling. Indices sampling method, which considers binning over VX, VY and VZ first and then spatial locality, achieves better accuracy than all the other methods. As data sampling percentage decreases, the advantage of our method becomes even more prominent.

To summarize our discussion, we can observe the following. Traditional methods from statistics, i.e., simple random and stratified random sampling, cannot get accurate samples as they are not considering enough features of the data. KDTree-based sampling, which is more focused on spatial locality, achieves good accuracy on sector means and Q-Q plots. However, the histogram result is not as good as for bitmap index sampling. Our method, which considers the value distribution first and then spatial locality, is able to generate a better histogram, while at the same time achieving good accuracy for sector means and Q-Q plots compared to KDTree-based sampling. It also achieves a better result than all the other methods when multiple attributes need to be considered while sampling. Furthermore, our method allows flexibility in choosing bin levels, and thus, users can adjust the bin size
and level to get the desired tradeoff between accuracy and efficiency. Finally, as we will elaborate later, another advantage of our method lies in its ability to pre-calculate error levels.

5.3.3 Error Prediction Accuracy

As we have stated throughout, an important and distinct feature of our approach is the ability to pre-calculate error levels. However, we need to verify if the predicted error results are close to the actual error results. We now describe results from an experiment designed for this purpose using the POP dataset. The sampling percentage is 0.1%.

In this experiment, we first calculate predicted error metrics with the methods described earlier in Section 5.2.2, then compute the actual error metrics by scanning over the entire sample dataset and compare the two sets of results. Figure 5.10 compares the predicted and actual errors for sector mean values, histogram and Q-Q plots, using the index sampling(small bin) method. The two sets of lines are either always or almost always identical, which shows that for index sampling(small bin) method, our error pre-calculation is able to accurately reflect actual error results.

Figure 5.11 compares the predicted and actual errors for sector mean values, histogram, and Q-Q plots, now using the index sampling(big bin) method. Here, we use the mean value as the representative value for each big bin. In the left figure (means), if we compare the predicted errors with the actual errors, we can see that there are only small value differences between the 60th sector and the 85th sector. In most cases, these two lines are identical. In the middle figure (histogram), we can see that there is some variation. This is because the index sampling with big bin method represents value distributions at a relatively coarse granularity. Each big bin can only be classified into one value interval in a histogram, but
each bin contains a value range and some values may belong to the neighboring intervals.

In the right figure (Q-Q Plot), again the differences are very small.

![Histogram and Q-Q Plot](image1)

Figure 5.10: Predicted and Actual Errors (Means, Histogram, and Q-QPlot): Small Bin

![Histogram and Q-Q Plot](image2)

Figure 5.11: Predicted and Actual Errors (Means, Histogram, and Q-QPlot): Big Bin

### 5.3.4 Efficiency Comparison with Different Sampling Methods

Earlier we have shown the benefits of sampling for improving the execution time when datasets are remote. However, so far we have not compared efficiency of our method against other methods. We now report such a comparison. Since a key feature of our approach is
error pre-calculation, we focus on a scenario where the samples must be generated so as to meet certain accuracy requirements. Thus, the total sampling time can be divided into two components: sample generation time and error calculation time. Moreover, with other methods, one may need to sample multiple times to obtain the right accuracy levels. The variable we use here is TEMP from the POP simulation, and the data size is 1.4 GB.

Figure 5.12: Time Cost Comparison across Sampling Methods

Figure 5.12(a) compares the sample generation time among the five sampling methods. The X axis shows different sampling percentages, (3.13%, 6.25%, 12.5%, 25%), and the Y
axis shows the execution time in seconds. We can see that simple random sampling takes the least time, which is not surprising. Stratified random sampling and KDTree-based sampling have similar sample generation time, each being somewhat slower than simple random sampling because of the time needed for generating strata. Another difference between stratified random sampling and KDTree-based sampling is that the latter requires $n \log(n)$ preprocessing time, which is not included here. In our method, the random sampling must be applied to each bitvector, which leads to higher time cost than the other three methods. This time depends upon the number of bins used. We can see that with the big bin method, which has one-tenth the number of bins compared to the small bin method, the time cost is only marginally higher than other methods. However, the index sampling (small bin) method has 1.19 to 3.98 times slowdown over KDTree-based sampling.

Figure 5.12(b) compares the error calculation time among the five sampling methods. With simple random sampling, stratified random sampling, and KDTree-based methods, we have to take a pass over the entire sampled dataset to perform error calculations. This is not only a high cost, but one that also increases with the size of the sampled dataset. In comparison, our method is able to pre-calculate error metrics based on bins (quite accurately, as we established earlier) before sampling. And the cost of performing the pre-calculation is not related to the sample size. From the figure, we can see that our method achieves at least 28x speedup compared with the other three methods while creating a 25% sample of the dataset. Note that these results are for a 1.4 GB dataset, and the advantage of our method will increase for larger sized datasets.

Figure 5.12(c) compares the overall efficiency among the five sampling methods. The X axis shows the resampling times, and the Y axis shows the total time cost in seconds. The sampling percentage is 6.25%. Because the first three methods cannot support error
prediction, the sample generation and error calculation process may have to be repeated multiple times until a satisfactory accuracy level is found. However, using index sampling, we can perform multiple error pre-calculations first (with different sampling levels) and then need only one round of sample generation. If we look at the first set of bars which correspond to the case where we sample only once, we can see that index sampling (small bin) method has a similar total cost compared with the other three methods, whereas the index sampling (big bin) method is significantly faster. However, if the sampling process needs to be repeated, both big bin and small bin methods are much faster than any of the other methods.

5.3.5 Data Sampling over Data Subsets

Another advantage of bitmap indexing is that it supports efficient subsetting over sub-sets of the original dataset, where these subsets may involve spatial (dimension-based) and/or value-based conditions. In this subsection, we show how our method is effective, i.e. data sampling efficiency improves if sampling is performed over a subset of values or spaces. Here we discuss value subsetting and spatial subsetting separately, although our method is able to support a combination of the two.

The left subfigure of Figure 5.13 shows the time incurred while sampling over different value-based subsets. The X axis shows the subsetting percentage, i.e. the fraction of the original dataset that meet the conditional predicate. The Y axis shows the sampling time, including both the index loading time and the sample generation time. The sampling rate is 25% in all cases, i.e. 25% of the data records that meet the conditional predicate are returned. From the figure, we can see that for both the small bin and big bin methods, the efficiency improves as the subsetting percentage decreases. Smaller value-based subset
implies not only smaller index loading time but also smaller sample generation time. Take the index sampling(small bin) method for example, sampling over 10% of the data takes 6.95 times less time than sampling over 100% of the data.

The right subfigure of Figure 5.13 shows the time cost of sampling with different spatial subsets. The X axis shows the spatial subsetting percentage and the Y axis shows the indices sampling time. The sampling percentage is still 25%. From the figure, we can see that the time cost decreases as the spatial subsetting percentage decreases, though the improvement is not as obvious as in the case of value subsetting. This is because for spatial subsetting, all indices still have to be loaded, so the only speedups are on the sample generation time.

5.3.6 Speedup with Parallel Sampling

This subsection describes how parallel indexing and sampling speeds up the entire sampling process. We compared the data sample generation time with different number of processes for both Big Bin and Small Bin methods. The number of processes is varied from 1
Figure 5.14: Scalability of Parallel Sampling with Different Process Number

to 32. The variable we used here is TEMP with 8 time steps, and the total data size is 11.2 GB. The sampling percentage is 25%.

Figure 5.14 evaluates the performance of parallel sampling with different number of processes. In this experiment, to emphasize the scalability of our method, we generated the equal number of index files as processes, with each process takes care of one index file which maps to one data block. The larger the number of processes we use, the smaller each index file is, and the faster the data sampling will perform. From the figure we can see that our method shows a good scalability as number of processes increases. Compared with 1 process, the relative speedup for Big Bin method varies from 3.55 to 18.64, and the relative speedup for Small Bin method varies from 3.63 to 20.38.

5.4 Related Work

Sampling of datasets has been widely studied, including work specific to scientific datasets and/or visualization.
Traditional statistical sampling methods [33], including simple random sampling and stratified random sampling, have been used often. We have performed a detailed comparison against these methods and demonstrated how our approach is more effective. KDTree-based sampling [128] uses a KDTree to divide data into sub-blocks and performs random sampling within each block. It needs to reorganize the entire dataset, with a time complexity of $O(n \log(n))$. We have also compared our method with this method, and shown that our approach outperforms this method in several aspects, and is comparable in other ways. The Z-curve order sampling method [85] involves a hierarchical indexing framework that uses a Z-order curve. However, it can only be applied to regular array-based datasets. Among the datasets we have used, this method will not even be applicable to the cosmology dataset. The WTSP Tree method [114] builds a wavelet-based time-space partitioning tree over large-scale time-varying datasets and supports multi-level data sampling on that. The entire dataset has to be reorganized and the WTSP Tree building process is time consuming.

Sampling has also been studied in the context of databases. One area of emphasis has been online aggregation, with initial work by Hellerstein et al [50]. Jermaine et al [53] proposed an online aggregation method for the DBO engine. Histograms [87] and wavelets [23] can be pre-computed and used. Chaudhuri et al [24] have conducted extensive studies on executing approximate aggregation queries using workload information and biased samples. More recent work in the database community has been in the context of speeding up map-reduce jobs with sampling. One initial study [46] proposed a framework to support incremental data sampling. EARL [68] involves a new sampling strategy with support for early error approximation based on bootstrapping, which has been widely employed in statistics and can be applied to arbitrary functions and data distributions. This
method is able to decrease the resampling times and achieve good accuracy. However, resampling is still needed to generate a satisfying sampling result.

Dissemination and analysis of large-scale and distributed datasets has been the focus of other studies as well. Some of the popular directions have been replica services [22, 30], reliable and predictable data transfers [10, 108], and constructing workflows [5, 40]. Chimera is a system for supporting virtual data views and demand-driven data derivation [44]. Metadata cataloging and metadata services have also been developed [41, 92]. The Metadata Catalog Service (MCS) [93] and Artemis [107] are collaborative components used to access and query repositories based on metadata attributes. Many middleware efforts have specifically focused on the needs of data-driven sciences [13], and enhancing and optimizing data transfer frameworks has been a popular topic [10, 63, 65, 73, 76, 60]. Our sampling techniques can work in conjunction with these efforts to make it feasible to analyze large-scale datasets.

5.5 Summary

This chapter has described a novel sampling method for massive scientific simulation datasets. We utilize the value distribution and spatial locality features of bitmap indices and have developed an accurate sampling method over multi-level bitmap indices. We also developed an error prediction mechanism to pre-calculate error metrics before sampling the data. Moreover, with the help of bitmap indexing, our method is able to support data sampling over any combination of value subset and dimension subset.

We have extensively evaluated our method with different types of datasets and applications. First, considering two applications, visualization and clustering, we have shown that server-side sampling can drastically improve the efficiency of analysis of remote datasets.
Next, we established that our method has much better accuracy than simple random and stratified sampling methods, and with respect to different metrics, either better or comparable performance to KDTree-based sampling. Yet another result is our error pre-calculation methodology gives very accurate estimation of error in sampled datasets. We have also analyzed the sample generation time with our approach, and have shown show that when error calculation time and the possibility of resampling to meet desired accuracy is included, our method outperformed other approaches. Finally, we show that we can combine our sampling method with value-based and/or dimension-based subsetting effectively.
Chapter 6: Supporting Correlation Analysis on Scientific Datasets in Parallel and Distributed Settings

Chapter 2 to Chapter 5 are focused on individual data analysis, where we support flexible server-side data subsetting, aggregation and sampling using bitmap indexing. This chapter will focus on improving correlation analysis efficiency using bitmap. We designed two indexing methods (Dynamic Indexing and Static Indexing) to improve the correlation analysis efficiency. We proposed two partition strategies for parallel correlation calculation. We also built an interactive framework to support correlation analysis over flexible subsets of the data.

6.1 Motivating Queries and Proposed System Interface

This section first introduces the popular correlation metrics, whose computations we are supporting. Next, we explain the query interface and the functionality we are intending to provide.

6.1.1 Background: Correlation Metrics

This subsection introduces several correlation metrics from information theory [37, 109, 110, 18] that have lately been used in scientific data analysis.
2-D Histogram: In statistics, a histogram is a graphical representation of the distribution of the data, or in other words, an estimate of the probability distribution of a continuous variable. A 2-D histogram reflects the value distribution of one variable regarding to the value changes of another, and is a useful metric to indicate the value distribution relationship between variables.

Shannon’s Entropy: In information theory, the information content of a random variable can be quantified by Shannon’s entropy [47]. Constant data (easily predictable) has a low entropy, while apparently random data (uniform probability) has a high entropy. Equation 6.1 shows the expression to calculate the Shannon’s entropy – here, $N_A$ represents the number of distinct values of the attribute $A$, and the probability distribution functions $P_A$ captures the probability of having each distinct value for $A$.

$$H(A) = -\sum_{j=1}^{N_A} P_A(x_j) \times \log_2(P_A(x_j))$$ (6.1)

Mutual Information: Mutual information [118] is the metric for computing the dependence between two random variables, and shows the amount of shared information between two variables in the number of bits. If the mutual information is low, then the two variables are independent. Conversely if mutual information is high, one variable provides information about the other. Equation 6.2 shows the expression to calculate the mutual information. Here, we index the data in the variables or attributes $A$ and $B$ by $j$ and $k$ separately, and use $x_j$ and $y_k$ to represent each distinct value. $N_A$ and $N_B$ represent the number of distinct values of each attribute, and three probability distribution functions, $P_A$, $P_B$ and $P_{AB}$, capture the probability of having each distinct value for $A$, for $B$, and for a pair of values of $A$ and $B$, respectively.
Figure 6.1: Example Showing Supported Query Interface

\[ I = \sum_{j=1}^{N_A} \sum_{k=1}^{N_B} P_{AB}(x_j, y_k) \times \log \left( \frac{P_{AB}(x_j, y_k)}{P_A(x_j)P_B(y_k)} \right) \]  

\[ (6.2) \]

6.1.2 Query Interface and Desired Functionality

We now show the functionality and the interface we intend to support, through an example shown in Figure 6.1. In the first step, the users are asked to input variable names for correlation analysis. In this example, users want to perform correlation analysis over ocean temperature (\( TEMP \)), salinity (\( SALT \)), and flow velocity (\( UVEL \)). In the following steps, users are able to input different queries that specify subsets of the data they want to perform correlation analysis on. Particularly, in the example shown, users want to first see the correlations among \( TEMP, SALT \) and \( UVEL \) when the ocean temperature is between 0 and 1 and the depth of the ocean is below 50 meters. After the query is submitted, the
system is able to calculate different correlation metrics among the variables directly and return the results (histogram, entropy, mutual information) as the output. Users are able to see the correlations and may be interested in further analysis. For example, a user may find that the mutual information between TEMP and SALT is much higher than that between TEMP and UVEL, which means TEMP and SALT are high correlated for this data subset.

Then, in the next step, they may want to further explore a subset of values for SALT, i.e., further specialize on the previous query to generate more specific correlations. This is supported efficiently through incremental analysis, which we will describe later. If users are not satisfied with the current result, our approach supports an undo operation to go back to the previous step. In this case, the system goes back to the second-last query. Thus, a new query can now specialize on top of the second-last query, and not the last query.

6.2 Algorithms for Correlation Analysis

This section describes correlation analysis algorithms we have developed. We start with a simple algorithm, explain its limitations, and then introduce bitmap indexing. We show a series of methods based on bitmap indexing, and then describe methods for execution in parallel and distributed settings.

6.2.1 Initial Method

A default algorithm that can be used for calculating any of the correlation metrics we listed comprises four steps. First, we need to load the entire data for the variables involved, say, variables A and B, into the memory. Next, if the correlation required is for certain subsets of the data, we take a pass through the data for the variables A and B to generate the data subsets based on queries. As a next step, we generate joint bins for A and B, i.e., first divide data for A and B into bins based on values, and thus generate $A_1, A_2, \ldots,$
$A_m$) and $(B_1, B_2, \ldots, B_n)$, and then generate joint bins based on individual bins and the dataset, which are \((A_1, B_1) \rightarrow \text{count}_0\), \((A_1, B_2) \rightarrow \text{count}_1\), \ldots, \((A_m, B_n) \rightarrow \text{count}_{mn}\), where $\text{count}_i$ is the number of elements located within the joint bin $i$. Finally, these counts are used to calculate the correlation metrics.

As one can see, the algorithm has a very high memory requirement. If the data corresponding to the two attributes cannot fit in memory, we need to orchestrate complex data movements, which can be very expensive.

### 6.2.2 Advanced Algorithm Using (Dynamic) Bitmap Indexing

We now describe an efficient method for computing correlations, which we refer to as dynamic bitmap indexing based method. This method requires single-variable indices (one bitmap index over one variable).
Correlation calculation between the variables $A$ and $B$ using bitmap indexing is shown in Figure 6.2. This method assumes that bitmaps have been constructed for each of the variables or attributes involved. The algorithm involves three steps. First, we directly find the subsets of bitvectors of the dataset for $A$ ($a_4$ to $a_i$, in the example) and $B$ ($b_0$ to $b_j$, in the example), which satisfy the current query and load them into the memory. Note that both dimension-based and value-based subsetting can be easily supported on bitmap indices, as they simply involve choosing certain rows and columns respectively from the bitvectors.

In the second step, we generate joint bins for $A$ and $B$. Because individual bins have been generated during the index generation phase and stored in the form of bitvectors, we only need to perform logic AND operations between bitvectors of $A$ and $B$ for this step. The total complexity of this step is $m \times n$, where $m$ and $n$ are the number of bitvectors (used in query) of $A$ and $B$. In the final step, we calculate different correlation metrics based on the joint bins, just like the original method.

If we compare the indexing based method with the original method, we find that there are at least three advantages of using bitmap indices. First, our method only needs to load the indices (which are much smaller in size) instead of loading the dataset into the memory. Compared with the original method, we have much smaller memory requirements and the time to load data is also reduced. Second, data filtering, especially for a value-based filtering condition, is very costly with the original method, as one needs to examine each element. However, our method achieves this step by simply loading the bitvectors that satisfy the current value-based condition. Finally, a key step of calculating correlation metrics is to generate joint bins. Without indexing support, the joint bins have to be generated by scanning through each element in the data subset. However, with the help of bitmap indexing, the joint bins is based on logic AND operations, and is much simpler.
One issue, however, is the cost of creating bitmap index on the data, which can be expensive. However, in many cases, an index may be created for a variety of reasons, like supporting data subsetting [97, 101] or sampling [99, 100].

6.2.3 Using Static Bitmap Indexing

A further optimization of the above method is based on the following motivation. Within most of the scientific datasets, some variables are highly correlated while others are not. Meanwhile scientists also have preferences on correlation analysis over specific variable set. Static indexing, which builds multi-variable indices (i.e., one bitmap index over multiple variables) involves higher up-front cost and storage, but can be used to answer queries for specific combination of variables efficiently.

Specifically, we note that the (dynamic) indexing based method described above still requires bitwise operations between the datasets of variables A and B to generate the joint bins. The purpose of static indexing is to further reduce the cost of bitwise operations by generating a more involved index, over multiple variables, as shown in Figure 6.3. During the bitmap index generation phase, instead of generating two separate bins (A and B), we
perform binning based on value subranges of both $A$ and $B$ (i.e., $A \cdot B$), and generate joint bitvectors based on the joint bins. In this example, the total number of joint bitvectors generated is 6. During the query process, we directly load the subset of joint bitvectors that satisfy the current query conditions into memory and generate joint bins by simply performing 1-bits counting operations over each bitvector. Based on that, the probability distribution of $A$, $B$ and $A \cdot B$ are generated and different correlation metrics are calculated. Moreover, static index can also be directly used to further calculate correlations between the current variable set and other variable or variable set, which is more efficient than dynamic indexing method.

If we compare static indexing with dynamic indexing, static indexing method has larger index generation and storage costs, but the advantage is that for each query, we can directly find the subsets of joint bitvectors and calculate correlation metrics efficiently based on that. In comparison, with dynamic indexing, there is a need to perform bitwise operations between the bitvectors for different variables for each query. Hence, static indexing is more
suitable for the cases where we know that certain variables are highly correlated, and/or it is known that the users will like to perform frequent correlation analysis on these variables.

6.2.4 Hierarchical Bitmap Indexing

Combining the benefits of both dynamic and static indexing, we have developed a hierarchical bitmap indexing framework to answer correlation analysis for scientific datasets. An example of this process is shown in Figure 6.4, where 11 variables are involved.

The entire process contains four steps. First, we build one bitmap index for each variable. These indices can be used for both individual data subsetting, as well as correlation analysis using dynamic indexing. Second, a simple clustering algorithm is applied over all variables to find and group those highly correlated variable pairs. In this step, an initial or approximate correlation values between each variable pair is calculated and each pair whose correlation results is larger than a certain threshold is clustered into one group. From the figures, we can see that $Var_1$ and $Var_2$ are clustered into one group, and $Var_5$ and $Var_6$ are clustered into another. In this step, users are also able to manually build up groups if they want to perform frequent correlation analysis between certain variables, e.g., $Var_9$ and $Var_{10}$. After clustering, in the third step, static indices, as described earlier, are built over those clusters that contain two variables. These indices can be used to process frequent queries more efficiently. Moreover, our framework is flexible enough that if users want to add another variable into an existing cluster (support correlation analysis between one variable and one variable set), we can simply perform bitwise operations between the index of the single variable and static multi-variable index to generate a larger static index, such as the one involving $Var_0$, $Var_1$ and $Var_2$ in this figure. Finally, a global metadata
is maintained to keep track of the entire hierarchy. When the system receives a query, the system can automatically find the right indexing method by looking up the global metadata.

6.2.5 Parallel Correlation Analysis

Our bitmap based sequential methods are the basis for the parallel and distributed algorithms we have developed. This section first describes two parallelization methods (dim-based partitioning and value-based partitioning). We assume that the dynamic indexing method is being used - parallelizing the method based on static indexing is similar but also simpler.

Dim-based partitioning (referring to dimension-based partitioning) is the more straightforward way of parallelization. Figure 6.5 shows the process of calculating correlation information between two variables in parallel using the dim-based partitioning method. The bitvectors for both variables A and B are first partitioned into 4 sub-index lists (A₀, A₁, A₂, and A₃; and B₀, B₁, B₂, and B₃) based on the dimensions. Each sub-index list
corresponds to one data sub-block of the original dataset. Suppose there are 4 worker nodes \((Node_0, Node_1, Node_2, and Node_3)\) in the parallel environment. Each node will be assigned with one sub-index of A \((A_i)\) and one sub-index of B \((B_i)\). For each correlation query, the bitwise operations between sub-index of A and B will be performed in parallel to generate the joint bins for data sub-blocks. After that, the joint bins will be sent to the master node, and the master node will calculate different correlation metrics based on that.

The advantage of this method is that it supports efficient indexing generation and analysis of each individual variable. In fact, during the index generation phase, instead of generating index-lists for the entire variable, the sub-index lists can be directly generated in parallel by different nodes. If subsetting conditions on individual variables are involved, they can also be easily applied. However, this method has a significant limitation during the stage when the correlations are computed. For several important correlation metrics such as the mutual information, the correlation results cannot be simply computed by taking counts from the sub-blocks of the data. Thus, each worker node must send the joint
bins to the master node, which adds a large load on the network. Finally, master node has to perform an expensive global combination operation to calculate the metrics.

In view of this, we have designed a value-based partitioning method to support efficient parallel correlation analysis, as shown in Figure 6.6. Instead of generating sub-index lists based on dimensions, this method generates sub-index lists based on the partition of bitvectors (values). For example, suppose the total number of bitvectors for the variables $A$ and $B$ is 14 each. After partitioning, each sub-index list ($A_0$ and $A_1$; and $B_0$ and $B_1$) will have 7 bitvectors each. Now, each of the 4 worker nodes can be assigned a pair of sub-index, i.e., four sets ($A_i, B_j$) are created, where each of $i$ and $j$ can be 0 or 1.

Now, each worker node can perform bitwise operations on the pair that is assigned to it, generate joint bins, and calculate the correlation metrics results directly in parallel. The results of each worker nodes will be sent to the master node and master node only needs to combine the partial results together, which has very low cost. We can see that though there is some replication of data, the total number of bitwise operations for each worker node remains almost the same. Compared with the dim-based partitioning method, it has a very low network transfer cost and correlation calculation cost. The only disadvantage is that during the index generation phase, each worker node needs to scan the entire data block to generate indices for corresponding value sub-range.

6.2.6 Correlation Analysis in a Distributed Environment

Our bitvector based approach also forms the basis for correlation computations on datasets that are spread over geographically distributed repositories. The key advantage of our approach lies in the use of bitmaps as a space efficient summary of the original dataset, which also happens to be sufficient for calculating correlation metrics.
We further assume the following. The data over which correlations need to be computed are across multiple repositories, each of which could hold different variables or different dimensional partitions. We assume that bitvectors have been generated and stored within each repository together with the dataset, and further, they can be subset within the repository. We assume that no computational cycles are available at each repository, and thus, correlation computations can only be performed in a compute cluster. Figure 6.7 shows the environment.

The correlation query that needs to be processed is initially submitted to the master node. The master node decides how the computation and downloading of the data will be divided among the worker nodes. Accordingly, subsets of available bitvectors are downloaded from the data repositories. Subsequently, the rest of the processing is just like parallel computation of bitvectors, which we have already discussed. The key advantage of the approach is that the amount of data downloaded from the repositories is significantly smaller, leading to overall reduction in the time required for computing the correlations.
6.2.7 Correlation Analysis over Samples

As dataset sizes are growing rapidly, analyzing the entire dataset is often not feasible. It turns out that an added benefit of bitvectors is that they allow sampling to be performed, in a fashion that value distribution is preserved in the sample.

We describe index-based sampling method and its application to correlation analysis using an example, which is shown in Figure 6.8. In this example, we still want to generate the correlation results between variable A and variable B. Instead of correlation analysis over all data elements, we want to perform correlation analysis over only 50% of the data. Bitvectors of variable A and B that satisfy the current query are selected and loaded into the memory. From the figure, we can see that this small dataset contains 32 elements, so each bitvector has 32 bits. The bitvectors of A are $a_6(0, 1), a_7(1, 2), a_8(2, 3), \ldots, a_i(9, 10)$, and the bitvectors of B are $b_6, b_7, b_8, \ldots, b_j$.

Though one can sample data corresponding to each variable and then perform correlation computation, obtaining representative samples for each variable (preserving the value-distributions) can be expensive in practice. Thus, we select only one variable, which

Figure 6.8: Sampling Using Bitvectors and Correlation Analysis
is the variable with a smaller number of bitvectors, and generate samples based on it. This allows sampling to be efficient, and yet, we get a favorable reduction in the computation time, while preserving accuracy, for the correlation computation step. The bitvector based sampling we perform on each variable is as follows. Our goal is to preserve distribution of values in each spatial region. For this purpose, we divide bitmap indices into spatial sectors. In the figure, we can see that the variable \( A \) is selected as the variable for sampling, and every selected bitvector of \( A \) is divided into 4 sectors, such that there are 8 bits within each sector.

After creating these sectors, sampling is applied within each sector for each bitvector. Particularly, within each bitvector, a desired fraction (sampling rate) of bits that are 1 are chosen. This ensures that the value distribution within each sector is maintained. In Figure 6.8, we are generating 50% samples out of original dataset. We can see that \( sa_6, sa_7, \ldots, sa_i \) are identifiers of data records that are in the sample generated, and only half of bits that have the value 1 are picked. For example, after sampling, the number of bits 1 in the sample bitvector \( sa_6 \) is 6, which is only half of that in original bitvector \( a_6 \). After that, the bitwise operations are performed between the sampled bitvectors of \( A \) and the original bitvectors of \( B \), following the same steps as in the dynamic indexing method based on the entire data. This method still reduces the time spent on creating joint bins and computing correlations, as we will show through experimental results.

### 6.3 Putting It Together: System Overview

The approach and algorithms discussed in the previous section have been put together in an interactive system. The system supports a high-level query interface, and can provide incremental analysis by maintaining bitvectors for the last several queries.
Figure 6.9: System Overview of Correlation Analysis

Figure 6.9 shows a high-level overview of our system, which can be divided into three modules: Query Analysis Module, Individual Query Module and Correlation Query Module. The Query Analysis Module takes structured queries (using an SQL-like syntax) as the input, generates internal query requests by parsing the query and analyzing the corresponding metadata. Individual Query Module supports flexible data subsetting and sampling over each variable [97, 99], as described in Chapter 2. Correlation Query Analysis supports the interactive correlation queries among multiple variables, which is the main contribution of this work. It contains five components:

1) Indexing Service: The bitmap indexing services, irrespective of whether they are static or dynamic indexing, take the query request as the input, perform indexing operations and generate joint bins across the variables involved as the output. The joint bins are used in the Correlation Calculation module to compute different correlation metrics.
2) **Sampling Service**: Sampling is used when the entire dataset cannot be analyzed in a timely fashion. This module performs data sampling directly over the bitvectors and outputs only a small sample of the data (in the form of *sampled joint bitvectors*). Then, the correlation calculation is performed based on this sample. As we described in Section 6.2.7, bitvector based sampling can generate “accurate” samples, and thus they provide the flexibility of accelerating correlation analysis with only a small sacrifice in the accuracy.

3) **Correlation Calculation**: This component calculates different correlation metrics (histogram, entropy, mutual information) based on the joint bitvectors (bins). One feature that is supported in our current implementation and has not been discussed so far is the use of *multi-level* indexing. A multi-level index uses a coarse-grained binning at the high-level level and a fine-grained binning at the lower-level. Subsequently, low-level bitvectors can be used to generate more accurate calculation of the correlations, though with additional time cost. In contrast, high-level bitvectors provide much faster response time, but generate correlation information in a coarse level.

4) **Support for Incremental Analysis and “Undo”**: Two important features of the system for supporting interactive analysis are - incremental analysis and an “Undo” operation. After users have seen results on a particular subset of data, they can further specialize in the query by adding another condition. Users can also do an “Undo” to step back to the earlier results, and specify additional conditions for incremental analysis on top of them. This functionality is supported efficiently by keeping the query bitvectors at each step (or a certain $k$ previous steps). On one hand, new subsetting conditions can be applied on top of the bitvectors stored from the previous steps. At the same time, because the size of bitvectors is much smaller than the dataset, we can store bitvectors from the last several steps without very high overheads.
5) **Data Reader:** Our system also allows users to view the actual data. Suppose during interactive correlation analysis, users specify a subset of data that turns out to be intriguing. Because our system records the bitvectors corresponding to this subset, users can obtain the original data corresponding to this subset efficiently.

### 6.4 Experimental Results

In this section, we report results from a number of experiments conducted to evaluate our correlation analysis approach and algorithms. We designed experiments with the following goals: (1) We compare the correlation analysis efficiency among the original *no indexing*, *dynamic indexing*, and *static indexing* methods in a sequential environment, and show that correlation analysis with the help of bitmap indexing can improve the efficiency. (2) We show the scalability of our parallel indexing method with the increasing number of nodes and compare the *value-based partitioning* method with the *dim-based partitioning* method. (3) We show that in an environment where data is stored on geographically distributed repositories, our method is able to speed up the correlation analysis process compared to a simple method that does not use any indexing. (4) We show that if correlation analysis is performed over samples, and not the entire dataset, what kind of speedup we can achieve and how much accuracy is lost.

The dataset we used here is generated by the Parallel Ocean Program (POP) [56], which is an ocean circulation model. The simulation we used has a grid resolution of approximately 10 km (horizontally), and vertically it has a grid spacing close to 10 m near the surface, increasing up to 250 m in the deep ocean. POP generates 1.4 GB output for each variable per time-slice. The total number of variables in the dataset is 26, and each variable
is modeled with either two dimensions (longitude and latitude) or three dimensions (longitude, latitude, and depth). The data is stored in the NetCDF format. The size of bitmap indices ranges from 12.1% to 26.8% compared to the size of its corresponding variable. The total number of bitvectors of each variable is from 203 to 431, depending on the value ranges of variables. We chose the same binning scales for the no indexing, dynamic indexing and static indexing methods so that the correlation results of all methods are same. Also, none of the reported results include the bitvector generation time, since once these indices have been calculated, they can be used for a variety of queries, not limited to correlation analysis (for example, subsetting [97] and sampling [99]). All of our experiments were conducted on the Glenn cluster from Ohio Supercomputing Center, where every node has 8 cores, 2.6 GHz AMD Opteron(TM) processors, with 64 GB RAM and 1.9 TB local disk space.

6.4.1 Efficiency Improvement Using Bitmap Indexing

The experiments in this subsection compare the correlation analysis time among the original or the no indexing method, the dynamic indexing method, and the static indexing method. Two variables were chosen and we calculate entropy of each variable and the 2D histogram and the mutual information between them.

Figure 6.10(a) shows the correlation analysis time among these three methods based on different queries. We selected 1000 queries with different dimension-based and value-based subsetting conditions, and then divide them into five categories based on data subsetting percentage (¡20%, 20%-40%, 40%-60%, 60%-80%, ¿80%). Each variable is 5.6 GB (4 timestamps merged together) in size. The time cost of the original or the no indexing method includes the data loading time, the data filtering time (where subsetting conditions
are applied), the joint bins generation time, and finally, the correlation metrics calculation time. Without indexing support, the entire datasets for the two variables involved have to be loaded into the memory, and subsequently, filtering conditions are applied to examine each data element. The joint bins are generated by binning over each element within the data subset, which is also time consuming.

The time cost of dynamic indexing includes bitvectors subset loading time, bitwise operation time to generate joint bins, and the correlation metrics calculation time. Compared with no indexing method, dynamic indexing method has much lower data loading time, as the size of the bitvectors is much smaller than the size of the data block, and only a subset of bitvectors that satisfy the current query conditions need to be loaded into the memory. Moreover, the joint bins are generated based on fast bitwise operations. This is reflected in the experiments, and from the figure, we can see that, irrespective of the data subsetting
percentage, the *dynamic indexing* method always achieves better efficiency than the *no indexing method*. The speedup factor varies from 1.78x to 3.61x, becoming smaller as data subset percentage increases. This is because the larger the subsetting percentage is, a larger fraction of bitvectors have to be loaded into the memory, and more bitwise operations need to be performed. The time cost of the *static indexing* method includes only the joint bitvectors loading time and the correlation metrics calculation time. Compared with *no indexing* method, the speedup is from 11.4x to 15.35x.

Figure 6.10(b) shows the correlation analysis time among three methods over different sizes of the data. Here the correlation metrics calculation is over the entire data blocks of two variables, without any subsetting. The size of the dataset for each variable ranges from 1.4 GB to 11.2 GB. From the figure we can see that even without data subsetting, for all different cases, both *dynamic indexing* and *static indexing* methods perform well, with their advantage even increasing as the dataset sizes increase. This is because our indexing based methods require less memory.

### 6.4.2 Scalability of Parallel Indexing

The experiments in this subsection show the speedup of parallel correlation analysis using multiple nodes. During this evaluation, we also compare the efficiency of *value-based partitioning* method with the *dim-based partitioning* method. Correlation analysis is performed over two variables, and the data sizes for each is 28 GB (20 timestamps merged together). We use between 1 and 32 nodes for our experiments, and because the method is memory-intensive, only 1 core per node is used.

Figure 6.11(a) shows the scalability of our parallel correlation analysis method with different number of nodes. The X axis shows number of nodes (from 1 to 32) used. The
correlation calculation here is over the entire data blocks (all elements), i.e., no subsetting condition is involved. From the figure we can see that both value-based partitioning and dim-based partitioning methods show good speedup as number of nodes increases. For the value-based partitioning method, the speedup using 2, 4, 8, 16, and 32 nodes is 1.87x, 3.4x, 4.53x, 7.68x, and 11.79x, respectively. For dim-based partitioning method, the speedup using 2, 4, 8, 16, and 32 nodes is 1.73x, 2.77x, 3.18x, 4.32x, and 5.96x, respectively. The reason for higher efficiency of value-based partitioning is because the master node is a bottleneck for dim-based partitioning. Note that the speedups for value-based partitioning are still not close to linear. This is because different bitvectors can have different number of 1s, which leads to different amounts of time for the bitvector operations.

Figure 6.11(b) shows the efficiency of both partitioning methods with different queries, which are then classified with respect to the subsetting percentage involved. The number of nodes here is 16. From the figure we can see that for both methods, the execution time
increases as data subset percentage increases, as we expect. However, if we compare the two partitioning methods, we can see that the relative improvement from the value-based partitioning method becomes more significant than the dim-based partitioning method as data subsetting percentage increases (relative improvement ranges between 1.17x to 1.58x). The reason is that as data subsetting percentage increases, the number of joint bins generated by each process also increases, which imposes a more significant network overhead for the dim-based partitioning method.

### 6.4.3 Efficiency Improvement in Distributed Environment

The experiments in this subsection analyze the efficiency of performing correlation analysis in a distributed environment, where data is stored over geographically separated data servers, and analysis is performed over a single cluster. In such a case, without bitmap indexing, datasets corresponding to variables involved need to be downloaded to the cluster used for the computations. Instead, if the indices have been generated and stored together with the dataset, only the index files need to be downloaded to the cluster. In our experiment, we use 16 compute nodes for parallel correlation calculation. We use a local data server (1 Gb/sec bandwidth connection to the compute-cluster) and a remote data server (200 Mb/sec bandwidth connection to the compute-cluster) in separate sets of experiments.

Figure 6.12(a) compares the total time between the dynamic indexing method and the no indexing method using local data server. The data size of each variable ranges from 7 GB to 28 GB. No subsetting is involved. The total execution time with either of the methods can be divided into two parts: the network data transfer time and the parallel correlation analysis time. As expected, our method reduces the data transfer time, because only bitvectors are being downloaded, and not the full dataset. For parallel correlation analysis, to make
a fair comparison, both methods used value-based partitioning, i.e., without indexing, the full dataset is partitioned on the basis of the values. While partitioning bitvectors based on values is trivial, there is a processing cost associated with partitioning the dataset based on values. We do not report the extra value partitioning time for the no indexing method. We can see that our method is still able to achieve much better efficiency than the no indexing method, because our method has much smaller memory data loading cost and the computation costs for joint bins are also lower. The overall improvement ranges from 1.87x to 1.91x.

Figure 6.12(b) compares the total time between the dynamic indexing method and the no indexing method using a remote data server, with lower (200 Mb/sec) bandwidth. From the figure, we can see that while the parallel correlation analysis time of both methods is similar (as in the previous experiment), our method further improves the efficiency by saving the data transfer time. The overall speedup ranges from 2.78x to 2.96x.
It should be further noted that our experiments did not include any data subsetting. If correlation queries involve any subsetting condition, the size of the bitvectors involved can be reduced further. In comparison, a data repository without any indexing support may not allow any subsetting, and users may have to download all the data for each variable being analyzed.

6.4.4 Efficiency and Accuracy Comparison with Sampling

Our last set of experiments compares the efficiency and accuracy of performing correlation analysis with different sampling levels (sampling applied to bitvectors). We selected 10 variables (each has three dimensions with 1.4 GB in size) from the POP dataset, and calculated the mutual information between each distinct pair of them. The total number of such pairs is 45. Dynamic indexing is used in these experiments.

![Time Cost](a) Time Cost  
![Accuracy](b) Accuracy

Figure 6.13: Efficiency and Accuracy of Correlation Analysis with Different Sampling Levels
Figure 6.13(a) shows the efficiency of calculating mutual information for 45 variable pairs with different samples. The X axis shows different sample percentages (100%, 50%, 25%, 10%, 5%, 1%) compared with original data size, and the Y axis shows the entire correlation analysis time. Overall, we can see that the entire analysis efficiency greatly improves as sample size becomes smaller. Compared with correlation analysis over the original dataset (100%), the speedup using 50%, 25%, 10% 5%, and 1% samples is 1.34x, 1.69x, 2.17x, 2.93x, and 6.84x, respectively. While the steps for constructing joint bins and calculating the metrics are accelerated by almost the same factor as the sampling level, the cost of reading the original bitvectors remains almost the same, and an additional cost of sampling is introduced.

Figure 6.13(b) shows the accuracy of mutual information results using different sampling levels. Here we generated 45 mutual information results (45 pairs) for each sampling level. We compute the relative value differences using expression \((original\_result − sample\_result)/original\_result\) for each pair. Then, we use a Cumulative Frequency Plot (CFP) to represent the relative mutual information differences for all pairs. In CFP, a point \((x, y)\) indicates that the fraction \(y\) of all calculated relative value differences are less than \(x\). Because the value differences should be as small as possible, it implies that a method with the curve to the left has a better accuracy than the method with the curve to the right. From the figure we can see that the accuracy using 50% samples is very close to the original dataset, as errors are very close to 0 for almost all points. Not surprisingly, accuracy becomes worse as sparser samples are taken. If we calculate the average accuracy lost based on the CFP, we find that the average accuracy loss with 50%, 25%, 10%, 5% and 1% sample is 1.53%, 3.42%, 7.91%, 12.57%, and 18.32%, respectively. Because our bitvector based sampling method preserves distribution of values, reasonably high accuracies are
maintained even with a small sample of data. Overall, our sampling method provides a way to accelerate the computations while obtaining reasonably accurate results. Sampling can also provide a way to further reduce the data transfer volumes when the data is stored in geographically distributed servers. In fact, as our previous work has shown [99], bitmap indexing can be used to perform server-side sampling of the data.

6.5 Case Studies Demonstrating Efficacy of Tool

All results presented in the previous section focused on demonstrating the efficiency of the methods. In this section, our goal is to demonstrate how the tool is useful to the scientists. Particularly, we show that the ability to compute correlations over various subsets through an intuitive and high-level query interface makes our tools extremely valuable for scientific discoveries.

The studies were conducted at the Los Alamos National Laboratory (LANL), using datasets generated from the POP simulation. Among the two scientists, one is a physical chemist interested in studying bio-geochemistry data, such as the production (“PROD”) and sinking (“FLUX-IN”) of Silicate (SiO2), Calcium Carbonate (CaCO3), and Particulate Organic Carbon (POC), as well as Oxygen (O2) production (“PROD”) and consumption (“CONSUMP”). The second scientist focused on studying the relationship between Temperature (TEMP) and Salinity (SALT), which are the two basic elements of the ocean data. Using our system, they are able to explore different correlations over different areas or value ranges by submitting different queries, and our system returns correlation metrics results (histogram, entropy, and mutual information) as the output.
Figure 6.14 shows different correlation results our system presented to the scientists. The description below focuses on showing (in intuitive terms) the observations that can be made from the data.
Subfigure 6.14(a) shows the histogram of $SALT$ based on different value ranges of $TEMP$. Let us first look at the $SALT$ value along the $X$ axis. Here we define $p$ as the value on the $X$ axis where the curve has the highest $Y$ value. When $TEMP < 5$, the $p$ value of $SALT$ is around 0.0349. However, as the temperature increases, the salinity decreases. When $TEMP >= 5$ and $TEMP < 10$, the $p$ value decreases to 0.0342. After that, the salinity increases as the temperature increases. When $TEMP >= 10$ and $TEMP < 15$, the $p$ value increases to 0.0345, and then when $TEMP >= 15$, the $p$ value of $SALT$ increases to 0.035. Thus, salinity is high when the temperature is either low or high. After talking to the scientists, we find that this is because on one hand, the deeper into the ocean, the colder the water gets, and the water also gets denser, which implies a higher salinity. On the other hand, the surface salinity also increases in areas close to the equator, because of hotter air. This happens because water evaporates faster, leaving more salt to a smaller amount of water. Moreover, if we look at the $Y$ axis, we are able to see the diversity of the salinity within different temperature ranges. For example, when $TEMP < 5$, over 23% of the data elements’ value is around 0.035. However, when $TEMP >= 15$, only around 6% of the data elements’ value is around 0.035.

Subfigure 6.14(b) shows the histogram of $SALT$ based on different areas of the ocean (dimension subsets). From the figure we can see that the $p$ value of $SALT$ within the Atlantic Ocean and the Mediterranean Sea areas is larger than the $p$ value within the Pacific Ocean and the Indian Ocean, which reflects the actual features of these areas.

Subfigure 6.14(c) shows the histogram of $SALT$ based on different value ranges of $TEMP$ inside the Mediterranean Sea. This figure is used to show that our system is able to perform correlation analysis over flexible combinations of dimension subsets and value subsets. From the figure we can see that although the value sub-range of $TEMP$ is the
same as in subfigure 6.14(a), the relationship between TEMP and SALT has changed: particularly, now salinity increases as the temperature increases. This is because this area is warm.

Subfigure 6.14(d) shows the histogram of sink volumes of Silicate SiO\textsubscript{2}\textsubscript{FLUX İN} based on different sink volumes of Calcium Carbonate CaCO\textsubscript{3}\textsubscript{FLUX İN}. We can see that the p value of CaCO\textsubscript{3}\textsubscript{FLUX İN} increases as SiO\textsubscript{2}\textsubscript{FLUX İN} increases. This is because more sinks of Silicate implies more plankton in this area, and more plankton implies more Calcium Carbonate generated and more sinks of Calcium Carbonate. Hence, 2D histogram is able to help scientists find different value distributions of one variable based on value changes of another.

Subfigure 6.14(e) shows an example of using entropy for correlation analysis. The goal here is to see the changes of entropies of SALT with respect to the different value subsets of TEMP. A total of 30 queries are submitted and each query specifies a TEMP subset with similar value intervals, which makes the entropy values of TEMP similar. From the figure we can see that while the entropy values of TEMP are similar, the entropy values of SALT have large differences. When the TEMP value is around one centigrade, entropy of SALT is the lowest, and it increases as the TEMP value increases. In another word, the value of salinity is more constant and predictable as temperature is around one centigrade, whereas one sees a diversity of salinity values as the temperature increases.

Finally, subfigure 6.14(f) shows the mutual information between TEMP and SALT based on the value distribution of TEMP. From the figure, we can see that mutual information between TEMP and SALT is high when the temperature values are either low or high. This implies that TEMP and SALT are highly correlated within these two value
ranges. For the other value ranges, where the mutual information is close to 0, correlation is very small between the two variables.

Overall, the representative results above show that correlation analysis over flexible subsets of data can help scientists confirm known facts, and even make new observations.

6.6 Related Work

Our work has some similarities to a number of efforts from high performance data management area, as well as visualization.

Closely related to our work, Fastbit [129] and FastQuery [32] apply bitmap indexing and parallel indexing to support efficient value-based subsetting (for individual variables). Our work builds on top of these, but is unique in applying bitvectors for correlation analysis. There has also been a growing trend towards building database-like functionality on top of native storage of the data. Examples include the NoDB approach [9] and automatic data virtualization [125]. Our work is an example of this approach, but, again, unique in its focus on correlation analysis. In recent years, many Array DBMSs, including SciDB [20] and RasDaMan [14] have been designed, and are gaining popularity. None of these systems have provided support for correlation analysis across variables.

Analysis of multiple variables and their relationships in scientific simulation outputs has been an ongoing topic of research. In a very recent work, Biswas et al. [17] have presented an information theoretic framework for exploring multivariate datasets in a “top-down” manner, where they divide the variables into groups based on their information overlap, and then identify representative variables from each group to conduct further relationship analysis. Wang et al. [115] used information theory for exploring the causal relationship among the variables of a time-varying multivariate dataset. In an earlier work, Jänicke
et al. [52] applied the dimensionality reduction on the high dimensional data where each dimension was analogous to a variable in the multivariate context. Another well-known multivariate exploration technique was developed by Di Yang et al. [134]. They use a Nugget Management System (NMS), where the nuggets represent the information that the users are interested in. Several authors have surveyed existing multivariate data analysis techniques [127, 39]. Almost none of these efforts have focused on scalability limitations, especially, what happens when the data does not fit into memory, or if the data is stored in geographically distributed repositories. Parallelization of the methods is another challenge that has not been addressed.

6.7 Summary

While the potential of data-driven discoveries for scientific advances is being increasingly recognized, several trends are making interactive and efficient data analysis hard. On one hand, the amount of data generated by scientific simulation (or instruments) is rapidly increasing. On the other hand, computing environments are becoming more and more constrained with respect to data movement (at all levels).

This chapter has focused on the problem of correlation analysis in parallel and distributed settings. We have developed a series of techniques, with the main underlying idea that bitmap indices can serve as a concise and representative summary of the original dataset, allowing computation of the correlation metrics more efficiently. Our algorithms have been incorporated in a system that offers a high-level API, where users can interactively choose subsets of the data to be analyzed, and sampling can be combined with correlation analysis.
We have extensively evaluated our system. We have compared the correlation analysis efficiency between the traditional method and our method in a stand-alone environment and have shown that our dynamic indexing method can achieve a speedup from 1.78x to 3.61x, and the static indexing method can achieve a speedup from 11.4x to 15.35x. We have also demonstrated the scalability of the parallel indexing method and compared the efficiency between two different parallelization methods. We have shown that in a distributed environment where computing resources and data storage are geographically separated, our method can further improve the efficiency compared (with the traditional method) with a speedup from to 1.87x to 2.96x. Also, our results show that if correlation analysis is performed over samples of the data (e.g., 25%), we can achieve a speedup (1.69x) with only a small accuracy loss (3.42%). Finally we have conducted a user-evaluation with domain scientists at the Los Alamos National Laboratory and demonstrated how our system is able to aid the data-driven discovery process.
Chapter 7: In-Situ Bitmap Index Generation and Efficient Data Analysis based on Bitmaps

In this chapter, we introduce our in-situ bitmap index generation and efficient data analysis using bitmaps method. In an in-situ environment, after the data is simulated and before it is written to the disk, if we can find a method to reduce the data volume while still support different kinds of analyses, it will greatly improve the efficiency. In this work, we propose a novel method that utilizes bitmap index \((\text{bitmaps})\) as a representation of the data and showed that many kinds of analyses can be supported by bitmaps without requiring the original dataset. Each time after the data is simulated in memory, we generate bitmaps. Subsequently, a number of different analysis steps are performed on bitmaps. Then, instead of writing the data, we only write the bitmaps (much smaller in size) back to disk, which greatly improves the I/O efficiency. Besides identifying bitmaps as an appropriate summary structure for scientific data, we have also demonstrated how to use bitmaps for both online (in-situ) analysis (e.g., time-steps selection) and offline analysis (e.g., correlation mining) efficiently. We demonstrated that even though our method takes extra time for index generation, we still can achieve better efficiency with the help of multi-core. We also showed that both online and offline analyses efficiency can be improved with the help of bitmap indexing. Finally, we compared in-situ sampling method with our method and
showed that our method can achieve better accuracy and even better efficiency in certain scenarios.

7.1 In-Situ Bitmaps Generation

This section first analyzed the advantage of using bitmap index as data summarization and then describes our in-situ bitmaps generation method.

7.1.1 Summarizing Spatial Data with Bitmaps

The following observations can be made with respect to bitmap index and their suitability as a summary structure for multi-dimension arrays. There is no binning (and thus no loss of precision) with respect to dimensional attributes, unlike almost any other method, such as a histogram. This turns out to be a very important advantage, especially for any application where spatial precision is critical. The value distribution is also maintained (though this is also true for histograms). At the same time, because each point is represented by a single bit, compression is used, continuity of dimensional attributes is exploited, and due to the hardware support for bit operations, we can achieve space and time efficiency. Note that like other methods discussed above, there is binning with respect to value-based attributes, but it seems unavoidable when space efficiency is important.

The other advantages of bitmaps are as follows:

- Bitmaps are much smaller in size compared with the original dataset. In most of the cases, the size of bitmaps is less than 30% in size compared with the original dataset, which improves both data I/O and memory usage.
Many kinds of analyses can be executed purely using bitmaps without touching the original dataset. In our previous work, we demonstrated that approximate data aggregation, data spatial join, correlation query, incomplete data analysis and subgroup discovery can be supported using bitmaps without touching the original dataset [3, 4, 98, 122, 123]. In this work, we will further show that time-steps selection and correlation mining can be supported using bitmaps.

Bitmaps can be generated efficiently in an in-situ setting with acceleration using multiple cores. More importantly, we observe that newer architectures have a large number of cores, which can be used to generate bitmaps without very high overheads. Moreover, the cost of bitmaps generation is easily offset by reduction in data movement times.

7.1.2 In-Situ Bitmaps Generation

Recently, several efforts [62, 66] demonstrate that the bitmap index generation speed can be greatly improved in an in-situ setting. Compared with these works, the main contribution of our work is that we proposed different core allocation strategies for parallel bitmaps generation and discussed the advantage and disadvantage of each strategy. We also designed a WAH-based in-place bitmaps compression algorithm which has very small memory cost and hence is suitable for in-situ setting.

Figure 7.1 shows the process of generating distributed bitmaps from the data output using an ongoing simulation, and using multiple cores on a single node. The same algorithm can be applied if the data is simulated in a cluster environment where each machine simulates a portion of the data. A Core Allocation Strategy is pre-defined, and is used to decide how to allocate cores for simulation and bitmaps generation. Using the strategy, a
set of cores ($core_0$ to $core_m$) is assigned for data simulation, while the remaining cores are responsible for bitmaps generation. Each simulation core will simulate a portion of the data and write the data into memory. After a time-step of data is ready, the data will be logically partitioned into $(n - m)$ sub-blocks based on a pre-defined partitioning algorithm. Then, each core among the cores assigned for bitmaps generation will access a sub-block of the data in memory and generate a series of compressed bitvectors. This way, the bitmaps generation is performed in parallel without having any dependency among different cores. After the bitmaps are generated, the original data can be discarded because bitmaps contain enough information for users to do certain kinds of online (e.g., time-step selection) and offline analysis (e.g. correlation mining), which will be described in the next two sections.

In our work, two cores allocation strategies (Shared Cores and Separate Cores) are designed. The Shared Cores method assigns all the available cores for both data simulation
and bitmaps generation. Each time after one time-step of data is generated (using all cores),
the simulation program will be paused and all the cores will be switched for bitmaps gener-
ation. After that, the simulation continues to generate next time-step data when the bitmaps
for current time step are ready. The Separate Cores method divides the available cores into
two separated sets. One set is always used for data simulation and another is responsible for
bitmaps generation. In this case, the data simulation and bitmaps generation is performed
in parallel, and a data queue is shared between simulation and bitmaps generation. Each
time when a new time-step data is simulated, it will be added to the tail of the data queue
if the queue is not full (the queue size is limited by the memory capacity). And if the data
queue is not empty, the data will be dequeued from the head of the queue and the cores
for bitmaps generation will generate bitmaps based on the data. Using this method, the
number of cores allocated between simulation and bitmaps generation becomes very im-
portant. Ideally, the average data simulation speed should be close to the average bitmaps
generation speed after the core allocation. In our work, we use Equations 7.1 and 7.2 to
decide core allocation. For each simulation, we first use an initial set of cores to run simu-
lation and bitmaps generation and calculate the average simulation time \( T_{\text{simulate}} \) and
bitmaps generation time \( T_{\text{bitmap}} \). Then the number of cores of allocation depends on
the ratio between \( T_{\text{simulate}} \) and \( T_{\text{bitmap}} \).

\[
Core_{\text{simulate}} = Core_{\text{total}} \times \frac{T_{\text{simulate}}}{(T_{\text{simulate}} + T_{\text{bitmap}})} \quad (7.1)
\]

\[
Core_{\text{bitmap}} = Core_{\text{total}} - Core_{\text{simulate}} \quad (7.2)
\]
Algorithm 5: Generate_Bitmap(Data, DataSize, BinNum)

1: id = 0;
2: allocate space for Segments with size BinNum;
3: allocate space for Result with size BinNum;
4: for i = 0; i < DataSize; i+ = 31 do
5: initialize elements in Segments to 0;
6: for j = 0; j < 31 AND j + i < DataSize; j+ = 1 do
7: VectorID = MapValueToID(Data[id + j]);
8: Segments[VectorID] = Segments[VectorID] | (1UL << (30 − j));
9: end for
10: for j = 0; j < BinNum; j+ = 1 do
11: &LastSeg = Result[j].back()
12: if Segments[j] == 0xffffffff then
13: if (LastSeg AND 0xc0000000) == 0xc0000000 then
14: LastSeg+ = 31;
15: else
16: Result[j].push_back(0xc0000001);
17: end if
18: else if Segments[j] == 0 then
19: if (LastSeg AND 0xc0000000) == 0x80000000 then
20: LastSeg+ = 31;
21: else
22: Result[j].push_back(0x80000001);
23: end if
24: else
25: Result[j].push_back(Segments[j]);
26: end if
27: end for
28: end for
29: Return Result
For in-situ analysis, besides the efficiency, another important factor is the memory limitation. We do not want bitmaps to occupy a huge amount of memory and thus affect the simulation process. From subsection 3.1 we know that bitmaps before compression can be much bigger than the data itself. Hence, it is certainly unacceptable that we first generate all uncompressed bitvectors and then do a one-time compression. Instead, we developed an algorithm which performs in-place bitvector generation and compression. The basic idea of this algorithm is to fetch data segment by segment, generate bitvectors for each segment and merge them into existing compressed bitvectors (we only need to scan the data once to generate bitmaps this way). After compression, the size of bitmaps is generally less than 30% compared to the size of the dataset. Thus, this method has very small memory overhead. Moreover, because our goal is to only keep bitmaps instead of data for certain kinds of analysis, we can keep freeing the memory of the data we have already scanned and use it to hold bitmaps we generated. This way, the available memory actually keeps increasing as bitmaps is generating. Algorithm 5 shows the pseudocode of this method.

The variable \( id \) is the index to iterate through the data block, variable \( Segments \) contains the uncompressed bitvectors for the current segments, and variable \( Result \) contains the compressed bitvectors for all previous segments. The for loop in the line 4 shows that we generate the bitvectors segment by segment (we use WAH compression, where every 31 elements forms a segment). Line 5 to line 9 shows the process of generating uncompressed bitvectors (\( Segments \)) for current segment. \( Segments \) is a 2D bit array (the first dimension is the number of bitvectors, the second dimension is 32 bits stored as one integer) and initially every bit is set to 0, as shown in line 5. Line 7 maps each element in current segment into corresponding bins (bitvectors) based on its value. Line 8 sets the corresponding bit of the mapped bitvector to 1 based on element position. Line 10 to line 28 shows the
process of merging current uncompressed bitvectors (Segments) into existing compress bitvectors (Result). Line 11 fetches the last compressed bitvector unit from Result[j]. If current Segments[j] contains all 1-bits (0x7FFFFFFF), and if the previous compressed bitvector unit is a fill word containing 1-bits (0xC0000000), we update the last compressed unit by adding 31 new 1-bits, as shown from line 12 to line 14. Otherwise, we add a new fill word (0xC000001F) containing 31 bits into Result[j], as shown from line 15 to line 17. Line 18 to line 23 shows the merging process when Segments[j] is a fill word containing all 0-bits (0), and the logic is similar as 1-bit fill word. If Segments[j] contains a mix of 0 and 1 bits, we treat it as a literal word, and directly add it to the tail of Result[j], as shown from line 24 to line 26.

7.2 Online Analysis Using Bitmaps

Our claim is that bitmaps can be an effective summary of the original data. To this end, we show how some of the key data selection steps can be performed using only bitmaps (and after original data has been discarded). Performing such a step using only bitmaps greatly reduces memory requirements, a critical issue on today’s accelerators and for upcoming architectures.

The specific data selection step we have implemented using bitmaps is time-step selection. Specifically, for most of the simulations, the data is simulated time-step by time-step. Certain time-steps contain more important information over others, and can be one of the representative time-steps for a post analysis step. Thus, the goal of a time-step selection algorithm is to find $K$ of the $N$ given time-steps that represent the evolution of the phenomenon.
7.2.1 Importance Driven Time Steps Selection

Importance driven time-steps selection has been well studied [116, 106, 69, 115]. The key question is how to measure the *importance* of a time-step – usually, the *importance* of a time-step is determined in two aspects: First, the output for the time-step itself may contain a high amount of *information*. Second, the time-step may convey a distinct type of information with respect to the other time-steps. Several correlation metrics from the information theory [37, 109, 110, 18] such as *Earth Mover’s Distance, Shannon’s Entropy, Mutual Information* and *Conditional Entropy* are applied to quantify the above two aspects.

The next question is selecting a subset of time-steps given a fixed number of time-steps. Wang et al [116] proposed a greedy algorithm, where the main steps are: 1) partition the time-steps into *intervals*, 2) calculate the *correlation* between the previous selected time-step (the one of the previous interval) and each time-step within this interval, and 3) select
the time-step with the minimum correlation. Now, the key consideration is generating these partitions. One obvious method will be fixed length partitioning, i.e., each partition contains the same number of time-steps. However, more sophisticated method like information-volume based partitioning can also be used, where the time-steps within each partition contain the same total amount of accumulated importance values. Note that the above method is a greedy method, and there are other possibilities as well. For example, Tong et al [106] proposed a method that uses dynamic programming. Though bitmaps can be used to accelerate computation of almost any such algorithm, we have implemented a greedy algorithm, because efficiency is the most important consideration.

Figure 7.2 shows the process of supporting time-steps selection using both full data and bitmaps. We first look at the full data method. From the figure we can see that, there is a total number of \(2m + 1\) time-steps, which are divided into three intervals. The first interval only contains one time-step \(T_0\), while the second and third interval contain \(m\) time-steps each. Initially we always choose the first time-step as the preselected time-step. To find the time-step in the second interval with minimum correlation regarding to the first time-step, we need to calculate different correlation metrics between the preselected time-step \(T_0\) and each time-step in the second interval \(T_1\) to \(T_m\). After the calculation, \(T_m\) contains the minimum correlation regarding to \(T_0\). Hence, we only keep \(T_m\) for the second interval and discard others. Then we use \(T_m\) as the preselected time-step and continue to select the time-step with minimum correlation in the third interval, so on so forth. This way, we are able to keep time-steps with maximum information regarding to each other. If we look at the bitmaps method, we can see that the calculation process is similar. However, instead of using the original data which is big in size, all the correlation metrics can be calculated using bitmaps. Because bitmaps is much smaller in size, this method has much smaller
memory cost. And in the following subsection, we will also show that the calculation of different correlation metrics can be performed more efficiently using bitmaps.

The algorithm we have accelerated using bitmaps uses two important correlation metrics, which are the Earth Mover’s Distance and Conditional Entropy. The calculation of Conditional Entropy, in turn, is based on two other metrics, which are Mutual Information and Shannon’s Entropy. The definition of Shannon’s Entropy and Mutual Information can be found in Section 6.1.1. To be able to describe how these metrics can be computed using bitmaps, we first give mathematical definitions of these metrics, and how they are computed using original data.

Earth Mover’s Distance: The earth mover’s distance (EMD) is the measure of the distance between two probability distributions over a region $D$ – it can be viewed as the cost of changing one distribution of the data to another distribution of the data. Let us say that we are looking at the difference of distribution between two time-steps $A$ and $B$. Then, $EMD$ is calculated by first dividing elements of attribute $A$ and $B$ into bins based on values (the binning range of different time-steps should be the same), then checking how many elements are different within each bin between $A$ and $B$, and finally, taking a cumulative sum of these differences together, as shown in Equation 7.3. Here $CFP(j)$ indicates the differences between variable $A$ and variable $B$ when element values are smaller than the value range of the $j$th bin. And EMD simply sums each $CFP(j)$ together.

$$EMD = \sum_{j=1}^{N} CFP(j),$$
$$CFP(j) = CFP(j - 1) + Diff(Bin(A_j), Bin(B_j)),$$
$$CFP(0) = 0.$$  
(7.3)
**Conditional Entropy**: Conditional entropy is defined with the consideration of both self-contained information and the information with respect to others. It is calculated by Shannon’s Entropy minus Mutual Information, as shown in Equation 7.4. The bigger the conditional entropy value of $A$ is, the more information $A$ contains with respect to $B$.

\[ H(A|B) = H(A) - I(A; B) \] (7.4)

### 7.2.2 Time Steps Selection Using Bitmaps

This subsection describes how to select time-steps more efficiently using bitmaps.

*Earth Mover’s Distance (EMD)* is summed based on CFP of each bin, and each CFP is calculated using the number of elements that are different within each bin pair (bins of two time-steps with the same value range). Hence, the calculation of EMD becomes the calculation of different number of elements for each bin pair between two time-steps.
are two methods to calculate the element differences. The first method is to simply count
the number of elements of each bin pair and compute the count differences between them.
The second method is to compute the element differences for each bin pair considering
spatial information. Because most of the scientific data is built over multi-dimensional
spaces, calculating elements differences considering their spatial positions is necessary in
many cases. For each bin pair, we need to scan each data element inside one bin and find if
there is a match at the same position of another bin.

Figure 7.3 shows the process of calculating EMD using bitmaps. Without bitmaps
support, we have to scan both time-steps $T_i$ and $T_j$ to classify the data elements into bins.
Then we have to compare the elements of $T_i$ and $T_j$ inside each bin pair and compute the
total number of elements that are different. Both steps are time-consuming for big scale
data. From the figure we can see that with the help of bitmaps, we already have bitvectors
(bins) of both $T_i$ and $T_j$. For the first difference calculation method, we only need to
perform 1-bits count operations for each bitvector pair of $T_i$ and $T_j$. Because each 1-bit
inside the bitvector indicates that there is one data element inside current bin, if we choose
the same binning scale as using the full data, we can get the same element differences
for each bin pair and hence the same EMD result. For the second difference calculation
method, we only need to perform $m$ number of bitwise $XOR$ operations between $T_i$ and
$T_j$ to calculate the element differences of each bin pair. Logic $XOR$ operation sets the
target bit to 1 if the same position of the two inputs has different bit values (one bit is 1
which means there is a data element in current position of one bin, and another bit is 0
which means there is no matching element for the same position of another bin). Hence, it
can quickly compute the different number of elements between bin pairs without scanning
through each element inside the original data. Also we are able to generate same element
Conditional Entropy is calculated by Shannon’s entropy minus mutual information, which are computed using the individual value distribution of time-steps $T_i$ and $T_j$, and the joint value distribution of $T_{ij}$. Here the key step is to generate the joint value distribution. Figure 7.4 shows the process of calculating Conditional Entropy using bitmaps. Without bitmaps support, we need to scan through each element of both $T_i$ and $T_j$ at least twice to decide the binning scale and then generate individual and joint value distribution. Using bitmaps, because each bitvector corresponds to one value or value range, the individual value distributions of $T_i$ and $T_j$ are already generated during the bitmaps generation process, which saves the effort. From the figure we can see that, using bitmaps, we only need to perform fast bitwise AND operations between $m$ bitvectors of $T_i$ and $m$ bitvectors.
of $T_j$ to generate all the joint bins, which is efficient. Logic $AND$ operation sets the target bits to 1 only if the same positions of both inputs contain 1-bit values, which means that each time we count 1 for current joint bin $T_{ij}$ only if both the value of $T_i$ and the value of $T_j$ in the same position satisfy the value range of current joint bin. Hence, the joint value distribution generated by bitmaps is the same as that generated by the full data, choosing the same binning scale. With the support of bitwise operations, bitmaps method can achieve much better efficiency, especially for those datasets with relatively small cardinality of the data.

The efficiency advantage of using bitmaps depends on the cardinality of the data. Suppose the data size is $N$, and the number of bitvectors (bins) is $M$. For most of the simulations, it generates a huge amount of data ($N$), but the data cardinality is fixed and relatively small ($M << N$), which makes bitmaps really efficient in the time-steps selection. Another advantage of this method is that because bitmaps are much smaller in size, we are able to keep more number of time-steps in memory and achieve better selection accuracy. All those kept bitmaps will be saved for further offline analysis such as correlation mining.

### 7.3 Offline Analysis Using Bitmaps

We can use bitmaps (only) to perform different kinds of offline analyses. In our previous work, we have demonstrated that we can perform approximate data aggregation, data spatial join, correlation query, incomplete data analysis and subgroup discovery using bitmaps [3, 4, 98, 122, 123]. In this section, we will focus on correlation mining.

#### 7.3.1 Motivation for Correlation Mining

There is often a need for complex analyses over datasets generated by scientific simulations. Such analyses can be classified into two categories: individual variable analysis
and correlation analysis. Individual variable analysis involves analysis over each variable or attribute independently, and can take the form of data subsetting, data aggregation, data mining or visualization. Much of the existing work, especially in data visualization, has focused on individual variable analysis. However, more recently, several efforts [17, 115] have focused on studying the relationship among multiple variables and making interesting scientific discoveries based on such analysis.

In our previous work [98], we proposed an interactive framework to support correlation queries over multiple variables. Using our tool, users can submit different SQL queries to specify the data subsets (either value-based or dimension-based subsets) they are interested in for correlation analysis. Our tool is able to calculate correlations of the subsets among multi-variables and return the result to users. With the help of bitmaps, the entire process can be executed very efficiently.

However, one open challenge remaining in this work is that the users often do not know what subsets could provide interesting correlation results. Thus, there is a need for a correlation mining algorithm, and not just correlation query processing.

### 7.3.2 Correlation Mining Using Bitmaps

In this work, we focus on correlation mining between two variables. Mutual Information, discussed in the previous section, is used here as an indicator of correlations between two variables. Our goal is to find the data subsets (either value or spatial subsets) with a high value of mutual information.

\[
I(A; B) = \sum_{j=1}^{N_A} \sum_{k=1}^{N_B} I(A_j; B_k), \quad I(A_j; B_k) \geq 0. \tag{7.5}
\]
\[ P_A(x) = P_A(1)(x) = P_A(2)(x); \]
\[ P_B(y) = P_B(1)(y) = P_B(2)(y); \]
\[ P_{AB}(x, y) = \frac{P_{A1B1}(x,y) + P_{A2B2}(x,y)}{2}, \quad P_{A1B1} > P_{A2B2}; \]

\[ I(A; B) < I(A_1; B_1). \]  \hfill (7.6)

In our solution, the mining process for value-based subsets and spatial-based subsets follows different orders. If we analyze the formula of mutual information in subsection 7.2.1, we can see that mutual information is accumulated based on the probabilities of value subsets. If the mutual information of a bigger value range is low, all its value subsets must contain lower mutual information. As shown in Equation 7.5, the mutual information \( I(A, B) \) is summed by the mutual information of each value subset \( I(A_j; B_k) \). Because \( I(A_j; B_k) \) is non-negative, we can see that \( I(A, B) \) is always larger than or equal to the value of any subset \( I(A_j; B_k) \). Hence, we use a top-down method for efficient value-based subsets mining, exploiting multiple levels of bitmaps. However, the situation is different for spatial subsets. We cannot draw the same conclusion if a bigger spatial area contains low mutual information. This is because it is highly possible that one spatial sub-area contains high mutual information while other sub-areas contain low mutual information, which make the final result low. A simple example is shown using Equation 7.6. Here we divide the data into two sub-spatial areas. To simply the scenario, we assume these two sub-areas contain the same number of elements and also the same individual value distributions for both variable \( A \) and \( B \) \( (P_A(x) = P_A(1)(x) = P_A(2)(x), P_B(y) = P_B(1)(y) = P_B(2)(y)). \) The difference is that one sub-area has bigger joint value distribution than another \( (P_{A1B1} > P_{A2B2}). \) In such case, we can easy see that while the individual value distribution is the same, the joint distribution of entire area \( P_{AB} \) value is the average of two sub-areas \( P_{A1B1} \) and \( P_{A2B2}. \) Because the mutual information is proportional to the joint value distribution, and \( P_{AB} \) is smaller
than $P_{A_1B_1}$, we can easily find that the mutual information of $I(A;B)$ is smaller than its sub-area $I(A_1;B_1)$. Hence, for spatial subsets, we follow a bottom-up order (spatial-based partitions) to calculate mutual information.

Without bitmaps, we have to manually divide the entire dataset into a huge number of value and spatial units and then calculate the mutual information between each unit pair, which is extremely time-consuming. With the help of bitmaps, although we follow different orders for value and spatial subsets mining, we do not need to process them separately. Figure 7.5 shows the process of correlation mining using one-level bitmaps, which comprises three steps. In the first step, we generate joint bitvectors based on logic AND operations between bitvectors of $A$ and $B$. We also count the number of 1-bits for each
joint bitvectors during the bitwise \textit{AND} operations. The total number of bitwise operations are $i \times j$, where $i$ is the number of bitvectors of variable $A$ and $j$ is the number of bitvectors of variable $B$. The second step is a pruning step. Although we generate $i \times j$ number of joint bitvectors, the 1-bits are distributed sparsely over the joint bitvectors. A pruning step is necessary to filter those joint bitvectors with small correlations. Here we calculate the mutual information for each joint bitvector and filter the joint bitvector if the result is smaller than a threshold $T$. The rule to decide $T$ here is that even if all the 1-bits of this joint bitvector is located within the same spatial unit, we still consider it as uncorrelated. In such a case, we assume all elements inside this joint bitvector (corresponding to the value subsets of two variables) are uncorrelated and further processing is not required. In the third step, for the remaining joint bitvectors, we partition each bitvector into basic sub-spatial units and generate the 1-bits distribution over all units. We keep only those units with mutual information larger than another threshold $T'$ and filter the others. All the remaining units are considered as spatial subsets with high correlations. Hence, with the help of bitmaps, we only need the bitwise \textit{AND} and \textit{1-bits Count} operations to detect all the interesting value or spatial subsets with high correlations.

Algorithm 6 shows the pseudo-code of correlation mining between two variables. Line 1 to line 3 iterates each bitvector of variable $A$ and $B$, perform logic \textit{AND} operation between each bitvector pair and generate joint bitvectors. Line 4 calculates the mutual information for the current joint bitvector. If the current mutual information is larger than a threshold (shown in line 5), we iterate each spatial unit of the joint bitvector and calculate the mutual information within each spatial unit (line 6 to line 11). Finally those spatial units with mutual information larger than another threshold is kept in the final result. All the other value and spatial units are treated as uncorrelated.
Algorithm 6: Correlation-Mining$(bitVectorsA, bitVectorsB)$

1: for each vector$A$ in $bitVectorsA$ do  
2:     for each vector$B$ in $bitVectorsB$ do  
3:         jointBitVector = LogicAND(vector$A$, vector$B$);  
4:         valueMutualInfo = CalMutualInfo(vector$A$, vector$B$, jointBitVector);  
5:         if valueMutualInfo $\geq$ THRESHOLD1 then  
6:             for unitBegin = 0; unitBegin < dataSize; unitBegin+ = unitSize do  
7:                 spatialMutualInfo = CalMutualInfo(jointBitVector, unitBegin, unitSize);  
8:                 if spatialMutualInfo $\geq$ THRESHOLD2 then  
9:                     AddResult(unitBegin, unitSize, jointBitVector);  
10:             end if  
11:         end for  
12:     end for  
13: end for

There are two optimizations in our work. For multi-dimensional dataset, to keep the multi-dimensional feature of each spatial unit, we use the Z-order curve [85] to iterate over the dataset during the index generation process. This way, when we partition the joint bitvectors, the basic spatial unit is the size of the smallest unit of Z orders. We can also choose different Z order granularity to match our efficiency goals. Moreover, because usually bitmaps are constructed at multiple levels, the higher level bitmaps contains smaller number of bins with bigger bin ranges, whereas, it is the other way around for the lower level. For efficiency consideration, we begin with high-level bitmap indices to quickly filter the low correlated value subsets. Then we only look at the low-level bitvectors belonging to the high-correlated bitvectors of high-level indices.

7.4 Experimental Results

In this section, we report results from a number of experiments conducted to evaluate our in-situ bitmaps generation and bitmaps based data analysis algorithms. We designed a
series of experiments with the following goals: (1) We compare the in-situ analyses time cost and memory consumption between bitmaps and the full data method, and show that our method can achieve better efficiency and less memory cost for different simulations on different multi-core systems. (2) We compare the efficiency of two cores allocation strategies (Shared Cores and Separated Cores) and show that a good core allocation strategy can generate bitmaps efficiently. (3) We show that bitmaps can also improve the data analyses efficiency in a parallel in-situ scenario. (4) We show that bitmaps based correlation mining (offline analyses) has much smaller time cost than the full data method. (5) We compare the efficiency and accuracy between in-situ sampling and in-situ bitmaps methods and discuss the scenarios where in-situ bitmaps is a better method than in-situ sampling.

We use two simulation programs and one scientific dataset for our experiments. Heat3D simulation [1] is developed to estimate the effect of different geologic structures on heat flow. A 3D mesh can be created to simulate heat flows. The variable generated by the Heat3D application is temperature. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (Lulesh) simulation [58] is an application which calculates the motion of materials relative to each other when subject to forces. It approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It is built on the concept of an unstructured hex mesh, and a node on the mesh is a point where mesh lines (edges) intersect. The variables of each node include coordinates, forces, velocity vector, and acceleration. Lulesh simulation is a complex simulation with more time and memory cost (to hold mesh lines). The dataset we used for correlation mining is generated by the Parallel Ocean Program (POP) [56], which is an ocean circulation model (the simulation code for POP was not unavailable to us for execution). The reason we use this dataset here is it contains multiple variables and
some of them have strong correlations within either the value or spatial subsets. POP has a grid resolution of approximately 10 km (horizontally), and vertically it has a grid spacing close to 10 m near the surface, increasing up to 250 m in the deep ocean. The total number of variables in the dataset is 26, and each variable is modeled with either two dimensions (longitude and latitude) or three dimensions (longitude, latitude, and depth). The data is stored in the NetCDF format.

For our single node experiments, we use two machines with large number of cores. One machine contains 32 Intel Xeon x5650 CPUs and 1 TB memory, and is located at the Ohio Super-computing Center (OSC). The second machine (Intel MIC) contains 60 Intel Xeon Phi coprocessors and 8 GB memory. Both these machines were chosen because they have a large number of cores on each node, a characteristic that will be common in the near future. In addition, the Intel MIC node has a small amount of memory per core, which is another expected feature for nodes of future supercomputers. For experiments in a distributed memory (cluster) environment, we used 32 nodes with 12 Intel Xeon x5650 CPUs and 48 GB memory from Oakley Cluster of OSC.

### 7.4.1 Efficiency and Memory Cost Comparison for In-Situ Analysis

The experiments in this subsection compare the efficiency and memory cost of in-situ analysis between the *full data* method and the *bitmaps* method.

Figures 7.6(a) and 7.6(b) compare the in-situ analysis efficiency between *full data* and *bitmaps* methods for Heat3D simulation. In our experiment, the simulation generates 100 time-steps and we select 25 time-steps from them using the *Fixed Length Partitioning* method, as described in Section 7.2.1. The selection metric used here is *conditional entropy*. Figure 7.6(a) shows execution on machine with the Intel CPU and 1 TB memory.
Figure 7.6: Select 25 Time Steps over 100 Using Different Simulations and Systems

The dimension scale of Heat3D is $800 \times 1000 \times 1000$ and the output data size per time-step is 6.4 GB. Figure 7.6(b) shows execution on the Intel MIC with 8 GB memory. Because of the memory limitation, the dimension scale here is set as $200 \times 1000 \times 1000$, and the output data size per time-step is 1.6 GB.

Heat3D simulation has regular temperature changes but the number of distinct values is relatively low. The number of bitvectors (bins) we used ranged from 64 to 206, depending the temperature range of different time-steps. The binning scale is set to retain 1 digit after the decimal point for each element. The execution time reported for the *bitmaps* method...
includes the simulation time, the in-situ bitmaps generation time, the time-steps selection time using bitmaps, and the time spent outputting bitmaps for the selected time-steps. The execution time reported for the full data method includes the simulation time, the time-steps selection time using full data, and the output time for the data from the selected time-steps.

From Figure 7.6(a) we can see that the bitmaps method has the additional bitmaps generation step, and it has a larger time cost than full data method if we only use 1 or 2 cores. However, it greatly reduces the time for time-steps selection and data writing. With the help of bitmaps, we achieved a speedup from 1.38x to 1.50x for time-steps selection (we support fast bitwise operations instead of scanning through 12.8 GB data for each calculation), and achieved a speedup around 6.78x for data writing (the bitmaps size is much smaller than the data size) compared with the full data method. The critical observation is that as an increasing number of cores are used, both bitmaps generation time and time-steps selection time is reduced almost linearly, whereas the data output time remains the same. From the figure we can see, the data writing time becomes the major bottleneck after we use 4 cores. If we consider the total execution time, our method can achieve a speedup from 0.79x to 2.37x compared with the full data method. More broadly, considering that nodes in current and future HPC machines have large number of cores but limited I/O bandwidth, we can see the clear advantage of bitmaps based approach.

Figure 7.6(b) compares the efficiency between the two methods on Intel MIC. Intel MIC contains more number of cores for computing, but the I/O bandwidth is even lower. As a larger number of cores are used, we are able to achieve a significant speedup. As a result, our method can achieve a speedup from 0.81x to 3.28x compared with the full data method on MIC.
Figures 7.6(c) and 7.6(d) compare the in-situ analysis efficiency between full data and bitmaps method for the Lulesh simulation. We select 25 time-steps out of 100 time-steps using the Fixed Length Partitioning method, and the selection metric here is Earth Mover’s Distance. Lulesh is a more complex simulation which generates nodes and edges among nodes. Each node is formed by four variables Coordinates, Force, Acceleration and Velocity, and each variable contains data value in three scales X, Y and Z. There are a total of 12 data arrays (each array size is equal to the number of data nodes) for each time-step, and we support in-situ analysis based on all of them. Figure 7.6(c) shows execution on the machine with Intel Xeon and 1 TB memory. The total number of nodes generated by Lulesh is 64 MB and the total data size for analysis is 6.14 GB (Edges are used to help calculate nodes value, so we do not include edge data here). Figure 7.6(d) shows execution on Intel MIC with 8 GB memory. Because of the memory limitation, the total number of nodes is 8 MB here and the data size for analysis is 768 MB. The number of bitvectors (bins) ranges from 89 to 314, depending upon the number of distinct values in the output at each time-step.

From Figure 7.6(c) we can see that, compared with Heat3D simulation, Lulesh is a more complex simulation and has much larger simulation time. For all the cases, the bitmaps generation time and the time-steps selection time is much smaller than the simulation time. Moreover, we are able to achieve a good speedup for time-steps selection using Earth Mover’s Distance. We only need to perform hundreds of XOR operations instead of scanning 12.28 GB data for each calculation. The speedup compared with full data method for time-steps selection is from 3.45x to 3.81x. As was the case with Heat3D simulation, our method achieves better efficiency as we use more cores, and the speedup
ranges from 0.84x to 1.47x. Figure 7.6(d) compares the efficiency between the two methods using Intel MIC. As we had observed in our experiments with the Heat3D simulation, because the MIC system has a larger number of cores and slower disk I/O speed, we are able to achieve even better speedup compared with the full data method. The speedup here is from 0.92x to 2.62x.

Figure 7.7 compares the memory cost between the two methods. In this experiment we kept 10 time-steps in memory for selection. The major memory cost for Heat3D using full data method is 1 previous selected time-step, 1 intermediate time-step (designed by the simulation) and 10 current time-steps. In comparison, the memory cost using bitmaps method is 1 intermediate time-step, 1 current time-step (to simulate next time-step), 1 previous selected bitmap, and 10 current bitmaps. Because bitmaps have much smaller size than the original data, the bitmaps method requires 3.59x (6.4 GB data) and 3.39x (1.6 GB data) smaller memory than the full data method. Lulesh is a more complex simulation. Beside the node storage cost, a large amount of memory is used to store the edges, which
are used to calculate nodes. However, even with the extra memory cost, our method still has much smaller memory cost, which is 2.02x (6.14 GB) and 1.99x (0.76 GB).

7.4.2 Comparing Core Allocation Strategies

The experiments in this subsection compare the simulation and bitmaps generation time between two core allocation strategies, which are the shared cores and separated cores strategies. Recall that the shared cores method involves assigning all the available cores to simulation and bitmap generation in sequence, whereas the separated cores method involves dividing the total cores into two sets and assigning one set to simulation and another to bitmaps generation.

![Core Allocation Methods](a) Heat3D - CPU  
(b) Heat3D - MIC  
(c) Lulesh - CPU

Figure 7.8: Efficiency Comparison between Shared Cores and Separated Cores Strategies

Figure 7.8 shows the execution times with these two strategies for Heat3D simulation and Lulesh simulation using Intel CPU and Intel MIC systems. Sub-figure 7.8(a) compares the total time cost (simulation time plus bitmaps generation time) over 100 time-steps for the Heat3D simulation. The total number of cores used here is 28 (Intel CPU), and the data size is 6.4 GB. From the figure, we can see that the left bar \( c_{all} \) indicates the shared
cores method. All 28 cores are first used for simulation and then bitmap generation, and the process is repeated. The right bar group \((c_i, c_j)\) indicates the separated cores method. Here \(i\) is the number of cores used for simulation, and \(j\) is the number of cores used for bitmaps generation. With the allocation \(c_{12}, c_{16}\), which indicates that we use 12 cores for simulation and 16 cores for bitmaps generation, the average time cost of simulation and bitmaps generation is the closest to each other and hence we are able to achieve the best efficiency. As indicated previously, Heat3D is a very simple numerical program, which also does not scale well with increasing number of cores. Thus, best results are obtained when more cores are dedicated to bitmap generation than the simulation itself. The time cost of \(c_{\text{all}}\) is larger than \(c_{12}, c_{16}\). This is because of limited scaling of the simulation program – for example, the speedup is only 1.3x when we use 28 cores compared with 12 cores. This shows that it is not necessary to use all the cores for simulation. When we have a large number of cores, we can use the extra cores for bitmaps generation, which will not have a significant impact on the simulation speed.

Sub-figure 7.8(b) compares the total execution time using Heat3D simulation on Intel MIC system (using 56 cores). The data size is 1.6 GB. From this figure we can see that when we allocate 32 cores for simulation and 24 cores for bitmap generation \((c_{32}, c_{24})\), we can achieve the best efficiency. Again time cost of \(c_{\text{all}}\) allocation is bigger than \(c_{32}, c_{24}\). Sub-figure 7.8(c) compares the total time cost using Lulesh simulation on Intel CPU system. We still use 100 time-steps here and the data size is 6.14 GB. Recall that Lulesh simulation has much bigger simulation time. From the figure we can see that, \(c_{20}, c_{8}\) achieved the best efficiency, which means we only need to use 8 cores for bitmap generation. Since most simulation programs involve a large amount of computing, we can expect to see good results by dedicating only a small fraction of the resources for bitmaps generation.
7.4.3 Efficiency Improvement in Parallel In-Situ Environment

The experiment in this subsection shows that our bitmaps based in-situ analyses method can scale quite well in a cluster setting. Moreover, in clusters or distributed memory machines where data has to be stored to disks that are not local to the node, we show how the bitmaps based method further improves efficiency.

Figure 7.9 shows the scalability of our method compared with the full data method. Same as the previous experiments, the time cost of full data method includes the simulation time, the time-steps selection time, and the data I/O time (either writing data to local disk or transferring data to a remote server). The time cost of bitmaps method includes the simulation time, the bitmaps generation time, the time-steps selection time, and the data I/O time (writing or transferring bitmaps instead). Here we used 32 computing servers from Oakley Cluster of OSC and 1 remote data server with around 100 MB/sec bandwidth connected to the compute cluster.
The simulation used here is Heat3D, which requires communication (MPI) among machines to update the boundary information. The data size is 6.4 GB. The number of machines varies from 1 to 32, and each machine uses 8 cores for parallel processing. We generate a total of 100 time-steps and select 25 out of them. The selection metric is Conditional Entropy. From the figure we can see that, although our method has an extra step for bitmaps generation, we are still able to achieve better speedup than the full data method. Our method saves both time-steps selection and data I/O time cost, and it also scales quite well. The Bitmap - Local and Full Data - Local methods directly write the data or bitmaps to the local disk of each machine. We can see that our method achieved a 1.24x to 1.29x speedup here. In many cases, the simulated data from multiple computing servers need to be sent to one remote data server for storage, which makes our method achieve further speedup as we have much smaller data transfer size. Using the network with around 100 MB bandwidth, the Bitmaps - Remote method can achieve a 1.24x to 3.79x speedup compared to Full Data - Remote method.

7.4.4 Efficiency Improvement for Offline Analysis

In this experiment, we compares efficiency of correlation mining between bitmaps and full data method. Here we used the ocean dataset generated by the Parallel Ocean Program (POP).

Figure 7.10 compares the two methods using different data sizes. The two variables we used here are temperature and salinity. The data size of each variable varies from 1.4 GB to 11.2 GB. From the figure we can see that for all the cases, the bitmaps method has smaller execution time than the full data method. This is because of a number of reasons. First, our method has much smaller data loading time and memory cost compared with the full data
method because bitmaps are much smaller in size. Moreover, the mining process of our method involves performing $m \times n$ bitwise AND operations ($m$ is the number of bitvectors of temperature and $n$ is the number of bitvectors of salinity), filtering the uncorrelated units, followed by 1-bits count operations based on partitioned spatial units, and calculating the mutual information. Both logic AND operations and 1-bits count operations can be executed very efficiently. On the other hand, the full data method needs to reorganize the entire data into small value and spatial units and then do an exhaustive calculation within each unit (The same filter rule is also applied to full data method, but it still has much bigger calculation cost). Also multi-level indices can further improve the efficiency, while building different scales of value range based on original data is time consuming. As a result, our method is able to achieve a 3.83x to 4.91x speedup compared with the full data method. The larger the dataset size, the better speedup we can achieve. It is also important to note that there is no accuracy loss compared with the full data method. This is because both methods use the same binning scale to calculate mutual information. Note that number

![Figure 7.10: Efficiency Improvement for Correlation Mining](image-url)

...
of bins can be varied with either of the methods, but larger (smaller) number of bins will lead to higher (lower) execution time for both methods.

### 7.4.5 Comparing Bitmaps and Sampling

Bitmaps can be viewed as a data reduction method. One simple approach for data reduction is sampling – i.e., simply selecting a smaller number of output elements for further processing. The experiments in this subsection compare the efficiency and accuracy between in-situ sampling and in-situ bitmaps based analyses. The machine we used here are Intel CPU with 32 cores.

Figure 7.11 compares the execution time between *bitmaps* and *sampling* method. We used Heat3D simulation here, which selects 25 time steps out of 100 and the size of each time step is 6.4 GB. The number of cores used here is 32. The *Bitmaps* method includes simulation, bitmaps generation, time-step selections using bitmaps and outputting bitmaps. The *Sampling* method includes simulation, down-sampling, time-steps selection using sample and sample writing.
The left bar shows the time cost of bitmaps method and the right bar group shows the time cost of sampling method with different sampling levels (from 30% to 1%). From the figure we can see that the sampling method, which only selects a subset of the data from the original dataset, can generate different sample levels efficiently, while bitmaps generation, which requires binning and compression, has much bigger time cost than sampling. Moreover, the data size after sampling is effectively reduces, which also greatly improved the efficiency for the time-steps selection. However, from the figure we can see that the bitmaps method still achieves better efficiency than the sampling method using 30% samples. This is because the major bottleneck here is still the disk I/O speed. With the help of multiple cores within the node, we can significantly reduce the time differences for data processing (bitmaps generation or sampling) and time-steps selection between these two methods.

Figure 7.12: Accuracy Lost of In-Situ Sampling Method
Besides the time cost, another big advantage of bitmaps method is that, using bitmap indexing for time-steps selection and correlation calculation does not incur any accuracy loss. In comparison, sampling involves loss of accuracy. Figure 7.12(a) shows the accuracy loss with different sampling levels, when sampling is performed before time-steps selection. For bitmaps method, the accuracy is the same as using the original dataset if we choose the same binning scales. For sampling method, we calculate the conditional entropy values between each time step pairs and then compute the absolute value differences between the sample result and the original result. Then, we use a Cumulative Frequency Plot (CFP) to represent the absolute conditional entropy differences for all pairs. In CFP, a point \((x, y)\) indicates that the fraction \(y\) of all calculated relative value differences are less than \(x\). Because the value differences should be as small as possible, it implies that a method with the curve to the left has a better accuracy than the method with the curve to the right. From the figure, we can see that the smaller the sample level we use, the more accuracy loss we incur, which is not surprising. If we calculate the relative accuracy loss related to the original result using expression \((\text{original result} - \text{sample result})/\text{original result}\) for each pair, using 30% of the sample incurs an average of 21.03% information loss, using 15% of the sample incurs an average of 37.56% information loss, and using 5% of the sample incurs an average of 58.37% information loss. Hence, we can see that although the sampling method can achieve better efficiency than our method if we use a small sample level (less than 30%), the information loss is very significant. On the other hand, 30% of the sample does not help the performance, but still involves a measurable loss of accuracy.

Figure 7.12(b) shows the accuracy loss with different sampling levels for offline analyses (Correlation Mining). For correlation mining, we continued to use POP data. The variables used here are temperature and salinity with 1.4 GB data size for each variable.
Here we first divided the variables into 60 spatial and value subsets. Then we calculate the 
mutual information between these two variables within each subset. The value differences 
between the original dataset and samples are also shown in CFP plot. From the figure, we 
can also see that while bitmaps method does not have any information loss, the smaller 
sample level we use, the more accuracy loss we incur. Compared with the original result, 
using 50% of the sample incurs an average of 3.14% accuracy loss, using 30% of the sam-
ple incurs an average of 7.56% accuracy loss, using 15% of the sample incurs an average 
of 10.15% accuracy loss, using 5% of the sample incurs an average of 17.03% accuracy 
loss, and using 1% of the sample incurs an average of 32.23% accuracy loss.

7.5 Related Work

In scientific data analysis, much of the recent focus has been on in-situ analysis. ADIOS [64] 
makes it possible to independently select the IO methods being used by each grouping of 
data in an application, so that end users can use those IO methods that exhibit best perfor-
mance based on both IO patterns and the underlying hardware. GLEAN [112] is a flexible 
and extensible framework taking application, analysis and system characteristics into ac-
count to facilitate simulation-time data analysis and I/O acceleration. DIY [86] is a in-situ 
analyses prototype library of scalable core components for the decomposition and move-
ment of data, targeted for analysis workloads that are data-intensive and bound by I/O and 
communication. The recent work [62] demonstrates that the index generation efficiency 
can be greatly improved in an in-situ parallel environment. DIRAQ [66] provides a parallel 
in-situ, in network data encoding and reorganization technique that enables the transforma-
tion of simulation output into a query-efficient form.
7.6 Summary

With growing computational capabilities of parallel machines, scientific simulations are being performed at finer spatial and temporal scales, leading to a data explosion. The growing sizes are making it extremely hard to store, manage, disseminate, analyze, and visualize these datasets, especially as neither the memory capacity of parallel machines, memory access speeds, nor disk bandwidths are increasing at the same rate as the computing power. Hence, in an in-situ environment, how to effectively reduce the data size for efficient data analyses becomes an important issue.

In our work, we proposed an approach which utilizes bitmap index as a representation of the data with much smaller size. We also showed that many kinds of analyses can be supported by bitmaps without touching the original data. Each time after the data is simulated in memory, we add one step which is to generate in-situ bitmaps based on the data. Then instead of writing the data, we only write the bitmaps back to disk, which greatly improves the IO efficiency. We also demonstrated that we are able to directly use bitmaps for both online and offline analysis efficiently. We have extensively evaluated our method with different simulations and applications, and demonstrated the effectiveness of our approach.
Chapter 8: Future Work

In Chapter 7, we proposed an in-situ bitmap index generation and data analyses method to improve both online and offline analyses efficiency. Based on this work, our future work has three directions. One extension is to support in-situ importance-driven data subset selection based on bitmaps. The second extension is to support feature mining using bitmaps and also apply it in the in-situ environment. The third extension is to improve bitmap index generation efficiency for skew datasets.

8.1 Extension of the In-Situ Importance-driven Data Subset Selection Using Bitmaps

Our previous work in Chapter 7 supports efficient parallel bitmap index generation in an in-situ environment using multi-core. After the bitmaps are generated, we discard the original data and only keep bitmaps for online and offline analyses. We showed that we can perform a series of analyses (time step selection, correlation mining, and histogram spectra) using bitmaps. Moreover, our method not only is more efficient for this kind of analyses, but also saves time for data writing because bitmaps size is much smaller than the original data size. An extension to this work is that instead of keeping bitmaps for data analyses, if we can quickly find the subsets of the data which are of interest to the users and only write the data subsets to disk, it will also greatly improve the efficiency.
There are two traditional methods for importance driven data subsets selection. (1) The original data based subset selection: each time after the data is simulated, this method directly scan the entire data block and find the subsets of data that are of interest to users. This method achieves good performance only if the subsets selection is based on metrics with a small number of data scans, such as mean and variance. It keeps the data subsets which satisfy a certain predefined threshold. However, this kind of metrics usually have very limited functionalities for important data detection, e.g., it cannot support either correlation or feature mining.

(2) The histogram based subset selection: histogram is widely used to help perform data analyses. Each time after the data is simulated, this method first perform one scan of the data and generate the histogram of current data. Then, it performs different kinds of analyses (e.g., error prediction of samples based on histogram spectra) only using histogram and find the important data subsets based on that. The advantage of this method is that histogram can be generated efficiently and several kinds of analyses can be supported using histogram. The disadvantage is that histograms do not keep the data spatial information, which is an important factor for many scientific data analyses. Hence, it can not help the spatial-based data analyses.

Our proposed method is the bitmaps based subset selection. Remember that bitmaps not only keep all the information of histogram, but also contains the data spatial locality (each bitvector contains all positions of the original dataset). Besides all the analyses that histogram supports, our method also supports spatial-based data selection. For example, with the help of bitmap indexing, we can quickly find those spatial or value subsets with interesting correlations among different variables and then only write these subsets to disk, which greatly improves the efficiency. Hence, the basic idea of our work is that in an in-situ
environment, each time after a time step is simulated, we utilize the multi-core to quickly generate bitmaps, then use bitmaps to perform efficient data subsets selection and finally only keeps data subsets for further analyses. There are two challenges we want to solve:

- Bitmap index generation is time-consuming. Compared with the histogram generation which only scans data once, bitmap index generation process includes data binning and bitvectors compression, which are more time-consuming. However, in Chapter 7, we have demonstrated that the bitmap generation time can be greatly improved using multi-core. Researches on accelerating applications using GPUs, Intel MIC and heterogeneous resources also are being studied [25, 26, 27, 28, 29, 82, 102, 103, 104, 71, 72, 70]. The works have been focusing on different aspects including scheduling, memory hierarchy, cache, data compression, as well as pattern specific optimizations. In our future work, it is worthwhile to combine these techniques with in-situ bitmap index generation to further improve the efficiency.

- Support efficient correlation mining in the in-situ environment. The correlation mining work proposed in Chapter 7 is executed offline because it requires exhaustive calculation over data subsets. Although the efficiency has been greatly improved with the help of bitmaps, it is still time costly if applied in the in-situ environment. Hence, how to use some data pruning technologies to decrease the calculation times and how to perform faster parallel correlation mining utilizing multi-core is worthy a deeper study.

### 8.2 Extension of Supporting Feature Mining Using Bitmaps

Most of interesting scientific features such as eddies in ocean simulation and halos in cosmology simulation are formed by several different factors. Reflected to the data
level, these features are identified by a combination of different attributes of the data. For example, eddies are formed based on velocities in both east to west ($u$) and north to south ($v$) directions, and halos or sub-halos are identified based on multi-dimensional spaces and velocities. By studying the relationship between multiple attributes, these data features could be identified.

Several different algorithms have been designed to detect eddies or halos. However, these algorithms need to scan through the entire dataset several times in order to identify the potential eddies or halos. And many of them have to sacrifice the accuracy to improve the feature finding efficiency. We propose a method to perform feature mining based on bitmap indices of multiple variables with the goal to improve the eddy finding efficiency.

Next we describe how bitmap index is able to accelerate eddy finding efficiency using one eddy detection algorithm (Okubo-Weiss method). The formula of Okubo-Weiss method is shown in the following. Here, variable $u$ refers to the east-west velocity component, and $v$ to the north-south velocity component. Then, $x$ and $y$ are locally east-pointing and north-pointing vectors, respectively. If $OW$ is larger than $0$, it will not be eddy; otherwise it might be eddy.

In the first step, we build up multi-level bitmap indices over variables $u$ and $v$. For variable $u$, the bitmaps are built based on column-major sequence ($y, x$), and for variable $v$, the bitmaps are built based on row-major sequence ($x, y$). Within each bin of $u$ or $v$, there will be long sequences of 0 bits and 1 bits, and the continuous 1-bits sequences mean that either $\frac{\partial v}{\partial x}$ or $\frac{\partial u}{\partial y}$ are 0. If either part is zero, we can find that OW distance must

$$OW = \left( \frac{\partial u}{\partial x} - \frac{\partial v}{\partial y} \right)^2 + \left( \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right)^2 - \left( \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \right)^2$$
be equal or larger than 0, which is not eddy. Thus, eddies will never exist for continuous 1-bits within each bin. Using bitmaps, we can quickly identify those regions which do not contain eddy.

8.3 Efficient Bitmap Index Generation over Skew Dataset

Support efficient bitmap index generation over the skew datasets is challenge. Remember bitmap indices are built based on the value subsets (bins), and different binning scales have big effect on the query efficiency. The traditional method to decide the binning scale for skew data is that we first do one scan of the entire dataset, figure out the value distribution of the data and calculate binning scales based on the value distribution. Then we do another scan of the data and generate bitmap indices based on the decided binning scales. For big scale of data, this method is clearly inefficient because it needs two scans of the entire dataset in order to generate bitmap indices. Moreover, in an in-situ environment, the bitmap index is generated as data keeps simulated. Bitmap indexing needs to be generated by performing only one scan of the data in such case.

Facing such problem, one solution is to dynamically decide the binning scale during the index generation process. Before the index generation, we first choose a default binning scale. Then during the index generation process, we keep adjusting the binning scale based on the value distribution of already scanned data subsets. For example, we merge two neighbor bitvectors if both bitvectors contain small number of elements, and we split one bitvector into two if this bitvector includes big number of elements. This way, we are able to generate bitmap index with only one data scan. However, dynamically merging or splitting bitvectors are also time-consuming. Although this method only requires one scan of the data, it does not improve the index generation efficiency.
Several works [78, 79, 57] focused on value partitioning over the skew data. In these works, a annotated cube lattice is constructed using a data sample to estimate the data cardinalities and the partition factors are decide based on the data sample. The similar idea can be used in bitmap index generation. Each time we quickly select a small sample of the data and decide the index binning scale based on that. Then we generate bitmap index over the original data using this binning scale. This way, it can greatly improve the index generation efficiency.
Chapter 9: Conclusions

This is a summary of the contributions of my dissertation and future work.

9.1 Contributions

Nowadays we are facing the “big data” issue. The key challenge we are trying to solve in this thesis is that the dataset sizes continue to grow rapidly. However, the data transfer bandwidths and disk IO speed are growing at a much slower pace, making it extremely hard for scientists to analyze these rapidly growing datasets. In this dissertation, we introduce different strategies and optimizations to improve the efficiency of “big data” analysis. Our contributions can be summarized as following:

- We proposed a light-weight data management approach for scientific datasets stored in NetCDF and HDF5. Using this method, the low-level data formats and data access details are totally hidden from users. A metadata is designed to help users understand the data and perform queries. We support SQL select and aggregation queries specified over a virtual view of the data. We also designed a parallelization algorithm which supports efficient data selection and aggregation in a parallel environment.

- We added bitmap indexing support in our data virtualization framework to improve the data subsetting efficiency and decrease the memory cost. Facing the efficiency
issues for queries containing both value-based and dimension-based subsetting conditions, we proposed a two-phase optimizations (a partition strategy in index generation phase, logic bitwise operations in index retrieval phase) to improve the efficiency. We also designed a parallel architecture to improve the data query efficiency for multi-variable dataset using parallel bitmap indexing.

- We integrated our light-weight data management approach with wide-area data transfer protocols, by developing SDQuery DSI which is in GridFTP plug-in. One one hand, our tool supports very efficient and reliable data transfer. On the other hand, users can flexibly select any data subset they want to download. Besides the basic functionality, we also added three optimizations. We incorporated parallel indexing to perform indexing operations for each sub-block concurrently. We used a performance model to automatically select between indexing-based retrieval of data segments and full retrieval followed by in-memory filtering to improve data reading efficiency. We supported a parallel streaming technique, where different disk blocks are read concurrently and piped to different TCP/IP streams, which improves data transfer efficiency.

- In many cases when users want to analyze the entire dataset, or the data subset is still big in size, sampling is a good way to decrease the data size and improve data analysis efficiency. We designed a bitmap index based sampling method, which preserves both value distribution and spatial locality of the data and hence can achieve a much better sampling accuracy than other sampling methods. With only small amount of metadata, our method can also help assess loss of accuracy with a particular sub-sampling level efficiently. Moreover, this method can be flexibly applied to
any subset of the original dataset, which may be specified using a value-based and/or a dimension-based subsetting predicate.

- We proposed a bitmap indexing based method to improve the efficiency of correlation analysis over multiple variables. Without indexing support, correlation analysis requires scanning multiple big-scale data blocks for calculation. And there is no flexible method for supporting correlation analysis over subsets of the data. In our work, we showed that we can use fast bitwise operations to calculate correlations directly using bitmap. We also proposed two partitioned methods for parallel correlation analysis and combined correlation analysis with index-based sampling. We incorporated our work in a flexible correlation analysis which supports interactive correlation analyses over subsets of the data.

- Finally, we proposed a novel method which utilizes bitmap index as a representation of the data and showed that many kinds of analyses can be supported by bitmaps without touching the original dataset. Each time after the data is simulated in memory, we add one step which is to generate in-situ bitmap index based on the data. Then instead of writing the data, we only write the bitmaps (much smaller in size) back to disk, which greatly improves the IO efficiency. We also demonstrated that we are able to directly use bitmaps for both online (in-situ) analysis (e.g., time step selection) and offline analysis (e.g., correlation mining) efficiently.

In the future work, we will continue working on improving the efficiency of in-situ data analysis. Instead of keeping only bitmap index for data analyses, we want to find a method which can support efficient correlation mining in the in-situ environment. This way, only useful data (much smaller in size) is kept for further analysis. Bitmap indexing can be quite
helpful during this process for the efficiency concern. Support parallel index generation in GPU environment can further improve the efficiency. Moreover, feature mining (e.g., eddy finding, halo finding) also plays an important role in scientific discovery. We will work on supporting efficient feature mining using bitmap index in the in-situ environment. Finally we will focus on improving the bitmap index generation efficiency for skew dataset.
Bibliography


[70] Xing Lin and Rajeev Balasubramonian. Refining the utility metric for utility-based cache partitioning. In *The 9th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, San Jose, CA, 2011.


