Data Management and Data Processing Support on Array-Based Scientific Data

Dissertation

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

By

Yi Wang, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Dissertation Committee:

Gagan Agrawal, Advisor
Feng Qin
Srinivasan Parthasarathy
Abstract

Scientific simulations are now being performed at finer temporal and spatial scales, leading to an explosion of the output data (mostly in array-based formats), and challenges in effectively storing, managing, querying, disseminating, analyzing, and visualizing these datasets. Many paradigms and tools used today for large-scale scientific data management and data processing are often too heavy-weight and have inherent limitations, making it extremely hard to cope with the ‘big data’ challenges in a variety of scientific domains.

Our overall goal is to provide high-performance data management and data processing support on array-based scientific data, targeting data-intensive applications and various scientific array storages. We believe that such high-performance support can significantly reduce the prohibitively expensive costs of data translation, data transfer, data ingestion, data integration, data processing, and data storage involved in many scientific applications, leading to better performance, ease-of-use, and responsiveness.

On one hand, we have investigated four data management topics as follows. First, we built a light-weight data management layer over scientific datasets stored in HDF5 format, which is one of the popular array formats. Unlike many popular data transport protocols such as OPeNDAP, which requires costly data translation and data transfer before accessing remote data, our implementation can support server-side flexible subsetting and aggregation, with high parallel efficiency. Second, to avoid the high upfront data ingestion costs of loading large-scale array data into array databases like SciDB, we designed a
system referred to as SAGA, which can provide database-like support over native array storage. Specifically, we focused on implementing a number of structural (grid, sliding, hierarchical, and circular) aggregations, which are unique in array data model. Third, we proposed a novel approximate aggregation approach over array data using bitmap indexing. This approach can operate on the compact bitmap indices rather than the original raw datasets, and can support fast, accurate and flexible aggregations over any array or its subset without data reorganization. Fourth, we extended bitmap indexing to assist the data mining task subgroup discovery over array data. Like the aggregation approach, our algorithm can operate entirely on bitmap indices, and it can efficiently handle a key challenge associated with array data – a subgroup identified over array data can be described by value-based and/or dimension-based attributes.

On the other hand, we focused on both offline and in-situ data processing paradigms in the context of MapReduce. To process disk-resident scientific data in various data formats, we developed a customizable MapReduce-like framework, SciMATE, which can be adapted to support transparent processing on any of the scientific data formats. Thus, unnecessary data integration and data reloading incurred by applying traditional MapReduce paradigm to scientific data processing can be avoided. We then designed another MapReduce-like framework, Smart, to support efficient in-situ scientific analytics in both time sharing and space sharing modes. In contrast to offline processing, our implementation can avoid, either completely or to a very large extent, both data transfer and data storage costs.
This is dedicated to the ones I love: my parents and my fiancee.
Acknowledgments

I would never have been able to complete my dissertation without the guidance of my committee members, help from friends, and support from my family and fiancee over the years.

Foremost, I would like to express my sincerest gratitude to my advisor Prof. Gagan Agrawal for his continuous support of my Ph.D career. He is the researcher with the broadest interests I have ever seen, and he has offered me excellent opportunities to delve into a variety of research topics, including high-performance computing, programming models, databases, data mining, and bioinformatics. This helps me to learn diverse techniques, gain different methodologies, and develop my potential in an all-around way. I am also grateful that he is not a harsh boss who pushes immense workload with close supervision, but a thoughtful mentor who teaches me how to be self-motivated and enjoy the research. My 5-year exploration would not have been so rewarding without his patience, encouragement, and advice.

Besides my advisor, I would like to thank my candidacy and dissertation committee members, Prof. Feng Qin, Prof. Srinivasan Parthasarathy, Prof. Han-Wei Shen, and Prof. Spyros Blanas, for their valuable time and insightful advices. I would also like to thank Dr. Arnab Nandi, Dr. Kun Huang, Dr. Hatice Gulcin Ozer, Dr. Xin Gao, Dr. Jim-Jingyan Wang for the collaborations during my PhD career. I also need to acknowledge a CSE
staff, Aaron Jenkins, and I will always remember the nice time I worked with him in my first quarter.

My sincere thanks also go to my hosts Dr. Marco Yu and Dr. Donghua Xu, and my mentor Dr. Nan Boden, for their help in both work and life during the summer internship at Google. Under their guidance, I learned diverse techniques fast and was even able to apply them to the research projects after the internship. Working with them was quite a rewarding experience that I might not be able to obtain from academia. The insightful discussions with them can always bring me some inspirations of interesting research directions, some of which I have already fulfilled in the last year of my PhD study. Their enthusiasm and charms let me make the decision to join the same team at Google after graduation.

I also would like to thank the colleagues in our group, Data-Intensive and High Performance Computing Research Group. My Ph.D. career would not have been so colorful without the friendship from them. Former and present members of our lab have provided me with numerous help, in both daily life and work, and they are: Wenjing Ma, Vignesh Ravi, Tantan Liu, Tekin Bicer, Jiedan Zhu, Vignesh San, Xin Huo, Ashish Nagavaram, Liyang Tang, Yu Su, Linchuan Chen, Mehmet Can Kurt, Mucahid Kutlu, Jiaqi Liu, Roeel Ebenstein, Sameh Shohdy, Gangyi Zhu, Peng Jiang, Jiankai Sun, Qiwei Yang, and David Siegal.

Last but not the least, I express special thanks to my parents and my fiancee. My parents have been always considerate and supportive. My fiancee has been always cheering me up and standing by me through the ups and downs.
Vita

July 29th, 1987 .......................... Born - Wuhan, China

2010 .............................. B.S. Software Engineering,
Wuhan University, Wuhan, China

2014 .............................. M.S. Computer Science & Engineering,
The Ohio State University

May 2014 - August 2014 .................. Software Engineer Intern, Google Inc.,
Mountain View, CA

Publications

Research Publications

and Early Termination on a MapReduce-Like Framework for Online Analytics”. Under
Submission.

Yi Wang, Gagan Agrawal, Tekin Bicer, and Wei Jiang. “Smart: A MapReduce-Like
Framework for In-Situ Scientific Analytics”. In Proceedings of the International
Conference for High Performance Computing, Networking, Storage and Analysis (SC),
November, 2015.

Yi Wang, Gagan Agrawal, Tekin Bicer, and Wei Jiang. “Smart: A MapReduce-Like
Framework for In-Situ Scientific Analytics”. Technical Report, OSU-CISRC-4/15-TR05,
April, 2015.

over Scientific Datasets Using Bitmap Indices”. Technical Report, OSU-CISRC-3/15-
TR03, March, 2015.


Jim Jing-Yan Wang, Yi Wang, Shiguang Zhao, and Xin Gao. “Maximum Mutual Information Regularized Classification”. In Engineering Applications of Artificial Intelligence (ENG APPL ARTIF INTEL), 2015.


Fields of Study

Major Field: Computer Science and Engineering
Table of Contents

Abstract ................................................................. ii
Dedication ............................................................... iv
Acknowledgments ......................................................... v
Vita ........................................................................ vii
List of Tables ................................................................ xv
List of Figures ................................................................ xvii
List of Algorithms ......................................................... xxi

1. Introduction ............................................................... 1
   1.1 Motivation ............................................................. 1
   1.2 Contributions ....................................................... 2
      1.2.1 Data Management Support .............................. 3
      1.2.2 Data Processing Support .............................. 13
   1.3 Outline ................................................................. 17

2. Supporting a Light-Weight Data Management Layer over HDF5 ....... 19
   2.1 Background .......................................................... 19
      2.1.1 HDF5 .......................................................... 19
      2.1.2 High-Level HDF5 ...................................... 20
   2.2 System Design ....................................................... 20
      2.2.1 Motivation ................................................... 20
      2.2.2 System Overview ...................................... 22
4.2 Approximate Aggregations Using Bitmap Indices
  4.2.1 Background: Bitmap Indexing and Binning Strategies
  4.2.2 Bitmaps and Approximate Aggregation
  4.2.3 Approximate Aggregation Method
4.3 V-Optimized Binning
  4.3.1 Motivation
  4.3.2 Unbiased V-Optimized Binning
  4.3.3 Weighted V-Optimized Binning
4.4 Experimental Results
  4.4.1 Experimental Setup
  4.4.2 Aggregation Accuracy of Different Approximate Aggregation
       Methods
  4.4.3 Aggregation Costs of Different Aggregation Methods
  4.4.4 Aggregation Accuracy of Different Binning Strategies
  4.4.5 Indexing Costs of Different Binning Strategies
  4.4.6 Discussion
4.5 Related Work
4.6 Summary

5. SciSD: Novel Subgroup Discovery over Scientific Datasets Using Bitmap Indices
  5.1 Subgroup Discovery and Array Data
    5.1.1 Preliminaries
    5.1.2 An Example Involving Arrays
    5.1.3 Extended Subgroup Description
    5.1.4 Quality Function and Subgroup Combination
  5.2 SciSD Algorithm
    5.2.1 Tight Optimistic Estimates
    5.2.2 Search Strategy
  5.3 Subgroup Pruning and Combination
    5.3.1 Pruning Measures
    5.3.2 Combination Rules
  5.4 Algorithm Optimization Using Bitmaps
  5.5 Experimental Results
    5.5.1 Experimental Setup
    5.5.2 Comparison with SD-Map*
    5.5.3 Evaluation over Larger Datasets
    5.5.4 Effectiveness of Search Strategy
  5.6 Related Work
  5.7 Summary
   6.1 Background ................................................................. 160
      6.1.1 MapReduce and MATE ............................................. 160
      6.1.2 Scientific Data Formats ........................................... 161
   6.2 System Design .......................................................... 162
      6.2.1 System Overview .................................................... 162
      6.2.2 API for Integrating a New Data Format ......................... 165
   6.3 System Optimization .................................................. 166
      6.3.1 Data Access Strategies and Patterns ............................ 166
      6.3.2 Optimizing Different Types of Accesses ....................... 168
   6.4 Experimental Results ................................................ 171
      6.4.1 Applications and Environmental Setup ......................... 172
      6.4.2 Functionality Evaluation ......................................... 172
      6.4.3 Optimization Experiments ....................................... 177
   6.5 Related Work ......................................................... 179
   6.6 Summary ............................................................... 181

7. Smart: A MapReduce-Like Framework for In-Situ Scientific Analytics .... 184
   7.1 MapReduce and In-Situ Analytics .................................... 184
      7.1.1 Opportunity and Feasibility .................................... 184
      7.1.2 Challenges .......................................................... 185
   7.2 System Design ........................................................ 189
      7.2.1 System Overview .................................................. 189
      7.2.2 Two In-Situ Modes ............................................... 191
      7.2.3 Runtime System .................................................. 193
   7.3 System Optimization for Window-Based Analytics .................. 198
      7.3.1 Motivation .......................................................... 198
      7.3.2 Optimization: Early Emission of Reduction Objects .......... 199
   7.4 Experimental Results ................................................ 200
      7.4.1 Applications and Environmental Setup ......................... 201
      7.4.2 Performance Comparison with Spark ............................. 202
      7.4.3 Performance Comparison with Low-level Analytics Programs .. 205
      7.4.4 Scalability Evaluation .......................................... 207
      7.4.5 Evaluating Memory Efficiency ................................... 209
      7.4.6 Comparing Time Sharing and Space Sharing Modes ............ 211
      7.4.7 Evaluating the Optimization for Window-Based Analytics ... 214
   7.5 Related Work .......................................................... 215
   7.6 Summary ............................................................... 218
8. Conclusions and Future Work ........................................ 219

8.1 Contributions ......................................................... 219

8.2 Future Work ......................................................... 221

8.2.1 Bitmap-Based Data Analysis ................................. 222

8.2.2 Smart Extension .................................................. 224

Bibliography ......................................................... 226
List of Tables

Table | Page
-----|------
2.1  Type 2 and Type 3 Query Examples | 37
2.2  Aggregation Query Examples | 41
2.3  Data Transfer Volume Comparison against OPeNDAP | 42
3.1  Structural Aggregation APIs and Mathematical Definitions | 48
3.2  Summary of Model Parameters | 58
4.1  Definitions of the Aggregation Interface for Different Aggregate Operators | 96
4.2  Query Examples of Five Types | 106
4.3  Average Relative Error(%) of MAX Aggregation of Different Methods on the Real-World Dataset | 110
5.1  The Best 8 Subgroups Discovered on WOA09_5DEG by SD-Map* | 146
5.2  The Best 8 Subgroups Discovered on WOA09_5DEG by SciSD with Maximum CWE of 1.5 | 147
5.3  Comparison between SD-Map* and SciSD on WOA09_5DEG with Varying Maximum CWE | 148
5.4  The Best 8 Subgroups Discovered on WOA13 by SD-Map* | 150
5.5  The Best 8 Subgroups Discovered on WOA13 by SciSD with Maximum CWE of 1.5 | 150
5.6 Comparison between SD-Map* and SciSD on WOA13 with Varying Maximum CWE ................................................................. 151

5.7 Search Efficiency Evaluation on WOA09_1DEG with Varying # of Bins and Maximum CWE of 1.5 .................................................. 156

6.1 Descriptions of the functions in the Scientific Data Processing API .... 182

6.2 Invoked Calls in NetCDF/HDF5 Lib for full read Implementation .... 183
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Execution Overview</td>
<td>23</td>
</tr>
<tr>
<td>2.2 Metadata Structure</td>
<td>25</td>
</tr>
<tr>
<td>2.3 Hyperslab Selector Module</td>
<td>27</td>
</tr>
<tr>
<td>2.4 Parallelism of Query Execution</td>
<td>30</td>
</tr>
<tr>
<td>2.5 Performance Comparison for Type 1 Queries</td>
<td>35</td>
</tr>
<tr>
<td>2.6 Performance Comparison for Type 2 and Type 3 Queries</td>
<td>37</td>
</tr>
<tr>
<td>2.7 Parallel Query Processing Times with Different Dataset Sizes</td>
<td>38</td>
</tr>
<tr>
<td>2.8 Parallel Query Processing Times with Different Queries</td>
<td>39</td>
</tr>
<tr>
<td>2.9 Performance Comparison of Content-Filtering over Atomic Datasets and Compound Datasets</td>
<td>40</td>
</tr>
<tr>
<td>2.10 Comparison of Data Aggregation Times</td>
<td>43</td>
</tr>
<tr>
<td>3.1 Structural Aggregation Examples</td>
<td>47</td>
</tr>
<tr>
<td>3.2 Examples of Different Partitioning Strategies</td>
<td>53</td>
</tr>
<tr>
<td>3.3 Processing Times (left) and Predicted Processing Times (right) on Synthetic Datasets of Varying Skew</td>
<td>68</td>
</tr>
<tr>
<td>3.4 Processing Times (left) and Predicted Processing Times (right) on Cloud Pressure Dataset</td>
<td>68</td>
</tr>
</tbody>
</table>
5.3 Use of Bitmaps in SciSD ........................................... 140

5.4 Evaluation on WOA09_1DEG Dataset: Impact of Varying Maximum CWE and # of Bins on # of Subgroups, Execution Times, and Average CWRAcc . 153

5.5 Evaluation on POP Dataset: Impact of Varying Maximum CWE and # of Bins on # of Subgroups, Execution Times, and Average CWRAcc . . . . . 154

6.1 Execution Overview of SciMATE ...................................... 162

6.2 Details of the Scientific Data Processing Module ......................... 164

6.3 Contiguity Experiments .................................................. 170

6.4 K-means: Data Processing Times (# of Nodes = 1) ...................... 173

6.5 PCA: Data Processing Times (# of Nodes = 1) ......................... 173

6.6 KNN: Data Processing Times (# of Nodes = 1) ........................ 174

6.7 K-means: Data Processing Times (8 Threads per Node) ................ 174

6.8 PCA: Data Processing Times (8 Threads per Node) ...................... 175

6.9 KNN: Data Processing Times (8 Threads per Node) ...................... 175

6.10 K-means: Data Loading Times (8 Threads per Node) .................. 176

6.11 PCA: Data Loading Times (8 Threads per Node) ....................... 176

6.12 PCA: Comparison between Partial Read and Full Read ................ 177

6.13 KNN: Comparison between Fixed-size Column Read and Contiguous Column Read on NetCDF Datasets .......................... 179

6.14 KNN: Comparison between Fixed-size Column Read and Contiguous Column Read on HDF5 Datasets .......................... 180

7.1 System Overview of Smart .............................................. 190

7.2 Two In-Situ Modes ....................................................... 192

xix
7.3 Performance Comparison with Spark . . . . . . . . . . . . . . . . . . . . . . 203
7.4 Performance Comparison with Low-Level Programs . . . . . . . . . . . . 206
7.5 In-Situ Processing Times with Varying # of Nodes on Heat3D (Using 8
Cores per Node) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 208
7.6 In-Situ Processing Times with Varying # of Threads on Lulesh (Using 64
Nodes) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 209
7.7 Evaluating the Efficiency of Time Sharing Mode . . . . . . . . . . . . . . 210
7.8 Evaluating the Efficiency of Space Sharing Mode . . . . . . . . . . . . . . 212
7.9 Evaluating the Effect of Optimization for Window-Based Analytics . . . . 214
## List of Algorithms

1. localAggregation(hyperslab, content-basedCondition, groupbyIndex, aggregationType) ........................................ 33
2. GridAgg(slab S, num_procs n) ....................................................... 52
3. SlidingAgg(slab S, num_procs n) ................................................. 65
4. ChunkBasedGridAgg(slab S, num_procs n) .................................... 66
5. Agg(bitmap B, statistics S, aggOperator Op, valBasedPredicateP_v, dimBasedPredicate P_d) .................................................. 94
6. partition(num_elements N, num_bins B) ........................................ 101
7. SciSD(dim-based attr. universal set $\Omega^D_A$, val-based attr. universal set $\Omega^V_A$, dim-range universal set $R^D$, val-range universal set $R^V$) ........................................ 134
8. run(const In* in, size_t in_len, Out* out, size_t out_len) ............ 195
9. reduce(Split split) ................................................................. 199
Chapter 1: Introduction

1.1 Motivation

Our overall research work in this dissertation has been motivated by two recent trends. First, the past decade has witnessed an unprecedented scientific data explosion, mostly collected from high-throughput instruments or generated from massive simulations. Along with the data structures like graphs, sets and trees, various array storages have been serving as the heart of the large-scale scientific data analytics [116]. Thus, effectively managing, disseminating, querying, and analyzing arrays becomes one of the key ‘big-data’ challenges today. This trend has sparked a new class of array-based scientific data management and data processing tools, including scientific data transport protocols like OPeNDAP [62] that simplifies the data access to remote data of arbitrary size, many Array DBMSs, including SciDB [39], RasDaMan [32] and MonetDB [258] that facilitate array manipulations, and MapReduce variants like SciHadoop [42] that supports MapReduce tasks over scientific data.

Second, since scientific datasets are often stored in different data formats as well as various underlying physical structures, before the data processing phase, most scientific data management and data processing tools require aforehand data transformation and data transfer, which are very likely to be prohibitively expensive for massive amounts of data.
Some prominent examples are as follows: 1) to hide the data formats and data processing details, current data transport protocols (e.g., OPeNDAP), require to translate the original data format into a standard OPeNDAP data format, and also require user client to download the entire data first and then write its own code to perform subsetting and aggregation, 2) current Array DBMSs including SciDB, RasDaMan and MonetDB, like their relational counterparts, require ingesting data into database systems before processing queries, 3) current MapReduce implementations often require that data be integrated into specialized file systems, like the Hadoop Distributed File System (HDFS), and 4) implementations of many data mining algorithms (e.g., VIKAMINE [14] that implements a subgroup discovery algorithm [27]), though they may be adapted to analyze array data, mostly they still require the data be transformed into relational data, and many of them cannot process large-scale datasets. However, with rapidly growing size of scientific datasets, those heavy-weight tools may not be feasible in many data-intensive applications [21].

Therefore, our overall work focuses on the design of high-performance data management and data processing support on array-based scientific data, which can significantly reduce the prohibitively expensive costs of data integration, data translation, data transfer, data ingestion, data processing, and data storage involved in many scientific applications. Generally, we \(^1\) would like to avoid data transformations and minimize data movements involved in the data flow of a variety of scientific applications.

### 1.2 Contributions

The dissertation focuses providing high-performance support in two aspects - *data management* and *data processing*. The former includes 1) a light-weight data management

\(^1\)The word “we” is used as a personal preference. The primary contributor on the work presented in this dissertation is the author.
layer over HDF5, 2) an implementation which uses native array storage as a database with the support for structural aggregations, 3) an approximate aggregation approach using novel bitmap indices, and 4) a novel subgroup discovery algorithm using bitmap indices. The latter includes two novel MapReduce-like frameworks, one for offline processing over multiple scientific data formats, and the other for in-situ analytics.

1.2.1 Data Management Support

Supporting a Light-Weight Data Management Layer over HDF5: 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 data explosion.

Finer granularity of simulation data offers both an opportunity and a challenge. On one hand, it can allow understanding of underlying phenomenon 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 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), 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.
In the context of visualization, for almost all simulations, the dataset sizes have grown well past the point that it is infeasible for a scientist to look through all of the data [111]. In addition, because of growing dataset sizes, most of the time while performing data visualization and analysis is now spent in I/O routines, reducing the effectiveness of a visualization effort.

Despite much emphasis on large-scale visualization, the state-of-the-art in dealing with large-scale datasets is very limited. If we look at the current popular tools, like ParaView and VisIt, most subsetting operations are done through an in-memory interface, where all data is loaded at once. Then, either through filtering, i.e., applying data transformations, or selection interfaces, the data is 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 (i.e. perform them prior to loading all the data in memory), to cope with the memory limitations. Similarly, while disseminating data, in many of the existing data dissemination portals (e.g., Earth System Grid (ESG)) data downloading is facilitated by OPeNDAP [62], which has only a very limited support for data subsetting, and no support for aggregation at the server side.

The key underlying reason for the limited state-of-the-art is that the management of scientific (array-based) data has received very limited attention over years. On one hand, while a typical database provides a high-level query language, it also requires all data to be loaded into the system, which is often extremely time-consuming. On the other hand, more ad-hoc solutions for data management, which do not require that data be reloaded in such a fashion, unfortunately involve format-specific coding and use of lower-level languages.
In this work, we describe an approach which involves the best of both of the two approaches. On one hand, our approach does not require data to be loaded into a specific system or be reformatted. At the same time, we allow use of a high-level language for specification of processing, which is also independent of the data format. We present an implementation of this approach for HDF5, one of the most popular formats for storing scientific data. (The following documentation lists some of the major users of HDF5\textsuperscript{2}.)

Our tool supports SQL select and aggregation queries specified over the virtual relational table view of the data. Besides supporting selection over dimensions, which is directly supported by HDF5 API also, we also support queries involving dimension scales and those involving data values. For this, we generate code for hyperslab selector and content-based filter in our system. We also effectively parallelize selection and aggregation queries using novel algorithms.

We have extensively evaluated our implementation with queries of different types, and have compared its performance and functionality against OPeNDAP. We demonstrate that even for subsetting queries that are directly supported in OPeNDAP, the sequential performance of our system is better by at least a factor of 3.9. For other types of queries, where OPeNDAP requires hyperslab selector and/or content-based filter code to be written manually, the performance difference is even larger. In addition, our system is capable of scaling performance by parallelizing the queries, and reducing wide area data transfers through server-side data aggregation. In terms of functionality, our system also supports certain state-of-the-art HDF5 features including dimension scale and compound datatype.

\textsuperscript{2}http://www.hdfgroup.org/HDF5/users5.html
Array Storage as a DB with Support for Structural Aggregations: Large-scale arrays of different sizes and dimensions are used for storing images, sensor data, simulation outputs, and statistical data in a variety of scientific domains, including earth sciences, space sciences, life sciences, and social sciences [116]. With rapid growth in dataset sizes in each of these areas, effectively managing, querying, and processing such array data is one of the key ‘big-data’ challenges today. Many studies have observed the intrinsic mismatch between the array model and relational table view [32, 39, 61, 79, 91, 96, 97, 199, 258], and thus, it is well understood that the relational model (and the systems implementing this model, i.e., the relational databases) cannot be used (efficiently) for addressing this challenge.

In recent years, many Array DBMSs, including SciDB [39] and RasDaMan [32] have been designed to address this mismatch. As their names suggest, the key feature of these systems is that arrays, and not the relational tables, are the first-class citizens. Various complex operations desired in scientific applications, including correlations, curve fitting, and clustering, are naturally defined in terms of arrays (and array elements). Overall, the array-view, instead of the relational table view, often leads to better expressibility as well as performance.

A key functionality associated with array databases is the set of operations classified as structural aggregations. These operations collect and aggregate elements within each group, where these groups are created based on positional relationships. This is in contrast to groupings based on the same value that is common with the relational table view. As scientific applications where data needs to be stored and/or viewed as array typically need support for such operations, Array DBMSs like SciDB [39],
RasDaMan [32], and MonetDB (which provides a prototype implementation of the language SciQL [258]) include the support for structural aggregations.

Array DBMSs, like their relational counterparts, involve an expensive *data ingestion* phase, where arrays are restructured so as to speedup query processing later. These systems are clearly well suited for situations where data needs to be loaded once and then queried frequently, and thus, the cost of data ingestion is well justified. On the other hand, there are applications, such as analysis of simulation data [200, 206, 234, 237], where massive amounts of data is analyzed only infrequently, and hence the cost of data ingestion cannot be justified. In fact, with current Array DBMSs, because of the use of specialized data formats in various scientific domains, ingesting data into a database system can require a series of transformations. For example, the current recommended way to import high-volume satellite imagery data, which is captured in the HDF-EOS format, into SciDB involves converting the HDF-EOS files into CSV text files, transforming the CSV files into the external files in the SciDB specialized format, loading the data into SciDB in the form of temporary 1-dimensional arrays, and finally, casting the loaded 1-dimensional arrays into n-dimensional ones within SciDB [172]. Clearly, the process is extremely time-consuming and requires availability of additional memory and disk space for transformations and storing data in intermediate formats. In practice, such data ingestion is extremely expensive, taking easily 100x the time for executing a simple query over the same amount of data.

As an alternate solution to databases requiring data ingestion costs, a new paradigm of using native storage as a DB and providing database-like support has recently been shown to be an effective approach for dealing with infrequently queried data. This paradigm aims to develop a database engine on top of the native storage. Examples include the NoDB
approach [22] and automatic data virtualization [238]. Although this paradigm can be a promising approach for handling massive arrays that are not queried often, to our best knowledge, so far it has only been applied to relational tables stored in flat files [22] or for simple selection queries over arrays [238].

Applications that generate massive arrays, such as the scientific applications, often store the data in one of a small number of popular array storage formats, like NetCDF and HDF5. Thus, it is important to examine if database-like querying support can be provided on top of such storage of data. In this work, we describe an approach of using array storage as a DB with support for structural aggregations. This approach has been implemented in a system we refer to as Structural Aggregations over Array storage (SAGA). We focus on efficiently implementing and effectively parallelizing key structural aggregation operations, including grid, sliding, hierarchical, and circular aggregations. We propose different partitioning strategies, with the goal of handling both computationally expensive and inexpensive aggregations, as well as dealing with possibly skewed data. Moreover, we design an analytical model for choosing the best scheme for a given query and dataset. We have also developed aggregation methods for improving the performance on chunked array storage. Our system can process data from any array storage formats (including formats like HDF and NetCDF that are frequently used for scientific data) and even chunked or compressed array storage, as long as an interface for loading an array slab is provided [200, 206, 234].

We have extensively evaluated our structural aggregation approaches by using multiple real-world and a set of synthetic datasets of varying skew. We show how the relative performance of different partitioning strategies changes with varying amount of computation in the aggregation function and different levels of data skew. Moreover, we
demonstrate the effectiveness of our cost models for choosing the best partitioning strategy. By comparing performance with SciDB, we show that despite working on native array storage, the aggregation costs with our system are significantly lower. Finally, we also show that our structural aggregation implementations achieve high parallel efficiency.

**Approximate Aggregations Using Novel Bitmap Indices:** An important trend over the years has been the advent of approximate query processing (AQP), which is a cost-effective technique for handling large-scale data. The idea here has been sacrificing some level of accuracy to meet stringent response-time requirements. Since aggregation serves as a key functionality for databases, various approximate aggregation techniques have been proposed [45, 101, 105, 110, 174, 220, 221]. However, almost all of the proposed techniques have targeted relational databases and data warehouses. With wide use of arrays to store images, sensor data, simulation outputs, and statistical data, and with growing size of such data, it is very important to be able to apply approximate aggregation over array data.

In this work, we develop a novel approximate aggregation approach suitable for large-scale array data. Our work is based on a novel application of one of the indexing structures, the bitmap or bitvectors, with the insight that bitmap is a summary structure that can preserve both value and spatial distributions within the array. We also note that data repositories are likely to leverage bitmap indexing to accelerate selection operation (or subsetting) over arrays [56, 77, 201], and we are proposing to use bitmaps not only for selection query processing, but also as stand-alone synopses for approximate aggregations. The aggregate operators supported by our approach include the common aggregations that are widely used and supported, and moreover, by customizing the pre-aggregation statistics stored in the metadata, our approach can also support certain
user-defined aggregation functions. Further, flexible dimension-based and/or value-based predicates can be applied, and no data reorganization is needed.

In the process, we have developed a novel binning algorithm to improve the aggregation accuracy over skewed data. The conventional binning strategies have been *equi-width* and *equi-depth* [246]. By taking inspirations from the *v-optimal* histogram techniques [88, 103], we have developed a *v-optimized* binning strategy to improve the aggregation accuracy. An important contribution is performing such binning (or its approximation) efficiently. Moreover, we design a weighted extension to further improve the accuracy when the probability of querying each element is not necessarily equal.

We have extensively evaluated our approximate aggregation approach using both real-world and synthetic datasets. Our results demonstrate that 1) our approximate aggregation approach can be effectively applied to any subsets specified by dimension-based and/or value-based predicate(s); 2) our bitmap-based aggregation provides higher accuracy and/or lower processing overheads over methods like sampling or multi-dimensional histograms, and significantly lower storage costs than multi-dimensional histograms; and 3) our v-optimized binning provides higher accuracy than other conventional binning strategies for most cases, even if the data is skewed, while the indexing costs with such binning are only slightly higher than equi-depth binning.

**Novel Subgroup Discovery Over Scientific Datasets Using Bitmap Indices:** *Subgroup discovery* [119, 240] is a broadly applicable exploratory technique, which identifies interesting subgroups with respect to a property of interest. The underlying goal is to extract relations with interesting characteristics between a *dependent* or *target* variable and multiple *independent* or *explaining* variables. For example, professional basketball
players are mostly taller than the general population, where the attribute ‘occupation’ (basketball player) is the explaining variable and the attribute ‘height’ (greater than the average of the general population) is the target variable.

In both data management and data analytics areas, there is an increasing emphasis on array data [56, 206], whereas the existing work on subgroup discovery has been primarily restricted to application on relational or tabular datasets. In this data-driven era, it is highly desirable to customize the subgroup discovery technique to discover interesting patterns when data is stored as arrays. For example, given a set of arrays that are output from a scientific simulation, scientists may be interested in identifying the underlying relationships between variables. As a specific instance, in the output from an ocean simulation, the relationship between large value of salinity and other variables such as temperature and depth, is of great interest to scientists [55]. Analysis of array-based data imposes unique challenges – e.g., temperature is a value-based attribute that is stored as a separate array, whereas depth is a dimension-based attribute that corresponds to one of the array dimensions. In practice, it can be observed that the subsets of output grid points, which have a low depth and/or a high temperature are very likely to have a salinity value significantly higher than the average of the entire dataset. Clearly, the data subsets (i.e., subgroups) described by depth and/or temperature ranges can be a set of interesting subgroups, and identifying such subgroups can be of great interest. However, there is no work that can identify interesting subgroups over array data automatically. A seemingly viable approach can be to apply existing relation-based subgroup discovery algorithms directly over array data by treating dimension-based attributes as additional relational attributes. However, this approach requires data reorganization, which can often be
prohibitively expensive for large datasets, in terms of both computation and storage costs [203].

Developing subgroup discovery algorithms for array-based scientific data involves at least three sets of challenges. First, as we have already stated above, unlike the conventional subgroup discovered over relational data, a subgroup identified over array data can be described with value-based attributes and/or dimension-based attributes. Second, in scientific datasets, the attributes are mostly numeric, whereas most conventional subgroup discovery algorithms [28, 115, 119, 130, 133, 240] mainly target binary or categorical attributes. Third, the conventional algorithms for subgroup discovery are only able to operate with small datasets. In comparison, datasets of interest to us, particularly, from simulation datasets, can be extremely large, and are likely to become larger in the future.

In this work, we present a novel algorithm, SciSD, for exhaustive but efficient subgroup discovery over array-based scientific datasets, in which all attributes are numeric. Our algorithm is able to effectively discover interesting subgroups of both great generality and high quality. Moreover, our algorithm directly operates on the compact bitmap indices instead of the raw datasets, and utilizes fast bitwise operations on them, allowing processing of larger datasets.

We have demonstrated both high efficiency and effectiveness of our algorithm by using multiple real-life datasets. We first experimented on a small dataset to compare the performance (execution efficiency) as well as the quality of the output subgroups from our algorithm against SD-Map* [27], a popular subgroup discovery algorithm. The results have shown that our algorithm not only produces subgroups of significantly greater generality and higher quality, but the execution times are also lower by up to two orders of
magnitude. We also evaluated our algorithms over larger datasets. We find that by using only a small number of bins, we obtain both high-quality subgroups and low execution times, which demonstrate the practicality of our algorithm.

1.2.2 Data Processing Support

A Novel MapReduce-Like Framework for Multiple Scientific Data Formats: Gaining insights from data to facilitate scientific discoveries has emerged as the fourth paradigm for research, in addition to theory, experimentation, and computation. At the same time, the amount of data collected from instruments or generated from simulations is increasing at a massive rate. Thus, there is a growing need for tools that can help the development of data analysis applications, which can then be executed in a scalable fashion on large datasets.

In both scientific and commercial sectors, there has been a lot of interest in data-intensive computing. Much of the activity in this area has been around the MapReduce paradigm for implementing data-intensive applications [63]. MapReduce and its variants are quite effective in simplifying the development of data-intensive applications, through the use of a simple API, and with a robust runtime system.

Despite the popularity of MapReduce, there are several obstacles to applying it for developing scientific data analysis applications. Current MapReduce implementations require that data be loaded into specialized file systems, like the Hadoop Distributed File System (HDFS). On the other hand, high performance computing systems where large-scale simulation data is produced use file systems like PVFS. Though certain recent developments could enable execution of Hadoop jobs on PVFS [42, 210], scientific data tends to be stored in formats like NetCDF, HDF5, or ADIOS. Hadoop is not capable of
transparently processing data stored in these formats. Moreover, with rapidly growing size of scientific datasets, reloading data in another file system or format is not feasible [21].

In this work, we describe a framework that allows scientific data in different formats to be processed with a MapReduce-like API. Our system is referred to as SciMATE, and is based on the MATE system developed at Ohio State [108, 109]. SciMATE is developed as a customizable system, which can be adapted to support processing on any of the scientific data formats. Particularly, it has a data format adaption API, which can be used to create an instance of the system for processing data in a particular format. Once such an instance has been created, data processing applications developed using a MapReduce-like API can be executed. Applications can be developed assuming a simple logical view of data. Thus, data analysis application developed for our system is independent of the specific data format used for storing the data.

We have demonstrated the functionality of our system by creating instances that can be processing NetCDF and HDF5 formats as well as flat-files. We have also implemented three popular data mining applications and have evaluated their execution with each of the three instances of our system.

**A MapReduce-Like Framework for In-Situ Scientific Analytics:** A major challenge faced by data-driven discovery from scientific simulations is a shift towards architectures where memory and I/O capacities are not keeping pace with the increasing computing power [141, 144, 228, 259]. There are many reasons for this constraint on HPC machines. Most critically, the need for providing high performance in a cost and power effective fashion is driving architectures with two critical bottlenecks — memory bound and data movement costs [121]. Scientific simulations are being increasingly executed on systems
with coprocessors and accelerators, including GPUs and the Intel MIC, which have a large number of cores but only a small amount of memory per core. As power considerations are driving both the design and operation of HPC machines, power costs associated with data movement must be avoided.

In response to this unprecedented challenge, *in-situ analytics* [118, 127, 255, 267] has emerged as a promising data processing paradigm, and is beginning to be adopted by the HPC community. This approach co-locates the upstream simulations with the downstream analytics on same compute nodes, and hence it can launch analytics as soon as simulated data becomes available. Compared with traditional scientific analytics that processes simulated data offline, in-situ analytics can avoid, either completely or to a very large extent, the expensive data movement of massive simulation output to persistent storage. This translates to saving in execution times, power, and storage costs.

The current in-situ analytics research can be very broadly classified into two areas: 1) in-situ algorithms at the *application level*, including indexing [117, 127], compression [128, 268], visualization [114, 250, 266], and other analytics [129, 219, 257]; and 2) in-situ resource scheduling platforms at the *system level*, which aim to enhance resource utilization and simplify the management of co-located analytics code [18, 37, 66, 167, 219, 262]. These in-situ middleware systems mainly play the role of a *coordinator*, aiming to facilitate the underlying scheduling tasks, such as cycle stealing [262] and asynchronous I/O [219].

Despite a large volume of recent work in this area, an important question remains almost completely unexplored: “can the applications be mapped more easily to the platforms for in-situ analytics?”. In other words, we posit that *programming model* research on *in-situ* analytics is needed. Particularly, in-situ algorithms are currently
implemented with low-level parallel programming libraries such as MPI, OpenMP, and Pthread, which offer high performance but require that programmers manually handle all the parallelization complexities. Moreover, because similar analytics may be applied in both in-situ and offline modes, another interesting question is “can the offline and in-situ analytics codes be (almost) identical?” Clearly, this is likely only if the implementation is in a high-level framework, where details like loading and staging data and complexity of parallelization are hidden from the application developer.

In this work, we describe a novel MapReduce-like framework for in-situ scientific analytics. To the best of our knowledge, this framework is the first work to exploit a high-level MapReduce-like API in in-situ scientific analytics. The system is referred to as in-Situ MapReduce liTe (Smart). Our system can support a variety of scientific analytics on simulation nodes, with minimal modification of simulation code and without any specialized deployment (such as installing HDFS). Compared with traditional MapReduce frameworks, Smart supports efficient in-situ processing by accessing simulated data directly from memory in each node of a cluster or a distributed memory parallel machine. Moreover, unlike the traditional implementations, we base our work on a variant of the MapReduce API, which avoids outputting key-value pairs and thus keeps the memory consumption of analytics programs low. To address the mismatch between parallel programming view of simulation code and sequential programming view of MapReduce, Smart can be launched from parallel (OpenMP and/or MPI) code region once each simulation output partition is ready, while the global analytics result can be directly obtained after the parallel code converges. Further, we have developed both time sharing and space sharing modes for maximizing the performance in different scenarios.
Additionally, for memory-intensive window-based analytics, we improve the in-situ efficiency by supporting early emission of reduction object.

We have extensively evaluated both the functionality and efficiency of our system, by using different scientific simulations and analytics tasks on both multi-core and many-core clusters. We first show that our system can outperform Spark [252] by at least an order of magnitude for three applications. Second, we show that our middleware does not add much overhead (typically less than 10%) compared with analytics programs written with low-level programming libraries (i.e., MPI and OpenMP). Next, by varying the number of nodes and threads, we demonstrate high scalability of our system. Moreover, by comparing with another implementation of our system that involves an extra copy of simulated data, we show the efficiency of our design (for time sharing mode). Further, we also evaluate how our space sharing mode is suitable for clusters with many-core nodes. Finally, we show our optimization for in-situ window-based analytics can achieve a speedup of up to 5.6, by comparing it with an implementation that disables early emission of the reduction object.

1.3 Outline

The rest of the dissertation is organized as follows. In Chapter 2, we describe a light-weight data management tool, which allows server-side subsetting and aggregation on scientific datasets stored in HDF5, one of the most popular scientific data formats. Chapter 3 presents algorithms, different partitioning strategies, and an analytical model for supporting structural (grid, sliding, hierarchical, and circular) aggregations over native array storage, and describes implementation of this approach in a system we refer to as Structural AGgregations over Array storage (SAGA). A novel approximate aggregation
method for massive array data using novel bitmap indices is presented in Chapter 4. Chapter 5 presents a bitmap-based subgroup discovery algorithm over scientific datasets. Chapters 6 and 7 introduce two MapReduce-like frameworks that process scientific data in offline and in-situ manners. The former system SciMATE processes disk-resident data in multiple data formats, and the latter system Smart supports in-situ analytics in both time sharing and space sharing modes. Lastly, we conclude and propose our future work in Chapter 8.
Chapter 2: Supporting a Light-Weight Data Management Layer over HDF5

In this chapter, we focus on managing scientific data in the conventional relational table view, and we discuss an SQL implementation over HDF5, one of the most popular scientific data formats. This implementation is a light-weight data management tool, which allows server-side subsetting and aggregation on HDF5 data.

2.1 Background

This section provides background information on HDF5 data format and its high-level libraries.

2.1.1 HDF5

HDF5 [3] 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.

### 2.1.2 High-Level HDF5

The HDF5 high-level APIs [4] 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.

### 2.2 System Design

In this section, we discuss the design and implementation of our system.

#### 2.2.1 Motivation

HDF5 dataset is stored in binary format. The advantage is that it supports self-describing, portable, and compact storage, as well as efficient I/O. At the same time, the disadvantage is that the API involves overly nitty-gritty details which make the data processing much harder, forcing scientists who are interested in data subsetting to have a
very detailed understanding of the data layout before being able to extract the subset. In addition, they also have to get familiar with the HDF5 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.

To address these problems, our solution involves three ideas. The first idea is that a virtual relational table view can be supported over HDF5 dataset, and standard SQL queries with SELECT, FROM, and WHERE clauses on such a view can provide a very convenient yet powerful way to specify the data subsetting. The second idea is that data aggregation and group-by, using the same abstractions, can be applied to extract data summaries, which can minimize data transfer costs. The third idea is that sequential subsetting of data can be prohibitively expensive with growing datasets, and therefore, it’s more desirable to exploit parallelism to accelerate data subsetting and aggregation queries.

An HDF5 dataset consists of a set of multi-dimensional arrays. These arrays typically involve spatial and/or temporal dimensions and, in many cases, corresponding dimension scales as well. Scientists are usually interested in querying a subset of the data based on the either dimension indices or coordinate system expressed by dimension scales. Hence, we can divide the queries into three categories: 1) Queries based on dimensional index value(s) - i.e., those related with the dataset physical layout; 2) Queries based on coordinate value(s) - Besides the dimension information, the scientists may also query data subsets based on spatial or temporal values, which are usually stored in dimension scales; 3) Queries based on data value(s) - The scientists could also query the target data value within a specific value range. Note that in HDF5 datasets, compound datatypes are used to describe the objects with composite structure, queries based on the values of member(s) of compound data object(s) should also be included.
Similarly, we can also categorize all the query conditions into three types: *index-based, coordinate-based*, and *content-based* conditions. The goal of our system is to support any combination of the above three types of queries.

### 2.2.2 System Overview

Figure 2.1 gives an overview of the execution flow of processing a typical query using our system. The purpose of our system is to translate an SQL query into efficient data access code in HDF5 context and to complete the query by parallel access. After an SQL query passes through an SQL parser, the SQL parser extracts all the paths (from the *FROM* clause) of HDF5 files involved in that query, and then it sends this information to a metadata generator. Afterwards, the metadata generator accesses to all the required HDF5 files according to the paths retrieved by the SQL parser, and then it generates metadata at runtime. The detailed information about the metadata and the metadata generation strategy will be introduced in Section 2.3.

The SQL parser is used to generate a 2-dimensional query list. First, all the elementary queries in the same *AND* clause will be organized in the same 1-dimensional query list. Second, all these 1-dimensional lists will be grouped in an OR-logic and hence organized in a 2-dimensional list. Our parser is implemented with certain modifications of the parser from SQLite [12].

The hyperslab selector module mainly performs three tasks: First, for all the coordinate values that appear in the queries, convert them into index values specific to the physical layout, by retrieving corresponding dimension scales from the metadata. Second, for each 1-dimensional query list organized in an AND-logic, combine all the elementary queries into a single composite (and probably reduced) query. Third, with the data layout
information provided by the metadata and the index value(s) provided in the query condition, each composite query can generate a hyperslab on the hyperspace. As a result, a collection of hyperslabs are selected, and a final 1-dimensional query list is generated. The detailed information about the hyperslab selector will be discussed in Section 2.4.

The query partition module is used to divide each query in the 1-dimensional query list into multiple subqueries. Each subquery is a query with the same content-based condition as the one in the original query, yet responsible for a disjoint subset of the original hyperslab. Thus, each process executes one subquery, and hence collective I/O can be used to achieve better performance compared with independent I/O. The detailed information about the query partition will be discussed in Section 2.4.
With the hyperslab information generated during the previous phases, the query processor is used to invoke native HDF5 calls to perform data accesses. After query processing, each subquery retrieves the required data block, but all the content-based conditions, which are dependent on data values, still remain to be checked. Content-based filter then performs a full scan over the data block retrieved after query processing, to filter out the data elements which cannot satisfy the condition. However, since aggregation query also requires the same full scan, content-based filtering actually can be performed while processing aggregation queries. In this way, there will be always one full scan at most during the entire execution flow.

Finally, each process obtains a local query result. For aggregation queries, a query combination is needed to generate the global query result. For non-aggregation queries, the combination is considerably expensive because it requires massive data transfer. Fortunately, most applications only require each process to output all local results in a specified order.

2.3 Metadata Generation Strategy

In this section, we discuss the design and implementation of the metadata generation strategy.

For each input query, a metadata generator is used to collect metadata at runtime, without requiring a loading process beforehand. The function of the metadata generator is to collect dataset physical storage information, dataset logical layout information, and user metadata (i.e. intrinsic header metadata) stored in the header. The motivation for this design is three-fold: First, a large HDF5 dataset may be distributed among multiple files. Query processing requires dataset physical storage information from each of these files,
and therefore, it is desirable to collect this information at one place. Second, certain dataset logical layout information such as dataset dimensions and dimension scales is often frequently required during query processing. Again, prefetching this information in advance is desirable, so we can avoid repeated I/O requests during query processing. Third, HDF5 datasets are organized in a hierarchical structure, so the user metadata may be dispersed in separate header blocks for each object. Therefore, collecting the scattered user metadata can also help reduce the header I/O overhead.

As Figure 2.2 shows, the metadata structure consists of three components: 1) Physical Storage Descriptor: It describes physical locations where each dataset is resident. All the paths of the required dataset files are specified in the FROM clause in an SQL query, thus
by recording the datasets contained in these files, each dataset involved in the query can be mapped to a set of corresponding file paths. 2) Logical Layout Descriptor: It exposes the low-level dataset layouts, including datatypes, dataspaces, and dimension scales. Specifically, dimension scales, which are relatively quite small datasets, are fully loaded to support the queries that are based on coordinate values. 3) User Metadata Descriptor: It contains the descriptions about user metadata, which is optionally defined and provided by the users. Some user-provided information such as minimum/maximum dataset values can facilitate the later processing such like hyperslab selection.

2.4 Query Execution

In this section, we discuss our query execution methods, including hyperslab selection, content-based filtering, processing of aggregation queries, and parallelization of both selection and aggregation queries.

2.4.1 Hyperslab Selection and Content-Based Filtering

Recall that in Section 2.2, we had identified three types of queries, which are queries based on dimensional index values, coordinate values, and data values, respectively. The standard HDF5 API can only support the first type of queries (based on dimensional index values). Even for these queries, one has to know the exact index range, which, in turn, requires a detailed understanding of the data layout. As the high-level HDF5 feature dimension scale becomes increasingly popular in real applications, a growing number of queries involves subsetting on spatial and/or temporal attributes, which are queries based on coordinate values. Currently, for such queries, users have to first query the dimension scales, 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
In our system, we use **hyperslab selector** combined with **content-based filter** to both improve efficiency and support higher-level operators. Figure 2.3 shows an example of how the **hyperslab selector** module works. In the example, we queried a 4-dimensional dataset called **salinity**, where 4 dimension scales time, cols, rows and layers are associated with these 4 dimensions, respectively. In an SQL query, all the target dataset names are required to appear in the **SELECT** clause, and all the required HDF5 file names have to be specified in the **FROM** clause.
After the initial query parsing, the SQL parser generates a 2-dimensional query list based on the query condition. As we mentioned earlier in Section 2.2, for a 2-dimensional query list, the lower dimension is an AND-logic dimension, and the higher dimension is an OR-logic dimension. Any original SQL condition can be broken into multiple elementary queries, each of which is organized in this fashion, and each elementary query corresponds to only one condition type. For instance, for the first row in the query list after query parsing, \( \text{dim1} \) refers to the index value of first dimension so that \( \text{dim1} \geq 300 \) is an index-based condition, time < 740000 is a coordinate-based condition, and salinity < 70 is a content-based condition.

In the next step, query reduction is performed to reduce the number of elementary queries. The goal of this phase is to avoid unnecessary or redundant data accesses in the query processing. We perform the query reduction mainly based on the following three query reduction rules: 1) If an elementary query condition is unconditionally false, then the 1-dimensional query list where this query condition belongs can be nullified; 2) If an elementary query condition is unconditionally true, then this single query condition in this dimension can be nullified; and 3) For any two elementary conditions in the same AND-logic dimension, if either query range is entirely covered by the other, then the condition with the larger query range can be nullified. Since all the boundary values of indices and dimension scales have been stored in the metadata, we can apply the above rules without loading the dataset. In the example, since the maximum values of layers and cols are 33 and 166 respectively, the second 1-dimensional query list which contains layers > 60 can be removed according to the first rule, and cols < 200 can also be removed from the last 1-dimensional query list because of the second rule. Specifically, the third rule is often applied after the later dimension scale mapping phase. Assume that
the query condition time<740000 in the first 1-dimensional query list is time>740000 instead, since time is associated with the first dimension, this query condition is equivalent to the index-based condition dim1>1000. Because this query range is entirely covered by another directly-specified query range dim1>=300 in the same query list, we can nullify the larger query range dim1>=300 based on the third rule. Additionally, since HDF5 library allows one dataset dimension to be associated with multiple dimension scales, a variance of the above case is a query condition involves two different dimension scales which yet correspond to the same dimension. Moreover, besides the boundary values of indices and dimension scales, if the minimum/maximum dataset values are provided in the original user metadata, then we can also apply the above rules, in this case using these boundary dataset values which would have been stored as part of the metadata. As a result, certain content-based conditions can be nullified, and hence the workload of the later content-filtering can be reduced.

During the dimension scale mapping phase, all the coordinate-based conditions are converted into index-based conditions. In the example, we can see that the coordinate value rows has been updated to its corresponding dimensional index value after this step. Followed by this mapping, an AND-logic combination is performed to combine any two joint query ranges in the same dimension into a single query range. As dim1>=300 and dim1<1000 are combined together, only an intersection of multiple joint query ranges remains.

In the hyperslab generation phase, all the index boundary values are added to the existing query ranges. Consequently, the previous 2-dimensional query list is reduced into a 1-dimensional query list, which is essentially a collection of hyperslabs with
content-based condition(s). With the generated hyperslabs, HDF5 APIs can be invoked to perform an initial subsetting.

Finally, for the content-based queries, a content-based filter is designed to handle the remaining content-based condition(s). The filter will scan the data subsets that correspond to the generated hyperslabs and extract the data elements that meet the remaining query constraints.

### 2.4.2 Parallelization of Query Execution

![Figure 2.4: Parallelism of Query Execution](image)

As shown in Figure 2.4, we exploit the parallelism of query execution mainly at two levels. On one hand, the high-level parallelism is mainly explored based on the physical
storage information provided by the metadata. For an SQL query, a *FROM* clause may provide multiple file paths, and a *SELECT* clause may provide multiple target dataset names as well. With the information retrieved from the *physical storage descriptor* in the metadata, we identified all the target datasets in each file. After the *hyperslab selection* phase, as described earlier, a 1-dimensional query list is generated for each target dataset. Since collective I/O operation is supported by HDF5 library, a group of processes can perform one query instance collectively, over different disjoint data subsets of the same file.

On the other hand, the low-level parallelism is exploited via the *query partition*. For each 1-dimensional query list, we first break it into multiple subqueries where each corresponds to one hyperslab. Afterwards, *query partition* module works on each individual subquery serially. According to the number of processes in the running environment, the hyperslab associated with each subquery is divided into the equal number of disjoint partitions, and then each partition is assigned to one processor so that the subquery can be processed in parallel.

Given that different partitioning strategies can result in vastly different I/O performance, we used an optimized partitioning strategy in the query partition. Generally, we perform partitioning based on the highest dimension as much as possible. In this way the original data contiguity can be protected to the largest extent, and hence more contiguous data accesses can lead to a better I/O performance.

After the partitioning, each processor invokes HDF5 APIs via the *query processor* to load the corresponding partition. If any content-based condition is attached to the subquery, then the *content-based filter* will be launched during the partition loading. Followed by the partition loading and possible content-based filtering, if data aggregation is involved
in the query, then a local combination will be performed to obtain the aggregation result from a subquery. Since all the subqueries are organized in an OR-logic, finally a global combination is performed to acquire the union set of all the local combination results.

### 2.4.3 Aggregation Queries

Currently, HDF5 libraries do not support data aggregation. However, in many user scenarios, this functionality is highly desirable. Scientists currently have to perform an aggregation by writing their own code, which can be quite challenging. Furthermore, the data volume before aggregation may be much larger than that after aggregation. Therefore, in a wide-area environment, the current approach often leads to a much higher volume of data transfer.

Our system addresses this problem by proposing two-phase data aggregations, i.e., *local aggregation* and *global aggregation*. Our implementation works as follows. The master process first takes a query as input and check if this query has the data aggregation requirement. The input query is partitioned into a collection of subqueries which are then assigned to different slave processes. Each process executes a subquery and then stores the *local aggregation* results in a sequence of buckets. Finally, a *global aggregation* is performed by the master process.

Algorithm 1 shows the pseudo-code of local aggregation. Each subquery comprises *hyperslab*, *content-basedCondition*, *groupbyIndex* and *aggregationType*. Within each *hyperslab*, different *aggregationGroups* are formed based on the different index values in the group-by dimension. By calling different aggregation functions with respect to different *aggregationTypes*, each slave process performs aggregation over all the *aggregationGroups* within the corresponding *hyperslab* serially. In the meanwhile,
content-based filtering is also launched during the same iteration according to the content-basedCondition associated with each subquery. Consequently, each partial aggregation result will be stored in a corresponding bucket in results.

After the local aggregation, each slave process will send the partial aggregation results to the master process, which then will perform a global aggregation. Additional merge information is needed for the global aggregation, and this information is generated by the query partition module before dispatching all the subqueries.

2.5 Experimental Results

In this section, we evaluate the functionality and scalability of our system on a cluster of multi-core machines. We designed the experiments with the following goals: 1) to compare the functionality and performance of our system with OPeNDAP, a scientific
data management system widely used to support data dissemination for environmental and oceanographic datasets [62], 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 demonstrate how data aggregation queries reduce the data transfer cost.

Our experimental datasets were generated from a real database available to the scientific community, the Mediterranean Oceanic Data Base (MODB)\(^3\). The MODB data mainly consists of two separate datasets, salinity and temperature, which were generated from a simulation for a 34-layer space in the Mediterranean sea. A sample data file available for download is modeled by 167 columns and 63 rows\(^4\). Because we did not have the access to the real database, we extrapolated this data by extending a time dimension to create larger datasets. To demonstrate that our system can also support compound datatype, we created a compound dataset cell, which comprises both salinity and temperature in the same simulation space. HDF5 version 1.8.7 was used for all our experiments.

### 2.5.1 Sequential Comparison with OPeNDAP

OPeNDAP provides data virtualization through a data access protocol and data translation mechanism. OPeNDAP supports data retrieval in various formats, including ASCII, DODS binary objects, and various other common formats. Specifically, one component HDF5-OPeNDAP data handler was developed so that general HDF5 data can be served via the OPeNDAP software framework and hence be used for our comparisons.

\(^3\)http://modb.oce.ulg.ac.be/
\(^4\)http://www.space-research.org/save\_hdf5.htm
OPeNDAP has several limitations. First, in order to hide the data formats and data processing details, OPeNDAP requires to translate the original data format into a standard OPeNDAP data format, which leads to undesirable overheads. Besides, neither relational constraint expressions (query based on dimension scales or data values) nor data aggregation is supported in OPeNDAP. Note that our system can execute data queries in parallel, whereas OPeNDAP can only support sequential query execution. 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 8 GB of memory, with Intel(R) Core(TM) i7-2600K 3.40 GHz CPU. The size of each data file was 4 GB.

As we mentioned in Section 2.2, there are three different types of queries: queries based on dimensional index values (type 1), queries based on coordinate values (type 2), and queries based on data values (type 3). Since OPeNDAP can only support queries based on dimensional index values for arrays and grids (i.e. the type 1 queries), we designed separate experiments for type 1 queries and type 2 and 3 queries.

![Figure 2.5: Performance Comparison for Type 1 Queries](image)

Figure 2.5: Performance Comparison for Type 1 Queries
In the first experiment, we only considered type 1 queries and compared our system with OPeNDAP. We generated 100 type 1 queries, which included subsetting based on different dimensions. As Figure 2.5 shows, the results were classified into 5 groups based on different proportions(<20%, 20%-40%, 40%-60%, 60%-80%, >80%) of data subset coverage.

The first bar in each group shows the execution time of intrinsic HDF5 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 dimension scale mapping or content-based filtering cost, the extra processing overhead is mainly brought by SQL parsing, metadata generation, and hyperslab selection, where these overheads are all quite small. We can see that the total sequential processing time for type 1 query is not distinguishable from the baseline or the intrinsic HDF5 query function. The third bar shows the total execution time of OPeNDAP. We can see that OpenDAP is consistently much slower, and moreover, scales poorly as more data has to be output, mainly due to the additional data translation overhead.

The next experiment involved type 2 and type 3 queries. Since OPeNDAP is unable to support relational constraint expressions for arrays and grids, user client has to download the entire data from the server first and then write its own filter to generate data subset results. Thus, we implemented a client-side filter for OPeNDAP and then used this modified version to compare with our system. In this experiment, we generated 100 queries, including the ones that combine type 2 and type 3 queries. Table 2.1 presents several examples of type 2 and type 3 queries.

Figure 2.6 shows the execution time over different queries. The first bar in each group is the baseline of our current experiment, which contains two sub-parts, execution time
Table 2.1: Type 2 and Type 3 Query Examples

<table>
<thead>
<tr>
<th>ID</th>
<th>Query Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL1</td>
<td>SELECT salinity FROM MODB WHERE time &gt; 589530 AND time &lt; 995760 AND salinity &gt; 32.4;</td>
</tr>
<tr>
<td>SQL2</td>
<td>SELECT temperature FROM MODB WHERE layers &lt;= 20 OR layers &gt;= 30 AND temperature &lt; 10.0;</td>
</tr>
<tr>
<td>SQL3</td>
<td>SELECT salinity, temperature FROM MODB WHERE (rows &gt; 15 OR rows &lt; 50) AND (cols &lt; 120 OR cols &gt;= 40);</td>
</tr>
<tr>
<td>SQL4</td>
<td>SELECT cell FROM MODB WHERE time &lt; 589530;</td>
</tr>
</tbody>
</table>

Figure 2.6: Performance Comparison for Type 2 and Type 3 Queries

of the intrinsic HDF5 query function, and the content-based filtering time. Since HDF5 library does not support subsetting based on data values, content-based filtering requires a full scan over the hyperslabs selected by the hyperslab selector. This baseline execution time is proportional to the amount of data subset. The second bar represents the execution time of our system. For type 2 and type 3 queries, we can see that it takes almost the same amount of time to perform hyperslab selection compared with the baseline, because the overhead of the additional dimension scale mapping is still quite trivial. The third bar
shows the execution time of OPeNDAP. Without hyperslab selection support for subsetting, OPeNDAP has to scan the entire dataset to perform content-based filtering. Thus, both the data query cost and the data filtering cost are much higher than our system.

2.5.2 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 experiments were conducted on a cluster of multi-core machines. The system uses AMD Opteron(TM) Processor 8218 with 4 dual-core CPUs (8 cores in all). The clock frequency of each core is 2.6 GHz, and the system has a 12 GB memory. We have used up to 128 cores (16 nodes) for our study.

![Figure 2.7: Parallel Query Processing Times with Different Dataset Sizes](image)

The first experiment evaluates the performance of parallel subsetting with different dataset sizes. In this experiment, the input queries simply fully scan the datasets. The experimental data sizes vary from 4 GB to 32 GB, while the number of nodes used for
parallel subsetting varies from 1 to 16. Figure 2.7 shows the results as we scale the number of nodes. We can see that our system is capable of scaling the performance of processing such queries.

Figure 2.8: Parallel Query Processing Times with Different Queries

In the second experiment, we evaluated the performance of parallel subsetting. With different 3 types of query conditions, we generated 300 queries which cover different proportions of the dataset based on different dimensions. Figure 2.8 presents the results, which are divided into 5 coverage proportion groups. The experimental data sizes were 16 GB, and the number of nodes used for parallel subsetting varied from 1 to 16. We can see that with our optimized partitioning approach, which leads to a better data contiguity of partitions, the parallelization time for different queries is approximately proportional to the amount of subsetting data.

Since the previous experiments didn’t involve any type 3 query on compound dataset, we conducted the third experiment to compare the performance of content-filtering over atomic datasets against compound datasets. In order to highlight the performance gap
between the type 3 queries on atomic datasets and on compound dataset, we generated 100 queries which perform a pure content-filtering over the compound dataset. Figure 2.9 presented the results as we scaled the number of nodes from 1 to 16 for 16 GB datasets. We can see that the overhead of content-filtering over compound datasets is approximately as twice high as the one over atomic dataset. This is because that, although content-filtering is performed based on the value of a single data member within a compound data element, the entire compound data element has to be loaded first. By contrast, the content-filtering over atomic datasets can avoid the cost of loading the query-unrelated data (i.e. other data member(s) within a compound data element), since different atomic data is stored in separate datasets. The reason why the overhead ratio is close to 2 is that, in our scenario, the size of the experimental compound datatype is accidentally as twice large as either experimental atomic datatype. We believe that if the compound datatype consists of more data members, or the data member sizes become larger, the performance gap might become huger accordingly.

Figure 2.9: Performance Comparison of Content-Filtering over Atomic Datasets and Compound Datasets
Table 2.2: Aggregation Query Examples

<table>
<thead>
<tr>
<th>Type</th>
<th>Query Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG1</td>
<td><code>SELECT AVG(temperature), AVG(salinity) FROM MODB;</code></td>
</tr>
<tr>
<td>AG2</td>
<td><code>SELECT COUNT(temperature) FROM MODB GROUP BY time HAVING time&gt;=319288;</code></td>
</tr>
<tr>
<td>AG3</td>
<td><code>SELECT MAX(temperature), MIN(temperature), MAX(salinity), MIN(salinity) FROM MODB GROUP BY layer;</code></td>
</tr>
</tbody>
</table>

This subsection is designed to compare both the data transfer cost and the data aggregation performance between OPeNDAP and our system, by performing different types of data aggregation. Recall that OPeNDAP does not support data aggregation. Thus, to perform data aggregations with OPeNDAP, users have to first download all the data from the server side and then develop its own code to perform the aggregation at the client side. In contrast, our system supports parallel server-side data aggregation which can lead to a much smaller cost for both data transfer and data aggregation.

Table 2.2 shows aggregation types and examples used in our evaluation. We generated 100 aggregation queries and categorized them into three types: 1) AG1: query only includes aggregations; 2) AG2: query includes both aggregations and `groupby` with `having` subsetting; and 3) AG3: query includes both aggregations and `groupby` without `having` subsetting. To highlight the efficiency improvement of parallel data aggregation, we didn’t involve any `WHERE` clause in the queries in this experiment. The sizes of the experimental datasets were 16 GB.
Table 2.3: Data Transfer Volume Comparison against OPeNDAP

<table>
<thead>
<tr>
<th>Aggregation Type</th>
<th>Avg. Data Transfer Volume (byte)</th>
<th>OPeNDAP</th>
<th>Our System</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG1</td>
<td>15,979,863,952</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>AG2</td>
<td>15,979,863,952</td>
<td></td>
<td>188</td>
</tr>
<tr>
<td>AG3</td>
<td>15,979,863,952</td>
<td></td>
<td>2040</td>
</tr>
</tbody>
</table>

One advantage of our parallel server-side data aggregation is the significant reduction in data transfer volume. This advantage is very likely to bring about considerable performance improvements when the server and the client need to communicate over the wide area network. Table 2.3 compares the total data volume that is transferred over the network between OPeNDAP and our system. We can see that, regardless of aggregation type, the entire dataset has to be transferred over the network before OPeNDAP performs data aggregation. On the other hand, in our system, AG1 has the smallest data transfer cost. This is because as no GROUP BY clause is involved, only a grand aggregation result is required to be transferred. Moreover, AG2 has smaller data transfer volume than AG3, because AG2 involves HAVING clause that will reduce the number of aggregation groups, and hence a smaller number of aggregation results are transferred. To conclude, for all the three aggregation types, our system demonstrated a tremendously smaller data transfer cost than OPeNDAP.

Besides the performance improvement in data transfer, our method can lead to a much smaller data aggregation cost. Our last experiment compared the data aggregation performance of OPeNDAP with our parallel system. We used up to 16 nodes to perform different types of aggregation queries over 16 GB datasets. The results presented in
Figure 2.10 demonstrated a good speedup of our parallel aggregation approach against the OPeNDAP’s sequential aggregation, and a good scalability as the number of nodes increases. The relative speedups are demonstrated up to 8.13, 7.35 and 6.55, for the three types of aggregation queries, respectively.

2.6 Related Work

The topics of scientific data management have attracted a lot of attention in recent years. Because of the large volume of work in this area, we concentrate on work specific to the tools or approaches that can be used with scientific data formats like HDF5, and then we provide a brief overview of the most significant other work.

OPeNDAP [62] supports data virtualization via a data access protocol and data representation. We have conducted extensive comparison of functionality and performance between our system and OPeNDAP. Barrodale Computing Services (BCS) [17] has developed a plug-in called the Universal File Interface (UFI) [13], which establishes a virtual table interface to allow Informix DBMS to transparently access HDF5
files as they were relational tables. However, it can support neither data aggregation nor parallel query processing. The NetCDF Operator (NCO) library [9] and its parallel implementation [223] have been extended to support parallel query processing over NetCDF files. The queries supported by this tool have to be expressed in very specialized scripting languages, which unfortunately severely undermines their ease-of-use. MLOC [76] can optimize the storage layout of scientific data and hence facilitate queries based on coordinates, data values and even precision. Besides that, it can support precision-based query, which can lead to a reduced I/O and computation overhead if lower resolution of data is required. However, it requires beforehand data transformation which may be prohibitively expensive, while our approach can directly manipulate on the original dataset. SciHadoop [42] and SciMATE [237] have integrated map-reduce and its variant with scientific library to enable map-reduce tasks over scientific data. While map-reduce can be used to implement selection and aggregation queries, SQL queries are simpler to write.

On the other hand, besides subsetting and aggregation, more well-known database techniques have been applied to scientific data management. Scientific Data Manager (SDM) employs the Meta-data Management System (MDMS) [166] and provides a high-level, user-friendly interface which interacts with database, to hide low-level parallel I/O operations for complex scientific processing. In comparison, our approach is more light-weight, and it can serve more HDF5 features such as dimension scale, compound datatype, and user metadata. FastQuery [107] utilizes parallel bitmap indexing to accelerate searches on scientific data files. By contrast, we focus on supporting various types of queries over scientific dataset with a standard high-level interface. However, their work does not support a high-level API, or hide the indexing and NetCDF details from
users. In the future, we will extend our work by incorporating more database techniques such as indexing.

Scientific data management has drawn much attention lately. Several groups have articulated the requirements in this area [19, 79, 163]. Recently, there has been a significant interest in extending (relational) database technology to support the need of (extreme-scale) scientific data [207], leading to multiple Extreme Scale Databases Workshops (XLDB). The requirements arising have been summarized in a position paper by Stonebraker [199], 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 into a database system. Our approach also provides a light-weight solution, which can be made available to different application domains more easily. This work can be viewed as an implementation of the more general data virtualization approach introduced in the work [238]. More recently, we reported a similar implementation for NetCDF [200]. The specific contributions of this work are in handling HDF5 features, like the more distributed metadata and the notion of hyberslabs. The design reported in this work can optimize more complex queries also.

2.7 Summary

This work describes implementation of a light-weight data management approach for scientific datasets stored in HDF5, which is one of the most popular scientific data formats. Our tool supports parallel data subsetting and aggregation in a virtual data view that is specified by SQL. Besides supporting selection over dimensions, which is also
directly supported by HDF5 API, our system also supports queries based on *dimension scales* and/or *data values*. 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 the queries, and reducing wide area data transfers through server-side data aggregation.
In contrast to the previous chapter, we realize that it is more natural to manage array-based scientific data in an array-view rather than a relational table view. Particularly, we focus on implementing a number of structural aggregations, which are unique in array data model. All the operations are still supported in a light-weight manner - over native array storage.

3.1 Structural Aggregations

This section lists different types of structural aggregations that arise in scientific applications and are supported in our system.
Table 3.1: Structural Aggregation APIs and Mathematical Definitions

<table>
<thead>
<tr>
<th>API</th>
<th>Mathematical Definition (for the Aggregation Operator SUM in a 2-Dimensional Space)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gridAgg</code> (array slab $S$, grid size $g$)</td>
<td>$\forall (x, y) \in S, (x - x_0) \mod g = 0 \text{ and } (y - y_0) \mod g = 0: \sum_{i=0}^{g} \sum_{j=0}^{g} S(i, j)$</td>
</tr>
<tr>
<td><code>slidingAgg</code> (array slab $S$, grid size $g$, stride $s$)</td>
<td>$\forall (x, y) \in S, (x - x_0) \mod s = 0 \text{ and } (y - y_0) \mod s = 0: \sum_{i=0}^{g} \sum_{j=0}^{g} S(i, j)$</td>
</tr>
<tr>
<td><code>hierarchicalAgg</code> (array slab $S$, initial radius $r$, step size $s$)</td>
<td>$\forall (x, y) \in S, (x - r) \mod s = 0 \text{ and } (y - r) \mod s = 0: \sum_{i=0}^{g} \sum_{j=0}^{g} S(i, j)$</td>
</tr>
<tr>
<td><code>circularAgg</code> (array slab $S$, initial radius $r$, step size $s$)</td>
<td>$\forall (x, y) \in S, (x - r) \mod s = 0 \text{ and } (y - r) \mod s = 0: \sum_{i=0}^{g} \sum_{j=0}^{g} S(i, j)$</td>
</tr>
</tbody>
</table>

3.1.1 Background: Structural Groupings and Structural Aggregations

Grouping has been one of the most important operations provided by databases, most often specified by a GROUP BY clause in standard SQL. Additionally, aggregation functions and corresponding aggregation filters can be added to the grouping. Most common grouping used in a database management system is the value-based grouping, where elements of the same value comprise one group. Over years, SQL extensions, particularly SQL:2003, introduced structural grouping, where elements are mapped to a group based on positional relationships. It turns out that such groupings are very common when arrays are the main structure. For example, calculating a simple moving average over array elements involves a positional relationship (i.e., a set of adjacent values).

In this work, we refer to aggregations that are based on any structural or positional grouping as structural aggregations - they include not only traditional aggregates such as COUNT, SUM, AVG, as well as MIN and MAX, but also user-defined aggregation functions for domain-specific analysis. For our presentation, we consider structural aggregations of three types, which are the grid, sliding, and hierarchical (or circular) aggregations, described in the following paragraphs.
The structural aggregations we consider process a rectilinear section of any size and dimensionality from an array (or an array slab). Unlike the aggregations based on array access patterns, which can load multiple array subareas into one group and then return a single aggregation result, structural aggregations divide an array slab into multiple and possibly overlapping groups and then return multiple corresponding aggregation results at one time. Structural aggregations can also be combined with value-based filtering conditions, as we will show through an example later.

Figure 3.1 shows the structural aggregation examples graphically, while the APIs and mathematical definitions are summarized in Table 3.1. This description assumes that we have a 2D array slab $S$, where the upper left element, the centroid, and the number of elements are denoted as $(x_0, y_0)$, $(x_c, y_c)$, and $|S|$, respectively. For simplicity, we assume that the aggregation operator is SUM.

### 3.1.2 Grid Aggregation

Grid aggregation is widely used in many scientific applications. It involves dividing an array slab into multiple disjoint smaller blocks and then aggregating values within each smaller block. For example, in astrophysics, vast astronomy events are stored in massive multi-dimensional arrays. The data is often broken into smaller disjoint spatial grids of equal grid size, and a corresponding multi-dimensional histogram can be produced by binning the events over those spatial grids [116]. In Figure 3.1(a), a $4 \times 4$ array is split into 4 non-overlapping $2 \times 2$ grids, and then values within each smaller grid are aggregated.

### 3.1.3 Sliding Aggregation

Sliding aggregation calculates a sequence of aggregates within an array slab through a sliding grid of a fixed grid size. The grid moves from a starting element to an ending
element with a \textit{stride}, where the stride value is 1 by default. Sliding aggregation also arises in a number of scientific domains. For example, in earth sciences, it is common to apply certain image denoising algorithms, such as non-local means (NL-means), to preprocess satellite imagery data \cite{40,249}. More specifically, a Gaussian kernel function is applied on a \textit{sliding window} to smooth out the outliers. As Figure 3.1(b) illustrates, a $3 \times 3$ grid slides within a $4 \times 4$ array in row-major fashion.

\subsection*{3.1.4 Hierarchical and Circular Aggregations}

In space sciences, scientists may be interested in looking into the gradual influence of radiation from a source, which may be a pollution source or an explosion location, over its adjacent region. A set of regions of increasing radii (or sizes) are analyzed for this purpose. For hierarchical aggregation, the centroids of all the grids is the same as the centroid of the entire array slab, and each grid is entirely covered by a concentric outer grid. The radius of the innermost grid is specified by an \textit{initial radius}. For the other grids, the radius increases by a fixed \textit{step size}, until it reaches the array slab boundary. In Figure 3.1(c), three concentric grids are aggregated in a $6 \times 6$ array: the innermost one is a $2 \times 2$ grid, the middle one is a $4 \times 4$ grid, and the outermost one is the entire array.

A variant of such hierarchical aggregation is \textit{circular aggregation}, which calculates a sequence of aggregates of concentric but \textit{disjoint} circles instead of regularly shaped grids. Figure 3.1(d) illustrates a circular aggregation that corresponds to the hierarchical aggregation shown in Figure 3.1(c).

There is a difference between grid aggregation and three other aggregations we have introduced so far. Particularly, we can also refer to grid aggregation as \textit{non-overlapping} aggregation, since each input value is used towards aggregating a disjoint grid. In
comparison, sliding and hierarchical aggregations are all forms of overlapping aggregation. For all practical purposes, a circular aggregation is also an overlapping aggregation, since no library for scientific arrays provides an interface to exactly load a circle of array elements at a time.

3.2 Grid Aggregation Algorithms

We now focus on parallel algorithms for performing grid aggregations. We assume that array is stored in a shared file system from which processes on different nodes and cores can access this data. This model matches the configuration of many high performance systems today, and also supports parallelization across cores within a node and nodes in a cluster/cloud in a uniform fashion. As stated previously, we also assume that array has been stored in a native form that is not controlled by our system. This leads to unique challenges in our work, since a system such as SciDB that has a data ingestion phase reorganizes the array during such a phase, and therefore, algorithms involved can be quite different.

Given an array slab, it is relatively easy to implement sequential grid aggregation, but efficient parallelization involves several challenges. To understand possible challenges, we first consider a simple approach for parallelization.

3.2.1 Outline of a Simple Parallel Grid Aggregation Algorithm

The outline of the parallel grid aggregation method is shown as Algorithm 2. According to the grid size specified by the user, a set of grids are initialized and grids within this set are then partitioned. Note that a single grid may be within one partition or may be further divided, with sub-grids assigned to different partitions (or processors), since such partitioning is not performed specific to a particular query. There are two phases after the partitioning, i.e., a local aggregation phase and a global aggregation phase.
phase. In the local aggregation phase, each processor is responsible for aggregating each grid in its own partition. In the global aggregation phase, for those grids distributed among different partitions, one of the processors serves as a *master processor* and merges the local aggregates of all partial grids.

**Algorithm 2: GridAgg(\textit{slab }S, \textit{num\_procs }n)**

1: Initialize \( m \) grids \( G \) and \( m \) corresponding aggregates \( A \)
2: Divide \( S \) into \( n \) partitions (set \( P \))
3: Each grid \( G_i(i = 1, \ldots, m) \) is assigned to one or more partitions
4: Each processor loads its partition \( P_j(j = 1, \ldots, n) \)
5: for each (possibly partial) grid \( G'_i \) in \( P_j \) do
6: \( A_i \leftarrow \text{aggregate}(G'_i) \) \{local aggregation\}
7: \text{if } G'_i \neq G_i \text{ then}
8: \quad Merge \( A_i \) of all partial grids \{global aggregation\}
9: \text{end if}
10: end for
11: return \( A \)

This simple algorithm has multiple advantages. First, by ensuring that each element is read by only one process, we avoid any redundant reads or computation. Second, workload distribution is static, and therefore, there is no runtime overhead of assigning work to processors.

### 3.2.2 Challenges and Different Partitioning Strategies

The challenges in executing the algorithm effectively and achieving high parallel efficiency are: 1) how to ensure that the set of partitions to be processed by a particular process are physically contiguous on the disks, so as to minimize disk seek times, 2) how to balance the workload across processes when the datasets are skewed - particularly when a *value-based filtering condition* is added to a compute-intensive aggregation (as
explained further below), and 3) how to minimize the communication cost when calculating global aggregates. The specific importance of each depends upon the nature of the queries.

Many aggregates of interest for scientific applications are domain-specific, user-defined, and compute-intensive (possibly even quadratic or cubic with respect to the grid size). Now, to further explain the second challenge we listed, consider the following query: suppose the aggregation involves computations over all the elements that are above a certain threshold (or not equal to a pre-defined missing value) within each grid. The use of this threshold (or excluding missing values) is what we also referred to as a value-based filtering condition above. Because of the distribution of values across different partitions, the number of arithmetic operations may vary significantly.

Figure 3.2: Examples of Different Partitioning Strategies
To address the above challenges, we have developed different partitioning strategies, each of which addresses one or more of the challenges.

**Coarse-Grained Partitioning:** Coarse-grained partitioning divides the given array slab into multiple equal-sized partitions, each of which comprises contiguous grids. As an example, see Figure 3.2(a), where 10 grids in a 2D array slab are partitioned for 4 processors, with grid 3 evenly distributed to processors 1 and 2, and grid 8 to processors 3 and 4. Broadly, the array slab is evenly partitioned, using the *highest dimension* (e.g., rows when the layout is row-major), so that the grids are contiguous. If the number of grids is divisible by the number of processors, then all the aggregates can be calculated locally. Otherwise, any grid that contains the partition boundary will be covered by two adjacent partitions, resulting in an extra (parallel) merge operation over each two local partial aggregates.

To summarize, this approach gives good I/O performance and low communication cost. On the negative side, in presence of data skew, this approach does not provide load balance.

**Fine-Grained Partitioning:** Fine-grained partitioning evenly distributes elements within each grid from the given array slab across all processors. An example can be seen from Figure 3.2(b), where a 2D array slab that consists of 10 grids is partitioned across 4 processors. Each processor calculates a partial aggregate for each grid during the local aggregation, and then the global aggregation phase involves an all-to-one or all-to-all reduction over all the local aggregates.

As the example shows, the partitioned data, although contiguous within the same row of grids, is scattered among different rows. Thus, the I/O performance will not be as good as with the coarse-grained partitioning, and the communication cost can also be higher.
However, a fine-grained distribution is likely to handle skew better than a coarse-grained distribution.

**Hybrid Partitioning:** Hybrid partitioning combines elements of coarse-grained and fine-grained partitioning. First, each partition covers an equal number of grids in the given array slab, with assignment in a round-robin fashion. If the number of grids is not divisible by the number of processors, the extra grids can be partitioned in a coarse-grained or fine-grained manner. As Figure 3.2(c) shows, 10 grids in a 2D array slab are partitioned into 4 processors. Starting from the grid 1, every 4 consecutive grids are mapped to 4 processors in order, and then grids 9 and 10 are partitioned in a coarse-grained manner.

Overall, hybrid partitioning has an advantage with respect to communication cost and dealing with skew. This is because most aggregates can be calculated locally, and round-robin distribution ensures that one is likely to be more skew-tolerant. However, grids within a single partition are not contiguous.

**Auto-Grained Partitioning:** Among the partitioning strategies we have presented so far, both fine-grained and hybrid schemes can deal with data skew. However, they do so in an ad-hoc fashion, by round-robin distribution of grids or grid elements across the partitions, which is likely to balance the data amount satisfying value-based predicates across processes. In comparison, the strategy we introduce now is designed specifically for handling data skew and workload imbalance. Rather than evenly partitioning the input data (and hence balance the data loading costs), auto-grained partitioning aims to equally divide the total processing cost. Thus, the size of each partition is not necessarily equal.

Auto-grained partitioning assumes we have a certain estimate of data skews for different grids, and thus, the different underlying processing costs. Overall, it works as follows. First, given a value-based filtering condition, a lightweight uniform sampling is applied at
runtime, to estimate the density of all the grids after filtering. In practice, such sampling is performed in parallel over different parts of the array slab. Second, based on the sampled density, for each grid we have an estimate of the total processing cost, which is sum of the loading cost (fixed) and the computation cost (which depends upon the density and the computation complexity of aggregation). Thus, the input of the partitioning algorithm can be reconsidered as a one-dimensional cost array, where the value of each element is the estimated cost of a corresponding grid. This problem can be modeled as a balanced contiguous multi-way partitioning problem, where the objective is to find a sequence of separator indices to divide this cost array into contiguous partitions such that the maximum cost of all the partitions is minimized.

We have designed two different algorithms to solve this problem. The first approach is a dynamic programming algorithm with a $\Theta(N M^2)$-time complexity, where $N$ is the number of processors, i.e., the number of partitions, and $M$ is the number of grids. Other than $M$ and $N$, another input is the cost array of $M$ elements, denoted as $A$. The decomposed sub-problem $P(n, m)$ indicates the maximum partition cost after dividing the first $m$ elements of $A$ into $n$ contiguous subarrays in a balanced way, so that the output is reformulated as $P(N, M)$. The problem can be decomposed as follows:

$$P(n, m) = \begin{cases}  
 A_1, & \text{if } m = 1 \\
 \sum_{i=1}^{m} A_i, & \text{if } n = 1 \\
 \min \left( \max_{1 < k < m} (P(n-1, k), \sum_{i=k+1}^{M-m} A_i) \right) & \text{else}
\end{cases}$$

The second approach is a greedy $O(M)$-time algorithm to find a suboptimal partitioning. It can be completed with two passes over the cost array. First, we calculate the average partition cost with the given cost array and the number of partitions. Second, by linearly scanning the cost array, we remove a partition from the cost array when the accumulated cost is close to the average partitioning cost.
When the number of grids is relatively small, the system uses the dynamic programming algorithm to gain an optimal workload balance. When the number of grids is large, the system chooses the greedy algorithm. In this case, the greedy algorithm runs much faster, and it can produce a suboptimal solution that is almost as good as the optimal solution provided by the dynamic programming algorithm.

Overall, apart from the good workload balance, the auto-grained partitioning has a very low communication cost, since all the aggregates are calculated locally. Moreover, its I/O performance is also as good as the coarse-grained partitioning, as all the grids in the same partition are contiguous. The disadvantage is the overhead of parallel sampling and runtime partitioning.

3.2.3 A Cost Model for Choosing Partitioning Strategy

As we discussed above, the partitioning strategies we have introduced involve trade-offs with respect to communication and data loading costs, as well as their ability to handle skew. To automatically choose the partitioning strategy that results in the lowest total processing cost, we have developed a cost-based partitioning strategy decider.

Broadly, we can divide the total processing cost of parallel grid aggregation into five parts: the initialization cost for result values, the loading cost for partitions to be read, the filtering cost, i.e., applying a value-based predicate before aggregation, the computation cost of calculating aggregates, and the communication cost of merging local aggregates. For simplicity, we can ignore both the initialization cost and the filtering cost, since the former is trivial and the filtering phase costs can be merged with those of the data loading phase. Finally, it turns out that with an exception for fine-grained partitioning with very small grid sizes, communication costs can also be ignored. To keep our model simple, we
### Table 3.2: Summary of Model Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{glb}$</td>
<td>App</td>
<td>Number of elements in total</td>
</tr>
<tr>
<td>$N_{loc}$</td>
<td>App</td>
<td>Number of elements in a partition</td>
</tr>
<tr>
<td>$N_{glbGrids}$</td>
<td>App</td>
<td>Number of grids in total</td>
</tr>
<tr>
<td>$N_{locGrids}$</td>
<td>App</td>
<td>Number of grids in a partition</td>
</tr>
<tr>
<td>$N_{grid}$</td>
<td>App</td>
<td>Number of elements in a grid</td>
</tr>
<tr>
<td>$N_{procs}$</td>
<td>App</td>
<td>Number of processors (also the number of partitions)</td>
</tr>
<tr>
<td>$P_{elem}$</td>
<td>App</td>
<td>Processing cost per element</td>
</tr>
<tr>
<td>$N_{f_{ldGrid}}$</td>
<td>Dataset</td>
<td>Number of elements after filtering in a grid</td>
</tr>
<tr>
<td>$F_{loc}$</td>
<td>Dataset</td>
<td>Local filtering factor for a partition</td>
</tr>
<tr>
<td>$F_{glb}$</td>
<td>Dataset</td>
<td>Global filtering factor for the entire dataset</td>
</tr>
<tr>
<td>$R$</td>
<td>Env</td>
<td>Ratio between the computation cost and the I/O cost for a single element</td>
</tr>
<tr>
<td>$f_{coarse}$</td>
<td>Env</td>
<td>Loading cost factor for the coarse-grained partitioning</td>
</tr>
<tr>
<td>$f_{fine}$</td>
<td>Env</td>
<td>Loading cost factor for the fine-grained partitioning</td>
</tr>
<tr>
<td>$f_{hybrid}$</td>
<td>Env</td>
<td>Loading cost factor for the hybrid-grained partitioning</td>
</tr>
<tr>
<td>$P_{hybrid}$</td>
<td>Env</td>
<td>Penalty factor for the hybrid-grained partitioning</td>
</tr>
<tr>
<td>$P_{auto}$</td>
<td>Env</td>
<td>Penalty factor for the auto-grained partitioning</td>
</tr>
<tr>
<td>$c_{samp}$</td>
<td>Env</td>
<td>Sampling cost for the auto-grained partitioning</td>
</tr>
</tbody>
</table>
assume that fine-grained partitioning is not an option when the grid sizes are small. Thus, we can simplify the total processing cost as:

$$C_{total} = C_{load} + C_{comp}$$

(3.1)

Here, $C_{load}$ represents the loading cost, and $C_{comp}$ is the computation cost.

The parameters used in the cost model are defined in Table 3.2. Those parameters can be categorized into three types: 1) Application Parameters, which correspond to the query parameters, including the given array slab, grid size, computation involved in calculating aggregates and other similar parameters, 2) Dataset Parameters, particularly, those reflecting data skew, which we assume we have broad knowledge of in advance, or acquire through a preprocessing round, and 3) Environment Parameters, which correspond to the architecture and disk system, which are learnt through sample runs.

Now, since all grids are dense, we have $N_{grid} = N_{glb}/N_{glbGrids}$. Second, with the exception of when auto-grained partitioning is used, the sizes of all the partitions are equal, i.e., $N_{loc} = N_{glb}/N_{procs}$ and $N_{locGrids} = N_{glbGrids}/N_{procs}$.

**Loading Cost Analysis:** The loading cost is clearly independent of the data skew. In addition, through extensive experiments, we also observed that this cost is almost independent of grid size ($N_{grid}$), unless the grid size is very small. At the same time, the loading cost does depend upon the partitioning strategy and the partitioned data size ($N_{loc}$), since they impact the disk seek times. Rather than modeling seek times analytically, we take an experimental approach, where the impact of disk seek for a particular partitioning strategy is measured for a particular environment. This cost is then represented as a weighing function, called the loading cost factor, and we estimate the loading cost by multiplying the loading cost factor with the size of partitioned data. We denote these loading cost factors by $f_{coarse}$, $f_{fine}$, and $f_{hybrid}$, for coarse-grained,
fine-grained, and hybrid partitioning, respectively. Then, the expression for the loading cost of a single partition, shown for coarse-grained partitioning as an example, will be

\[ C_{load}^{coarse} = f_{coarse} \times N_{loc} = f_{coarse} \times \frac{N_{glb}}{N_{procs}} \]  

(3.2)

In the environment we used, these three weighing functions were learnt using a set of representative queries. These queries did not involve the same array slab size or aggregation functions as the ones used in our experiments, and thus, our experiments validated the cost model. The specific values we obtained were: \( f_{coarse} \approx 12.2 \), \( f_{fine} \approx 19.5 \), and \( f_{hybrid} \approx 16 \).

For the auto-grained partitioning strategy, we estimate the total loading cost rather than the loading cost of a single partition. Because the partitioned grids are contiguous, the loading cost factor for coarse-grained partitioning can be used, and thus we have:

\[ \sum_{N_{procs}} C_{load}^{auto} = f_{coarse} \times N_{glb} \]  

(3.3)

**Computation Cost Analysis:** The computation cost for a partition is the sum of the computation cost for all the local grids. The computation cost for a grid is dependent upon the density of the grid as well as the nature of the aggregation computed (i.e., computation complexity). To capture the latter, the parameter \( P_{elem} \) denotes the per element processing cost for one grid aggregation. This parameter needs to be obtained for each aggregation function using sample datasets.

For the first three partitioning strategies, which evenly partition the input data, the overall computation cost should be the maximum cost among all the partitions, which can be formally stated as:

\[ C_{comp} = \max_{N_{procs}} \left( \sum_{N_{LocGrids}} R \times P_{elem} \times N_{f_{tdGrids}} \right) \]  

(3.4)

Here, \( R \) is the adjustable ratio between the computation cost and the I/O cost for a single element. The value of the parameter \( R \) is dependent upon the environment and can be
determined during a learning phase, similar to how the loading factor parameters are obtained. We now elaborate on how other parameters are estimated for each of the strategies:

**Coarse-Grained Computation Cost:** Since the number of elements to be processed (those left after filtering) differs across partitions in the case of coarse-grained partitioning strategy, the computation cost is dependent on the partition where the least amount of data is filtered out. Particularly, we use a parameter $F_{loc}$ to indicate the proportion of data after filtering in a given partition. We also have:

$$\sum_{N_{locGrids}} N_{ftdGrid} = F_{loc} \times N_{loc} \quad (3.5)$$

By using Equation 3.4 and 3.5, the coarse-grained computation cost can be estimated as follows:

$$C_{coarse_{comp}} = \max_{N_{procs}} (R \times P_{elem} \times F_{loc} \times \frac{N_{glb}}{N_{procs}}) \quad (3.6)$$

**Fine-Grained Computation Cost:** Because the fine-grained partitioning strategy is likely to evenly distribute the data after filtering, the local filtering factor of any partition is equal to the global filtering factor. Thus, we have $F_{loc} = F_{glb}$. Because of such even distribution of filtered data among all the partitions, fine-grained partitioning strategy evenly distributes the computation workload. Therefore, to estimate the fine-grained computation cost, we can specialize Equation 3.4 and have:

$$C_{fine_{comp}} = \sum_{N_{locGrids}} (R \times P_{elem} \times N_{ftdGrid})$$

$$= R \times P_{elem} \times F_{loc} \times \frac{N_{glb}}{N_{procs}} \quad (3.7)$$

**Hybrid Computation Cost:** Similar to the fine-grained partitioning strategy, the hybrid partitioning strategy can also lead to a reasonably well-balanced computation workload.
Therefore, we can infer the hybrid computation cost from Equation 3.7, but to capture possible small workload imbalance, we empirically add a penalty factor $p_{hybrid} - 0.06$:

$$C_{\text{comp}}^{hybrid} = (R \times P_{\text{elem}} \times F_{\text{glob}} \times \frac{N_{\text{glob}}}{N_{\text{procs}}}) \times (1 + p_{hybrid}) \quad (3.8)$$

**Auto-Grained Computation Cost**: Calculation of costs with the auto-grained partitioning strategy involves additional considerations. Particularly, an extra sampling cost should be added, and because the auto-grained partitioning strategy does not evenly partition the input data, we have to estimate both the loading cost and the computation cost altogether. Through extensive experiments, we observed a slight workload imbalance for the auto-grained partitioning. To formally capture this workload imbalance, we empirically add a penalty factor $p_{auto}$ to the total processing cost, where $p_{auto} = 0.06$:

$$C_{\text{total}}^{auto} = \sum_{N_{\text{procs}}} N_{\text{procs}} C_{\text{load}}^{auto} + \sum_{N_{\text{procs}}} N_{\text{procs}} C_{\text{comp}}^{auto} \times (1 + p_{auto}) + C_{\text{samp}} \quad (3.9)$$

where $C_{\text{samp}}$ refers to the sampling cost. Although the sampling cost can also be determined by the user with a given sampling probability, for simplicity, we consider it as a small constant, which is denoted as $c_{\text{samp}} = 2$.

Finally, the total computation cost in Equation 3.9 can be estimated as follows:

$$\sum_{N_{\text{procs}}} N_{\text{procs}} C_{\text{comp}}^{auto} = R \times P_{\text{elem}} \times F_{\text{glob}} \times N_{\text{glob}} \quad (3.10)$$

### 3.3 Overlapping Aggregation Algorithms

In this section, we present algorithms for overlapping aggregations, i.e., sliding, hierarchical, and circular aggregations.

Unlike non-overlapping aggregations, an overlapping aggregation can incur repeated loads and computations on the same data element. This leads to two important
considerations in the design of algorithms: 1) \textit{I/O Cost}: we will like to reuse the data loaded into memory to reduce repeated disk I/O operations, and 2) \textit{Memory Accesses}: we will like to avoid repeated accesses to the same element even in memory, since they can cause unnecessary cache misses.

Based on these two considerations, we have designed three approaches: a \textit{naive approach}, which, as the name suggests, is a simple scheme that ignores both I/O cost and memory access considerations, a \textit{data-reuse approach} that addresses I/O costs but does not reduce memory accesses, and an \textit{all-reuse approach}, which is conscious of both.

3.3.1 Naive Approach

To calculate each aggregate in an overlapping aggregation, one possibility is to simply load a grid of elements and then performs an aggregation over them, repeating the process for each aggregate to be computed. Thus, if there are $N$ aggregates to calculate in total, $N$ separate grids are loaded. To parallelize this operation, aggregations to be performed can be evenly distributed among each processor. Any of the partitioning strategies discussed in Section 3.2 can be used for this purpose. Obviously, the main drawback of this approach is that data resident in memory and/or cache is not reused, leading to redundant I/O and repeated memory accesses. Besides, significant workload imbalance may arise in hierarchical and circular aggregations due to different grid sizes.

3.3.2 Data-Reuse Approach

Unlike the naive approach, the data-reuse approach can reuse the data loaded into memory. We load a large array portion in each step, which can be either the entire array slab involved in the query, or its subset (e.g., the outermost grid in the hierarchical or circular aggregations). Aggregations that need this data can be processed without any
further disk I/O operations. Now, to parallelize the aggregation, the loaded array slab can be evenly distributed to all the processors. Certain aggregations will require data from multiple partitions (or even different array portions, which is the case even for the sequential version). These cases can be handled by performing a local aggregation followed by a global aggregation, as is needed in the case of non-overlapping aggregations.

Clearly, this approach can massively reduce disk I/O costs. However, other limitations of the naive approach still remain, i.e., it involves repeated memory accesses to the same element.

### 3.3.3 All-Reuse Approach

Similar to the data-reuse approach, the all-reuse approach also loads a large array slab at a time and reuses the loaded data to reduce disk I/O. However, the distinguishing characteristic of this method is that each element is accessed only once - even for multiple aggregations - leading to much fewer memory accesses. The main underlying idea is that it is more computationally efficient to iterate over elements and update the associated aggregates in the process, rather than iterating over aggregation results that need to be calculated and then accessing the required elements for producing these results.

Algorithm 3 shows sliding aggregation by using the all-reuse approach. This method can only be applied if the aggregation operator is algebraic (also referred to as associative and commutative). The array slab is evenly partitioned in the highest dimension to ensure data contiguity. After a partition is loaded, local aggregates corresponding to all sliding grids that overlap with this partition are initialized. Once an element is read, it is used to update all the aggregates it contributes to. In this way, when each element is loaded into
cache, it will be fully reused before being flushed out, leading to fewer cache misses.

Because some of sliding grids overlap with two neighboring partitions, corresponding partial aggregation results need to be merged in a global aggregation phase.

**Algorithm 3: SlidingAgg**(slab \( S \), num_procs \( n \))

1: Evenly divide \( S \) into \( n \) partitions (set \( P \))
2: Each processor loads its partition \( P_i (i = 1, \ldots, n) \)
3: Initialize \( m \) local aggregates \( A \) for the \( m \) sliding grids that \( P_i \) overlaps with
4: for each element \( e \) in \( P_i \) do
5: for each aggregate \( A_j (j = 1, \ldots, m) \) associated with \( e \) do
6: \( A_j \leftarrow aggregate(e, A_j) \)
7: end for
8: end for
9: Perform global aggregation
10: return \( A \)

The similar method can be applied for hierarchical aggregation. The main differences are, local aggregates are initialized for all concentric grids, rather than being initialized for all sliding grids that a partition overlaps with, and the global aggregation involves merging all local aggregates instead of the only a few ones that overlap with two neighboring partitions. Further, circular aggregation is a special case, since each element will contribute to exactly one aggregate that corresponds to a disjoint circle. Therefore, there is no need for an inner loop in Algorithm 3 to iterate over different aggregates.

### 3.4 Optimization for Chunked Array Storage

Our goal in this work has been to support an approach of directly using array storage as a DB, where no data ingestion costs are involved, and instead, operations are supported over native storage. The default array storage layout on disks is the same as the in-memory layout, which is a contiguous block stored in the row-major or the
column-major fashion. It turns out that many libraries (e.g., NetCDF and HDF5) that manage array storage on disks can also support a more optimized layout, which is called the chunked storage \[183\]. The chunked storage involves splitting an array into chunks based on a multi-dimensional partitioning. As a result, array elements are not stored contiguously along any of the dimensions, and hence dimension dependency can be alleviated. Libraries that support such data storage also have the feature that when a single element is loaded to the memory, all the other elements in the same chunk are also cached.

Because the grid aggregation methods discussed in Section 3.2 do not consider the chunked array storage, the algorithms can be quite inefficient when chunked storage is used. For example, if a chunk is shared by multiple logical grids, this chunk may be loaded repeatedly to perform aggregations on different grids. Similarly, if a chunk is shared by different partitions, this chunk is loaded redundantly by multiple processors. Avoiding such overheads is the motivation for our method (Algorithm 4).

**Algorithm 4: ChunkBasedGridAgg\((slab \ S, num\_procs \ n)\)**

1: Identify chunks comprising the queried array slab \( S \)
2: Evenly divide chunks into \( n \) contiguous chunk sets (set \( C \))
3: Set a load buffer \( B \) \{ buffer size is a multiple of chunk size and less than cache size \}
4: Each processor divides \( C_i (i = 1, \ldots, n) \) into subsets of chunks of size at most \( B \)
5: for each subset of size at most \( B \) do
6: Load chunk(s) into \( B \)
7: for each chunk \( c \) in \( B \) do
8: for each element \( e \) in \( c \) do
9: for each aggregate \( A_j (j = 1, \ldots, m) \) associated with \( e \) do
10: \( A_j \leftarrow \text{aggregate}(e, A_j) \)
11: end for
12: end for
13: end for
14: end for
15: return \( A \)
First, to minimize the number of chunks that have to be loaded by more than one processors, instead of mapping logical grids to processors, we map physical chunks to processors. Second, to favor I/O performance, the partitioning is performed in such a way that the chunks within one partition are contiguous to the extent possible. To fully reuse the cached data in the same chunk, once a chunk of data is read, all elements in it are used to update the aggregates they contribute to. Note that the size of a load buffer needs to be carefully tuned, i.e., it should be a multiple of the chunk size and less than the cache size. If both the chunk size and the buffer size are very small, extra overheads can be caused by frequent small I/O requests. If the buffer size is too large compared with the cache size, the first a few cached chunks will be flushed out before they are actually used in aggregation. Since the actual number of chunks in the load buffer is usually less than the number of processors used, the fine-grained partitioning strategy is not supported. Otherwise, one chunk will be redundantly loaded by multiple processors.

The similar chunk-based optimization can also be applied for overlapping aggregations, since the all-reuse approach does not require data to be loaded in any specific manner.

3.5 Experimental Results

In this section, we evaluate the performance of our approach and implementations for various structural aggregations, using two real datasets, as well as synthetic datasets created to vary the amount of skew. Our experiments were conducted on a cluster of nodes with a shared file system (for evaluating parallel performance) and on the Amazon EC2 platform (for comparing with SciDB). We designed the experiments with the following goals: 1) to demonstrate the performance of our proposed partitioning strategies for grid aggregation (especially their ability to handle skewed data) and the effectiveness of the
cost models for choosing the best scheme, 2) to compare performance of our system with SciDB, a popular Array DBMS, and 3) to evaluate the performance of parallel algorithms for different structural aggregations we have introduced.

Figure 3.3: Processing Times (left) and Predicted Processing Times (right) on Synthetic Datasets of Varying Skew

Figure 3.4: Processing Times (left) and Predicted Processing Times (right) on Cloud Pressure Dataset

3.5.1 Experimental Setup

Our experiments were conducted using two real large-scale array-based datasets. These are both two-dimensional satellite imagery datasets, which use the HDF5 format
that is popular in many scientific areas. These two datasets are referred to as cloud pressure and terrain pressure, since they record effective cloud pressure and terrain pressure, respectively. Each one comprises $1,000K \times 1K$ double-precision elements, and has a size of 8 GB. They are downloaded from the Land Parameter Retrieval Model (LPRM) Level 2 (swath) collection [6]. Additionally, in Section 3.5.2 we also used 4-GB synthetic datasets of varying skew, each of which comprises $512K \times 1K$ double-precision elements. Since we focused on evaluating the performance of our system on native array storage, the arrays are stored in the default row-major order on disks, i.e., the array storage was not preprocessed or reorganized. An exception is the set of experiments with the chunk-based grid aggregation. We used traditional aggregates including COUNT, SUM, AVG, as well as MIN and MAX in our experiments, Specifically, a compute-intensive user-defined aggregation function was used in Section 3.5.2 and 3.5.4.

The performance comparison between our system and SciDB was conducted on an Amazon EC2 instance with 32 GB of main memory and 8-core Intel(R) Xeon(R) E5-2670 CPU, and the clock frequency of each core was 2.6 GHz. The version of SciDB was 13.12. All the other experiments were conducted on a cluster of machines, where each node had
an AMD Opteron(TM) Processor 8218 with 4 dual-core CPUs (8 cores in all). The clock frequency of each core was 2.6 GHz, and the system had a 16 GB main memory. We used up to 16 compute nodes for our study. HDF5 version 1.8.10 was used for all our experiments.

3.5.2 Handling Skewed Data and Prediction Accuracy of Partitioning Strategy Decider

Our first experiment evaluates the performance of grid aggregation methods over skewed data and the prediction accuracy of the partitioning strategy decider. Recall that the relative performance of different partitioning strategies and our cost models are primarily impacted by two factors: the amount of computation involved in the aggregation and the level of data skew. Thus, we designed experiments to vary both, and compared the performance of different partitioning strategies and the prediction accuracy of our models. Note that all the processors we used in this set of experiments were on the different machines.

We varied skew as follows. We designed synthetic datasets focusing on execution on 2 processors, where we applied a value-based predicate to filter out 50% of the data elements in each dataset. We created four synthetic datasets with different distributions of the data after filtering over the two partitions: 1) very low skew: 24% in the first half, and 26% in the second half; 2) low skew: 20% in the first half, and 30% in the second half; 3) high skew: 10% in the first half, and 40% in the second half; and 4) extremely high skew: 0% in the first half (empty), and 50% in the second half (full).

Also, the amount of computation in aggregation was varied in the following fashion. First, note that while calculating traditional aggregates like COUNT and AVG, the dominant cost is I/O, as the amount of computation is very small. Because I/O costs can
be trivially balanced across processors with no impact from skew, we have used a user-defined and compute-intensive aggregation function - where data within each grid was clustered using the popular k-means algorithm. The amount of computation per element was further varied by using different number of iterations (before convergence) in the k-means algorithm.

The first set of experiments were with synthetic datasets of varying skew. The grid size was varied from 2,000K to 6,000K elements, and we set 30 centroids and 4 iterations for each grid. Figures 3.3(a) and 3.3(b) show the actual processing times and the predicted processing times from our models, respectively. With very low skew, the best results were obtained from the coarse-grained scheme. As the data skew increased, the performance of the coarse-grained partitioning began to get worse, and the best results were obtained from fine-grained and auto-grained schemes. As also predicted by our cost model, the processing costs with fine-grained, hybrid, and auto-grained schemes do not vary with skew.

Second, we experimented on both the cloud pressure and the terrain pressure datasets with 4 processors. The number of iterations for clustering was varied from 2 to 12 for the cloud pressure, and from 2 to 22 for the terrain pressure, with increasing the processing time per element. The grid size was varied from 2,000K to 6,000K elements, and we set 20 centroids for each grid. For the cloud pressure dataset, we filtered out all the missing values. This introduced significant skew - specifically, among the 4 partitions obtained using the coarse-grained partitioning, the proportions of data after filtering were 55%, 100%, 100%, and 56%, respectively. For the terrain pressure dataset, we filtered out all the values less than a threshold, and the fraction of the data in all the 4 partitions were 65%, 43%, 40%, and 62%, respectively. Figures 3.4(a) and 3.4(b) show the results on the cloud pressure dataset, and Figures 3.5(a) and 3.5(b) show the results on the terrain pressure dataset. The
observations from these charts are as follows. When there are only 2 iterations on both datasets, the I/O cost is the dominating cost. As a result, the coarse-grained partitioning gives the best performance, because it minimizes disk seek times and hence has the best I/O performance. When more iterations are involved, the computation cost begins to dominate the total processing cost, and skew potentially impacts workload balance. Therefore, both the fine-grained and auto-grained schemes result in better performance, as they are able to load balance even in the presence of skew. Additionally, we can also see that when the number of iterations is 10 in Figure 3.4(a) and 22 in Figure 3.5(b), the hybrid partitioning can outperform the coarse-grained partitioning, because the former can handle skew better. Lastly, when neither I/O cost nor computation cost overwhelmingly outweighs the other, the auto-grained partitioning will have the best performance. This is because it has a load balancing advantage over the coarse-grained partitioning strategy, and it has better I/O performance than the fine-grained and hybrid partitioning strategies.

We can also see that the predicted cost follows actual processing times, and in almost all cases, it can predict the relative performance of different partitioning strategies accurately. The only case where the model is not correct in predicting performance is when the performances of two strategies are very close. This is because it is a simple model, which has not considered factors like communication cost (this cost is trivial only with an exception for fine-grained partitioning with a very small grid size). Overall, we have shown that the model can help choose the appropriate partitioning strategy for a given query, in view of the dataset skew and amount of computation involved per element in the aggregation.

Additionally, two issues from this experiment need to be clarified. First, to demonstrate the predication accuracy, we have focused on the “crossover” of the
performance of different partitioning schemes. These results may give the appearance that the best scheme here might only marginally outperform the others. However, if we were to further increase the amount of computation, it turns out that the skew-tolerant partitioning schemes (fine-grained and auto-grained schemes) could outperform the coarse-grained scheme by more than a factor of 2. Second, for handling data skew, the fine-grained partitioning might appear to be sufficient, which may seem to imply that the other schemes are unnecessary. However, two disadvantages of the fine-grained scheme are not apparent from our experiments: 1) this scheme can incur significant communication costs when the queries involve a small grid size, and 2) as mentioned in Section 3.4, this scheme does not work for chunked array storage.

3.5.3 Performance Comparison with SciDB

Our next experiment compared the performance of our system with SciDB, a popular Array DBMS that could support many of the operations we support. We performed comparison on an EC2 instance with 8 cores. As we have discussed throughout, SciDB involves a considerable upfront cost of data transformation and reloading. Our figures only report SciDB’s query processing times after all the data was loaded, i.e., excluding the data ingestion time, since our goal is to show that our approach of using array storage as a DB can still allow efficient structural aggregations over array data. It should be noted that the data ingestion time for the dataset we have used could easily take 100x the time for executing a simple query over the same amount of data - this included the time for the transforming HDF5 datasets into CSV files and casting the temporary 1D arrays within SciDB into 2D arrays.
Because SciDB does not support hierarchical or circular aggregation, we only compared our performance of grid and sliding aggregations (referred to as window aggregations in SciDB). We used the 8-GB cloud pressure dataset, and we evaluated the performance of our system on both the default flat array storage and the chunked storage (with chunk size set as \(1K \times 512\), also equal to the chunk size used when the dataset was loaded into SciDB). Note that this chunk size (4 MB) matches SciDB recommended chunk sizes (4 - 8 MB)\(^5\).

To thoroughly evaluate the performance differences between the two systems, we created queries using different aggregation operators and varied different parameters like the grid size and the size of the array slab. Specifically, the grid size was varied from 32K to 256K elements for grid aggregation and from \(3 \times 3\) to \(7 \times 7\) for sliding aggregation. We also aggregated over array slabs of varying sizes, with ratio between the queried array slab size and the dataset size varied from 12.5\% to 100\%. With our system, the coarse-grained partitioning was used for grid aggregations, and the all-reuse approach was used for sliding aggregations, since they gave the best performance on this dataset and the queries we used.

**Performance Comparison of Grid Aggregation:** The comparison results of grid aggregations are shown in Figure 3.6. Since we did not observe any measurable difference in relative performance across different aggregation operators or grid sizes, we report only a single (average) processing time with each approach, and each ratio between array slab size and dataset size. We can see that our system had better performance irrespective of whether the array storage is flat or chunked, or whether the queried array slab is a subarray or the entire array. As shown in Figure 3.6(a), the sequential performance of our system is better by an average of 39\% and 25\% on flat arrays and chunked arrays, respectively. To

\(^5\text{http://www.scidb.org/forum/viewtopic.php?f=11&t=334}\)
the best of our understanding, this difference arises because our system loads grids or
chunks in order, leading to better I/O performance, whereas SciDB loads and processes
chunks in random order and does not preserve sequential I/O.

Moreover, as shown in Figure 3.6(b), the parallel performance of our system is also
superior to SciDB. We believe SciDB scales less well mainly due to the nontrivial
overhead of process scheduling. Note that although our sequential performance on
chunked array is worse than the performance on flat array, due to the overhead of
addressing the mismatch between the logical grid layout and the physical chunk layout,
processing queries on chunked array is likely to provide better scalability especially for a
data subset.
Performance Comparison of Sliding Aggregation: Some additional background is important before we elaborate on the performance difference for sliding aggregations. SciDB potentially simplifies sliding aggregations by replicating or overlapping chunk boundary elements while loading data [143]. The number of elements to be replicated across neighboring chunks should be set by the user at the time when the data is loaded into SciDB. Assuming sufficient number of boundary elements have been overlapped, sliding aggregation results can be obtained by processing data from each chunk independently, or what we will refer to as aggregation over overlapping chunks. If such chunk overlap has not been set while loading data, or if sliding aggregation with a grid size greater than the chunk overlap is performed, SciDB has to stitch neighboring chunks together when processing chunk boundary elements (aggregation over non-overlapping...
chunks), and this results in considerable performance loss. Because all possible queries and proper chunk overlap size may not be known before the data is loaded into databases, both aggregations over overlapping chunks and non-overlapping chunks are likely, and we have evaluated both of them. Note that overlapping chunked array storage is not supported by popular libraries for scientific datasets, like NetCDF or HDF5.

As we can see in Figure 3.7, our system outperforms SciDB with sliding aggregation over non-overlapping chunks (average of 83% for sequential performance and 88% for parallel performance), and even outperforms SciDB over overlapping chunks by an average of 79% for both sequential and parallel performance. To the best of our understanding, such performance difference is because of our all-reuse approach, which can provide completely sequential I/O and significantly reduce cache misses.

3.5.4 Performance of Parallel Structural Aggregations

Parallel Performance of Grid Aggregations: To evaluate our algorithm, we considered all four partitioning strategies and used the terrain pressure dataset, with both I/O bound and CPU-bound grid aggregations. The grid size was varied from 8K to 256K elements in the I/O bound aggregation that calculates traditional aggregates, and it was varied from 1,000K to 8,000K elements in the CPU-bound aggregation (k-means clustering, with 20 centroids and 20 iterations for each grid). The number of nodes we used varied from 1 to 16. Note that since there is no workload imbalance in the I/O bound aggregation, the auto-grained partitioning strategy becomes equivalent to coarse-grained one and is not shown here.

Figures 3.8(a) and 3.8(b) show the results of I/O bound and CPU bound parallel grid aggregations, respectively. Not surprisingly, for I/O bound parallel grid aggregations, the
coarse-grained partitioning results in the best performance, as it preserves better data contiguity. Both coarse-grained and hybrid schemes have better scalability than the fine-grained scheme, because of the lower communication costs for each. For CPU bound parallel grid aggregations, as the number of nodes grows, both coarse-grained and auto-grained schemes have better scalability because the grids are contiguous in all partitions. Again, not surprisingly, auto-grained scheme which explicitly accounts for skew has the best performance with 16 nodes. Overall, we can see that the best scheme has achieved at least 75% parallel efficiency.

**Chunk-Based Optimization Evaluation:** To evaluate the optimization for chunked array storage, we compared the performance of (I/O bound) grid aggregations with chunk-based optimization against the ones without such optimization. As mentioned earlier in
Figure 3.9: Parallel Performance of Grid Aggregations with and without Chunk-Based Optimization

Section 3.4, to avoid redundant chunk read by multiple processors, the fine-grained partitioning is not supported over chunked arrays. Thus, here we only compared the chunk-based grid aggregations implemented by coarse-grained and hybrid partitioning, with the original grid aggregation implemented by the coarse-grained partitioning, which has proved to have the best I/O performance on flat arrays. We transformed both two real datasets to use chunked storage. We used the same I/O bound grid aggregation parameters on both two real datasets in Section 3.5.4 with up to 16 nodes. The chunk size was $64K \times 1$.

As Figure 3.9 shows, the chunk-based grid aggregations implemented by both coarse-grained and hybrid partitioning can outperform the original implementation with the coarse-grained partitioning, by a factor of up to 2.75 and 2.30, respectively. This is because that, the original implementation allows different subsets of a data chunk to be read by different processors, and hence the same chunk is very likely to be loaded by multiple processors redundantly. In contrast, with our chunk-based optimization, each
Figure 3.10: Parallel Performance of Overlapping Aggregations
array chunk is loaded by exactly one processor, leading to better I/O performance. Note that the results here are not in contrast to the results in Figure 3.6. This is because that, although we can improve the grid aggregation performance over the chunked storage with our optimized algorithm, it is still worse than the performance over the flat storage, due to the extra overheads of handling the mismatch between the logical grid layout and the physical chunk layout.

**Parallel Performance of Sliding Aggregation:** We experimented on the two real datasets to evaluate performance of sliding aggregation algorithms, i.e., the naive approach, the data-reuse approach, and the all-reuse approach. The number of nodes we used was varied from 1 to 16. The sliding grid size was varied from $3 \times 3$ to $7 \times 7$. We used the coarse-grained partitioning strategy for the naive approach.

Figure 3.10(a) shows the results. We can see that the aggregations implemented by the naive approach is dramatically slower than the other two approaches, as we will expect. The all-reuse approach can outperform the data-reuse approach by a factor of 37.5%, 33.9%, 26.1%, 21.5% and 17.7%, for 1, 2, 4, 8 and 16 nodes, respectively, because it is more cache-friendly.

**Parallel Performance of Hierarchical and Circular Aggregations:** The last experiment evaluates the parallel performance of hierarchical and circular aggregations, implemented by the naive approach, the data-reuse approach, and the all-reuse approach. We experimented on the two real datasets. Up to 16 nodes were used for these two aggregations. The size of innermost grid was $512K \times 512$, and the grid expanded by 32K in the first dimension and 32 in the second dimension in each step, with a total of 16 concentric grids in all.
Figures 3.10(b) and 3.10(c) show the parallel performance of hierarchical and circular aggregations, respectively. Again, we can see that the all-reuse approach leads to the best performance, for the same reasons we had discussed earlier. However, unlike the previous results, we can see that the naive approach can even outperform the data-reuse approach at the beginning. This is because, unlike sliding aggregation, with hierarchical and circular aggregations, there are not too many frequent and small I/O requests even in the naive approach. Further, for the naive approach, the data loaded for each grid is always contiguous in memory, and the data will be sequentially accessed by cache during aggregation.

### 3.6 Related Work

We now compare our work with existing efforts on array database systems, array query languages, work on parallel aggregations in databases, and closely related techniques for handling data skew.

SciDB [39] is a popular Array DBMS which has drawn considerable attention recently. We have conducted extensive performance comparison between our system and SciDB, and demonstrated that we can process structural aggregations faster, in addition to alleviating the need for expensive data ingestion steps. RasDaMan [32] is another robust system, which is based on an array algebraic framework [31]. ArrayDB [152] is a prototype array database system, which is mainly used for processing small 2D images. MonetDB [217] is a column-store DBMS for spatial applications. RasDaMan, MonetDB, and SciDB all support certain structural aggregations similar to our system.

Apart from a variety of query languages designed for relational data, tree data, and graph data [38, 190, 216], query languages or operators for array operations have been
studied by the above efforts, as well as a number of other projects. RasDaMan uses RasQL [32], SciDB [39] supports both an SQL-like query language AQL (Array Query Language) and a functional language AFL (Array Functional Language), whereas MonetDB initially used RAM [214], and now uses SciQL [258]. In other efforts, the open source Postgres database [10] contains an array data type, with a fairly extensive language binding to SQL. AQL [140] involves a nested relational calculus to manipulate arrays. AML [151] is a generic array query language, which takes bit patterns as parameters for all the array operations. AQuery [138] makes use of ordered relational tables to process natural order-dependent queries, specifically over 1D arrays. Others projects have also made similar proposals [44, 61, 159, 177].

Parallelization of structural aggregations has already been carried out in context of RasDaMan [32] and SciDB [39]. Prior to that, Shatdal and Naughton developed parallel algorithms for value-based aggregations in relational databases [186]. Our all-reuse approach for overlapping aggregations has some similarities to the aggregations at multiple granularity on data cube [92], because such multi-level aggregation also involves overlaps among aggregated regions. However, the difference is that our method accumulates associated aggregation results concurrently, while OLAP multi-level aggregation executes in a bottom-up fashion.

Load imbalance caused by data skew has been recognized as an important issue in supporting array operations. Since prior systems involved a data ingestion phase, they addressed the problem by reorganizing data in a way that impact of skew was minimized [180, 191, 196]. Particularly, the techniques included storing the array in irregular chunks of unequal sizes but with the same amount of data, or shuffling the chunks to randomly distribute them across processors. Data skew problem has recently
been addressed in the context of MapReduce also. As also summarized by Kwon et al. [126], static optimizers [100, 124] leverage cost models to balance the expected processing cost, and dynamic optimizers [24, 125, 218] detect and repartitions the bottleneck tasks at runtime. Other efforts [25, 253] optimize MapReduce speculative execution.

Our work takes inspiration from the NoDB approach [22] and related work on automatic data virtualization [238]. Our contribution, however, is in building a database engine on top of the native array storage rather than the flat files in CSV format, which essentially present a relational table view. We have implemented our system on top of earlier efforts that were also based on the native array storage, but supported only the selection queries [200, 206, 234]. Other projects with a similar focus include the Oracle Database Filesystem (DBFS) [123], which can extend certain database features to unstructured data storage. PostgresRaw [22] supports advanced query processing over raw files on the fly, assuming they can be mapped to relational tables. MonetDB data vault [102] also intends to support transparent access to the data from an external scientific file repository. FastBit [241] and FastQuery [77, 117] accelerate the query processing over array storage using bitmap indices. By contrast, our system can support additional types of aggregations as well as optimized aggregations over skewed or chunked data.

3.7 Summary

This work has focused on providing the database-like support over native array storage, with a specific focus on implementing a number of structural aggregation operations. These operations arise in a number of scientific disciplines. We have demonstrated that by development of nuanced algorithms and cost models, efficient array
operations can be supported over native array storage. Detailed performance comparisons with SciDB further confirm this - despite no data ingestion overhead, our aggregation processing costs are lower. We have also shown high parallel efficiency with our algorithms.
Chapter 4: A Novel Approach for Approximate Aggregations Over Arrays

To tame big data, approximate query processing (AQL) often serves as a cost-effective technique, by sacrificing some level of accuracy to meet stringent response-time requirements. In this chapter, we utilize novel bitmap indexing to effectively perform flexible approximate aggregations that are frequently used in scientific data management systems. Compared with the other approximate aggregation methods, our method is specifically designed for processing fast and accurate aggregations on array data.

4.1 Challenges and Limitations

4.1.1 Challenges in Approximate Aggregations over Array Data

Extending the scope of approximate aggregation techniques to the domain of array data analysis, however, imposes certain unique challenges, which are summarized as follows.

Flexible Aggregation over Any Subset: Clearly, for both relational and array data, approximate aggregations may be needed over any subset of data. However, what is distinct about array data is that subsets may be constructed based on dimensions (or array index), in addition to subsets involving value-based predicates based on array element. Broadly, we can divide all the subsetting predicates involved in querying array data into three categories. Using a 3-dimensional dataset whose dimensions include *latitude*,
longitude, and depth, and attributes of array element include temperature and pressure, the three types of subsetting predicates are: 1) dimension-based predicate, which is specified by dimension indices (or coordinate system) and related to the array physical layout, e.g., \( \text{latitude} > 3 \) or \( \text{depth} < 5 \); 2) value-based predicate, which is described by a specific value range of array element, e.g., \( \text{temperature} < 4 \) or \( \text{pressure} > 7 \); and 3) combined predicate, which combines the above two types of predicates. Techniques developed in the context of relational databases can be trivially extended to array data by considering each array dimension as an attribute. However, as we will demonstrate experimentally later, this can be extremely expensive.

**Aggregation Accuracy:** Approximate aggregation methods often operate on a small amount of summary structure or data synopsis instead of the entire original dataset. Accordingly, two factors are extremely important while creating summary structures of array data so as to maintain accuracy. The first is value distribution, i.e., the value distribution of summary structure should be as close to the original dataset as possible. Preserving this factor in the summary structure is particularly critical while analyzing skewed data. The second is spatial distribution, i.e., the aggregation accuracy should be maintained not only for the entire dataset but also for various spatial sub-blocks. To the best of our knowledge, none of the existing approaches is able to preserve both distributions efficiently.

**Aggregation without Data Reorganization:** Although certain existing methods can allow summaries that preserve both value and spatial distributions, they require data reorganization before any query can be processed. For example, aggregation can be approximated based on KD-tree based stratified sampling [239]. However, before such sampling can be performed, costly data reorganization is required, leading to massive
memory and disk I/O costs. In addition, because data in the original layout may also be
required for other reasons, such data reorganization can lead to the need of maintaining
multiple copies of the original dataset. This implies at least a factor of two increase in the
storage costs. In summary, we need approximate aggregation methods that can avoid data
reorganization and maintain the data in the original layout.

4.1.2 Existing Approximate Aggregation Techniques and Limitations

As mentioned earlier, approximate aggregation methods operate on the summary
structures that are much more compact than the original dataset. Existing techniques to
obtain such summary structures include those based on sampling [29, 46, 171, 222],
histograms [87, 88, 90, 101, 162, 164, 175] and wavelets [104, 198].

**Sampling-Based Techniques:** Most of the sampling methods [171, 222] developed in the
context of scientific data management focus on the spatial distribution, but ignore value
distribution. On the other hand, value distribution based sampling [29, 46] has been well
studied and proven to be effective for handling possibly skewed datasets. These methods,
however, are not developed for array data, and do not even consider spatial distribution.
To the best of our knowledge, so far only random sampling has been employed by modern
array databases for approximate query processing, e.g., SciDB [39] supports Bernoulli
sampling for approximate aggregations.

**Histogram-Based Techniques:** Histograms have been extensively studied to serve as a
tool for approximate aggregations [87, 88, 101, 175]. The main advantage of a
(1-dimensional) histogram over other techniques is that it occupies very small space and
incurs only a small runtime overhead. However, histogram is unable to capture spatial
distribution of the original dataset. As a result, histogram-based techniques can only be
applied to relational data, where no spatial distribution information is needed (e.g., the user does not care whether an aggregated tuple is the first or the last tuple in a relational view). Now, one method for handling array data can be treating all dimension-based attributes as additional relational attributes, and constructing multi-dimensional histograms [90, 162, 164] for approximation. However, these histograms either incur a very high space cost, or the partitioning granularity needs to increase exponentially as array dimensionality grows. This, in turn, leads to substantial runtime overheads or high inaccuracy [90]. To the best of our knowledge, multi-dimensional histograms have been primarily applied to selectivity estimation [135, 176], which is an internal step in query optimization, and not for answering aggregate queries from users.

Wavelet-Based Techniques: Wavelets, which were proposed in the context of signal and image processing [104, 198], have been shown to be an effective mathematical tool for processing approximate range-sum queries over OLAP data cubes [45, 153, 220, 221]. By applying hierarchical decomposition functions, raw data can be transformed into wavelet coefficients. However, if these techniques are applied over array data, by mapping an OLAP data cube to an array, wavelets can only process dimension-based and not the value-based predicates. Alternatively, a value-based attribute can be added as an additional dimension to the data cube, but then data reorganization (e.g., sorting) is required. In addition, the aggregate operators supported by this method are restricted to SUM, COUNT, and AVERAGE – other desirable operators, including user-defined aggregation functions, cannot be supported.
4.2 Approximate Aggregations Using Bitmap Indices

This section first provides background information on bitmap indexing and binning strategies, then describes our bitmap-based approximate aggregation method.

4.2.1 Background: Bitmap Indexing and Binning Strategies

Indexing provides an efficient way to support value-based queries and has been extensively studied 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 for querying static (i.e., read-only or append-only) data. It was initially used in data warehouses [169, 247, 248], and more recently, has been used in scientific data management [194, 195, 235, 242]. Particularly, recent work has shown that bitmap indexing can help support efficient querying of scientific datasets stored in native formats [56, 77, 201, 206].

![Figure 4.1: An Example of Bitmap Indexing](image-url)

<table>
<thead>
<tr>
<th>ID</th>
<th>Value</th>
<th>e0</th>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
<th>e5</th>
<th>i0</th>
<th>i1</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Dataset: Low Level Indices

High Level Indices

Figure 4.1: An Example of Bitmap Indexing
Figure 4.1 shows an example of bitmap indexing. In this example, the dataset contains 8 elements with 6 distinct values. The low-level bitmap indices contain 6 bitvectors, where each bitvector corresponds to each distinct value. The number of bits within each bitvector is equal to the total number of elements in the dataset. In each bitvector, a bit is set to 1 if the value for the corresponding data element value 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 a certain binning strategy, and in this example, 3 high-level bitvectors are built.

This simple example only contains integer values. Bitmap indexing also has been shown to be an efficient method for floating-point values [245]. For such datasets, instead of building a bitvector for each distinct value (even in the case of low-level bitvectors), we first group a set of values together by a certain binning strategy, and build bitvectors for these bins. This way, the total number of bitvectors can be kept at a manageable level.

Even after binning, 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 potentially 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) [26] and Word-Aligned Hybrid (WAH) [243, 244] 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 store the number of continuous 0s or 1s. Another property of the run-length compression methods is that, they can support fast bitwise operations without decompressing the data.
As we stated above, binning is a critical part of using bitmaps. There are two conventional binning strategies for bitmap indexing, which are *equi-width* and *equi-depth* [246]. Equi-width binning divides the entire dataset value domain into equal intervals. This binning strategy is straightforward to implement and has been shown to perform well in some cases. Equi-depth binning involves creating partitions in such a way that each bin contains approximately an equal number of elements.

Bitmap indexing was initially designed for accelerating *selection* operation, where value-based predicate(s) are applied. When binning is used, the elements within the *non-edge* bins are fully selected, whereas the elements within the *edge bin(s)* need to be further evaluated. Specifically, an extra step is required to check whether certain candidate elements within the edge bin(s) can satisfy the specified predicate, which is referred to as the *candidate check* [182]. When the data is highly skewed, the candidate check for equi-width binning can be, in the worst case, as expensive as a sequential scan of the entire dataset. In contrast, equi-depth binning can ensure that the candidate check costs are equal for all query ranges. Thus, this method improves the worst-case response time.

Our goal is to use bitmap indexing for approximate aggregations, and we can expect choice of the binning strategy to impact accuracy and/or performance. However, since so far bitmap indexing has not been used for approximate aggregation, none of the existing binning strategies has considered aggregation accuracy.

### 4.2.2 Bitmaps and Approximate Aggregation

Bitmap indices can be viewed as a summary (potentially approximate) representation of the entire data, similar to those used for approximate query processing. However, bitmap indexing has so far been used almost exclusively for accelerating processing
selection queries [56, 77, 201, 206, 242], and not for approximate aggregation. One can attribute this to two possible reasons. First, bitmap indexing was initially used in relational databases where there is no concept of dimension-based predicate. For such cases, bitmap indices cannot support approximate aggregation any better than histograms, since each bin can be mapped to a bucket in a histogram. Second, existing binning strategies do not consider aggregation accuracy at all. In contrast, histograms can provide better query efficiency because they have a more compact data structure, and also higher accuracy because of the various nuanced construction methods that have been developed over time [87, 88, 101, 175].

<table>
<thead>
<tr>
<th>Statistics \ Bin #</th>
<th>( i_0 )</th>
<th>( i_1 )</th>
<th>( i_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>SUM</td>
<td>4</td>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>SQUARED SUM</td>
<td>6</td>
<td>25</td>
<td>86</td>
</tr>
<tr>
<td>MIN</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>MAX</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

User-Defined Pre-Aggregation Statistics

Figure 4.2: An Example of Pre-Aggregation Statistics as Applied to the Example in Figure 4.1

Since our focus is on array structure which includes both dimensions and values, bitmap indices’ ability to preserve dimensional information is now a clear advantage. The second issue, approximation accuracy, remains, but will be addressed in Section 4.3.

4.2.3 Approximate Aggregation Method

We now focus on how bitmaps can be used for aggregation processing over array data. To facilitate approximate aggregations, it turns out that some additional information needs
to be stored in conjunction with bitvectors. We refer to this information as *pre-aggregation statistics*. Figure 4.2 shows the pre-aggregation statistics corresponding to the example in Figure 4.1. The information stored here includes COUNT, SUM, SQUARED SUM, as well as MIN and MAX (i.e., the two boundary values of a bin). The use of this information for approximate aggregations will be shown later in this section. However, one way of looking at this information is the following. While the spatial distribution is preserved by the bitvectors in the index file, the value distribution is preserved by these pre-aggregation statistics. Note that the statistics can also be customized by the user to support user-defined aggregation functions, which will also be discussed later.

---

**Algorithm 5:** \( \text{Agg}(\text{bitmap } B, \text{statistics } S, \text{aggOperator } Op, \text{valBasedPredicate } P_v, \text{dimBasedPredicate } P_d) \)

1. Let \( B' \) be the subset of \( B \) after value-based filtering
2. if \( P_v \neq \emptyset \) then
3. \( B' \leftarrow \text{SELECT}(B, P_v) \) \{value-based filtering\}
4. else
5. \( B' \leftarrow B \)
6. end if
7. Let the lower bound bitvector and the upper bound bitvector in \( B' \) be \( b_i \) and \( b_j \), respectively
8. if \( P_d \neq \emptyset \) then
9. \( \text{Translate } P_d \text{ into a predicate bitvector } b_d \)
10. for each bitvector \( b_k \in B' \) do
11. \( b'_k \leftarrow b_k \land b_d \) \{dimension-based filtering\}
12. \( \text{CntArr}[k] \leftarrow \text{COUNT}(b'_k) \)
13. end for
14. end if
15. return \( \text{COMP}(\text{CntArr}, i, j, S, Op) \) \{see Table 4.1\}

The bitmap-based approximate aggregation method is formally shown as Algorithm 5. Given an aggregate query, all the predicates can be categorized into value-based and dimension-based predicates, denoted as \( P_v \) and \( P_d \), respectively. The aggregate operator is
denoted as Op. Additionally, for the queried dataset, both bitmap indices B and the corresponding pre-aggregation statistics S are assumed to be available. As Algorithm 5 shows, the approximate aggregation method takes five steps. Let us now use the example in Figure 4.1, further assume the given value-based and dimension-based predicates are Value > 3 and ID < 4, respectively, and follow the steps of the algorithm.

The first step is value-based filtering, which selects only a subset of the original bitmap indices by applying $P_v$. Applying this on our example, only two bins $i_1$ and $i_2$ will be selected, since they overlap with the value range Value > 3.

In the second step, $P_d$ is translated into a predicate bitvector $b_d$, in which each 1 indicates an array element that satisfies the predicate. Here the predicate bitvector is 11110000, and it indicates that only the first 4 elements are within the dimensional range ID < 4. Note that any dimension-based predicate over a multi-dimensional array can still be translated into a single 1-dimensional predicate bitvector. Such translation can, however, appear more expensive because it creates several distinct segments of contiguous 1s. It turns out that this step only takes a small fraction of total query processing time (typically less than 10%), and does not negatively impact the overall performance.

The third step is dimension-based filtering. In this step, a bitwise AND operation is performed over each bitvector in the bitmap subset and $b_d$, such that all the 1s in the output bitvectors indicate the elements that satisfy both $P_v$ and $P_d$. In this example, the two output bitvectors are 01000000 and 10010000, respectively.

The fourth step is to generate a count array $CntArr$ by counting the number of 1s in each output bitvector. This operation can also be supported by compressed bitmap formats, i.e., no decompression will be needed. Here, the two corresponding elements in the count array are 1 and 2, respectively.
Table 4.1: Definitions of the Aggregation Interface for Different Aggregate Operators

<table>
<thead>
<tr>
<th>AggOperator</th>
<th>COMP(countArray CntArr, lowerBound i, upperBound j, Statistics S, AggOperator Op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN</td>
<td>$S.min[k]$, where $k$ is the lowest non-empty bin ID</td>
</tr>
<tr>
<td>MAX</td>
<td>$S.max[k]$, where $k$ is the highest non-empty bin ID</td>
</tr>
<tr>
<td>SUM</td>
<td>$\sum_{k=i+1}^{j} S.sum[k] \times \frac{CntArr[k]}{S.count[k]}$</td>
</tr>
<tr>
<td>COUNT</td>
<td>$\sum_{k=i+1}^{j} CntArr[k]$</td>
</tr>
<tr>
<td>SQUARED SUM</td>
<td>$\sum_{k=i+1}^{j} S.squaredSum[k] \times \frac{CntArr[k]}{S.count[k]}$</td>
</tr>
<tr>
<td>LOG SUM</td>
<td>$\sum_{k=i+1}^{j} S.logSum[k] \times \frac{CntArr[k]}{S.count[k]}$</td>
</tr>
</tbody>
</table>

Finally, let bins from the lower bound $i$ to the upper bound $j$ be the output bitvectors, the aggregation result can be computed by an aggregation function $COMP(CntArr, i, j, S, Op)$, where the definitions for different aggregate operators are described in Table 4.1. $S$ consists the statistics including count, sum, squaredSum, as well as min and max, for each bin.

Returning to our example and further assuming that the given aggregate operator is SUM, from the pre-aggregation statistics, we can know that the count values for the two output bins are 2 and 3, respectively, and similarly, the sum values for the same two bins are 7 and 16, respectively. According to the aggregation function for the SUM operator described in Table 4.1, the aggregation result, sum of the two bins, can be estimated as, $7 \times \frac{1}{2} + 16 \times \frac{2}{3} = 14.167$. Note that this result is an approximate answer very close to the precise result, which will be 14.

A few other observations on the method can be made. If no value-based predicate is involved, the entire bitmap needs to be loaded to perform the bitwise AND operation with the predicate bitvector, leading to the worst-case approximation cost. Note that in this
case, there will be no error for the COUNT aggregation, because the number of elements aggregated within each bin is exactly equal to the corresponding value in the count array, which is calculated based on the accurate bitwise operations. On the other hand, if no dimension-based predicate is involved, both dimension-based filtering and count array generation can be avoided. Thus, only the pre-aggregation statistics stored in the metadata are involved in the aggregation, and bitvectors stored in the index file need not be loaded. One can view the COUNT field in the pre-aggregation statistics as the default value of count array. In this case, our method is essentially equivalent to the techniques based on 1-dimensional histograms.

Returning to Table 4.1, the implementation of operators MIN, MAX, and COUNT is simple, whereas the remaining two operators follow the same ideas as the SUM operator. It is also worth noting that once SUM, SQUARED SUM and COUNT are calculated, other common aggregate operators like AVERAGE, VARIANCE and STANDARD DEVIATION can be easily derived. For instance, if we denote the results of VARIANCE, SQUARED SUM, SUM, AVERAGE, and COUNT by $var$, $squaredSum$, $sum$, $avg$, and $count$, respectively, we can have:

$$var = \frac{squaredSum - 2 \times sum \times avg + avg^2}{count}$$

Further, since it can also be desirable to support user-defined aggregation functions, pre-aggregation statistics can be customized to support them. The only constraint is that we should be able to estimate the aggregation results based on the count array and the pre-aggregation statistics. As a specific example, if the scientist needs to support an aggregation function geometric mean, then we can add a user-defined pre-aggregation statistic $log sum$, which computes the sum of all the natural log values of the elements in each bin. For any subset of an array, we should be able to estimate the log sum in a similar fashion as other
functions in Table 4.1, and then the geometric mean can be computed as:

$$GeometricMean = \exp\left(\frac{\logSum}{\text{count}}\right)$$

where $\exp$ denotes the exponential function in the natural base.

4.3 V-Optimized Binning

As mentioned in Section 4.2.3, the existing binning strategies may not be able to provide satisfying accuracy compared with other approximate aggregation methods, especially when the data is skewed. In this section, we propose a novel $v$-optimized binning strategy for better accuracy. First, we discuss a unbiased binning strategy, where we assume that the possibilities of querying each element are equal, Second, we extend such binning strategy to a weighted bin, for further improving the accuracy when the querying possibilities are unequal and can be taken as prior knowledge.

4.3.1 Motivation

Through extensive experiments, we observed that when the data is skew, the existing binning strategies can lead to highly inaccurate aggregation results. This is because the purpose of the conventional binning strategies is to reduce the sizes of bitmap indices and thus enhance the efficiency of selection queries. Nevertheless, there may exist prominent difference between the real data distribution in the original dataset and the data distribution captured by bitmap indices after the conventional binning. As mentioned in Section 4.2.1, since the extra candidate check step can guarantee the accuracy of selection queries, such difference is not considered by conventional binning strategies. In comparison, to meet stringent response-time requirement, our approximate aggregation method cannot involve the costly candidate check. Thus, a new binning strategy is needed
for reducing the difference between the two distributions, which is critical for aggregation accuracy.

A common metric to measure the difference between the two distributions is *sum squared error* (SSE). Formally, let $N$ and $B$ be the number of elements and bins in a bitmap, respectively, and let $n_i$ and $\bar{v}_i$ be the number of elements and representative (mean) value in each bin, respectively, where $i \in B$, and $\sum_{i=1}^{B} n_i = N$. Additionally, the $j$th element in the $i$th bin is denoted as $v_{ij}$, where $1 \leq j \leq n_i$.

$$SSE = \sum_{i=1}^{B} \sum_{j=1}^{n_i} (\bar{v}_i - v_{ij})^2$$  \hspace{1cm} (4.1)

As Equation 4.1 shows, the SSE of a bitmap is represented by the cumulative of variance of all the bins. Thus, given by a number of bins, we propose a novel binning strategy - *v*-optimized. This strategy aims to effectively reduce such SSE, which indicates the total variance over the entire bitmap. Note that the number of bins should be fixed (i.e., space bound), since it is critical to control the size of bitmap indices for aggregation efficiency.

### 4.3.2 Unbiased V-Optimized Binning

First, we simplify the problem by assuming that all the elements are queried with the same probability. This assumption is often applied when the data is randomly queried or no prior knowledge of the querying possibilities can be provided. We take inspirations from the *v-optimal* histogram technique [103], because for a given number of buckets, a space-bound v-optimal histogram can effectively minimize the SSE. Overall, there are two major challenges of designing an effective binning strategy that maintains such *v*-optimal flavor. On one hand, v-optimal histogram construction requires quadratic time in the number of buckets [103], and even most of the near-optimal algorithms [86, 88, 211]
are still problematic for large-scale array data in terms of time efficiency. On the other hand, unlike the histogram construction, which divides a sequence of data in certain sort parameter (e.g., data value, frequency or area) into contiguous segments, bitmap binning can group discrete elements, where their positions are preserved by the associated bitvector. Thus, some alternative heuristic histogram construction methods such as MaxDiff [175] and DNS [211] cannot be applied. To address these challenges, we propose a (unbiased) v-optimized binning strategy, which aims to provide both near-optimal partitioning quality (i.e., near-minimum SSE) and practical time efficiency (i.e., near-linear time complexity).

The v-optimized binning strategy comprises three steps: 1) initial binning: we begin with equi-depth binning, which is shown to be more skew-tolerant than equi-width binning, to set up initial bin boundaries; 2) iterative refinement: to reduce the SSE, we can refine the partitioning by iteratively adjusting bin boundaries; 3) bitvector generation: Generate a bitvector for each bin to mark the positions of all the elements within the bin.

Clearly, if two neighboring bins are merged, then the SSE will increase, and conversely, if a single bin is split into two, then the SSE will decrease. Based on this property, the partitioning can be refined by iteratively merging two neighboring bins with small SSE increment and splitting one bin with large SSE decrement. To partition the given $N$ elements into $B$ bins, we need to maintain two priority queues, a potential SSE increment queue $Q^+$ of size $B - 1$, and a potential SSE decrement queue $Q^-$ of size $B$, respectively. Each element in $Q^+$ corresponds to a bin boundary in order, and its value represents the potential SSE increment if this boundary is removed (i.e., two bins beside this boundary are merged). The minimum value has the highest priority. In comparison, each element in $Q^-$ corresponds to a bin in order, and its value indicates the potential SSE
decrement if this bin is split by further using equi-depth binning. The maximum value has
the highest priority.

Algorithm 6: partition(num\_elements N, num\_bins B)

1: Initialize $B - 1$ bin boundaries with equi-depth binning
2: Initialize $Q^+$ and $Q^-$, where $E^+_{\text{min}}$ is the minimum element in $Q^+$ and $E^-_{\text{max}}$ is the maximum
element in $Q^-$
3: while $E^+_{\text{min}} < E^-_{\text{max}}$ do
4: remove the $i$th boundary that corresponds to $E^+_{\text{min}}$
5: update $Q^+$ and $Q^-$
6: split the $j$th bin that corresponds to $E^-_{\text{max}}$ in two
7: if $i = j$ then
8: break
9: end if
10: update $Q^+$ and $Q^-$
11: end while

Algorithm 6 shows the first two steps of the \( v \)-optimized binning. After the initial
equi-depth binning, the two priority queues are built. At each iteration, the boundary
corresponding to the minimal potential SSE decrement in $Q^+$ is removed, the bin
corresponding to the maximal potential SSE increment in $Q^-$ is split in two, and then two
priority queues are accordingly updated. This step is repeated until no further move
decreases the total SSE can be made. Finally, once the bin boundaries are finalized,
bitvectors are generated according to the positions of the elements within each bin. The
time complexity of \( v \)-optimized binning is $\Theta(N \log N + \frac{MN}{B})$, where $\Theta(N \log N)$ is the
cost of the initial equi-depth binning, $M$ is the number of iterations, and $\Theta(\frac{N}{B})$ is the costs
of the merge, split and update at each iteration. Note that all the iterative refinements are
operated on the two priority queues, and no operations on the bitmap is involved during
this step, the cost of such refinement is relatively low. Another minor detail is that, though
the elements in a bin are non-contiguous, it only requires $\Theta\left(\frac{N}{B}\right)$ time to iterate over them, because we have recorded their positions in a separate $\Theta\left(\frac{N}{B}\right)$ space.

Note that since v-optimized binning strategy is designed for approximate aggregations, and hence unlike equi-depth binning, it may not be able to effectively reduce the candidate check cost in the worst case and optimize the performance of processing selection queries. Therefore, our v-optimized strategy does not intend to replace its counterpart, but serves as a complementary tool for aggregation tasks. Since bitmap indices are often very compact compared with the original datasets, multiple copies of bitmap indices generated by different binning strategies can be kept at the same time, for handling different tasks.

4.3.3 Weighted V-Optimized Binning

The unbiased v-optimized binning strategy holds an important assumption - the possibilities of querying each element are equal. In the context of relational databases where only value-based predicates, violation of such assumption brings trivial influence on the aggregation accuracy. This is because the error is only incurred by aggregating a subset of elements in the edge bin(s), while all the other queried bins can guarantee the accuracy with pre-aggregation.

However, in our context, a unique challenge needs to be addressed if the assumption does not hold. When the a dimension-based predicate is applied, only a subset of elements within each bin are queried, and hence error can be incurred by each bin. In this case, the querying possibility of different data subareas should be considered in the binning strategy for better accuracy. Thus, we extend our unbiased v-optimized binning strategy to a weighted variant. We assume that the querying possibilities can be taken as prior knowledge, e.g., from query logs or from domain knowledge.
First, by adapting from the conventional SSE formulation, we define a new metric - 
*weighted sum squared error* (WSSE) - to measure the difference between the real data 
distribution in the original dataset and the data distribution captured by bitmap indices, 
given by the querying possibilities of all the elements. Formally, Let $w_{ij}$ be the weight of 
the $j$th element within the $i$th bin, where $i \in B$ and $1 \leq j \leq n_i$. Let the value of $w_{ij}$ 
represent the corresponding querying possibility, so that $\sum_{i=1}^{B} \sum_{j=1}^{n_i} w_{ij} = 1$. $WSSE$ is 
defined as:

$$WSSE = \sum_{i=1}^{B} \sum_{j=1}^{n_i} w_{ij} \times (v'_i - v_{ij})^2$$  \hspace{1cm} (4.2)$$

where $v'_i$ is the representative value of $b_i$.

Clearly, our weighted $v$-optimal binning strategy aims to effectively reduce such WSSE 
that indicates the cumulative weighted variance over the entire bitmap, when not all subsets 
are queried with the same probability. Compared with the unbiased binning strategy, we 
need to make two modifications for such weighted variant. First, Algorithm 6 can be easily 
adapted, by letting the two priority queues maintain the increments/decrements of potential 
WSSE instead of SSE. Second, we need to determine the representative values of all the 
bins and store them in the pre-aggregation statistics. The representative values can be 
determined by Theorem 1.

**Theorem 1.** To minimize $WSSE$, for each given bin $b_i$ with $n_i$ elements, we should set its 
representative value

$$v'_i = \frac{\sum_{j=1}^{n_i} w_{ij} \times v_{ij}}{\sum_{j=1}^{n_i} w_{ij}}$$  \hspace{1cm} (4.3)$$

**Proof.** Once the binning is finalized, since the value range of each bin $b_i$ is fixed, $n_i$ is a 
constant. Thus, WSSE is minimized only if each term $WSSE_i = \sum_{j=1}^{n_i} w_{ij} \times (v'_i - v_{ij})^2$
is minimized. To find the minimum, we differentiate $WSS_{E_i}$ with respect to $\overline{v}_i$:

$$WSS_{E_i}' = 2 \times \sum_{j=1}^{n_i} w_{ij} \times (\overline{v}_i - v_{ij}).$$

We solve $WSS_{E_i}' = 0$,

$$2 \times \sum_{j=1}^{n_i} w_{ij} \times (\overline{v}_i - v_{ij}) = 0$$

$$\Leftrightarrow \sum_{j=1}^{n_i} w_{ij} \times \overline{v}_i - \sum_{j=1}^{n_i} w_{ij} \times v_{ij} = 0$$

$$\Leftrightarrow \overline{v}_i = \frac{\sum_{j=1}^{n_i} w_{ij} \times v_{ij}}{\sum_{j=1}^{n_i} w_{ij}}$$

\[\Box\]

4.4 Experimental Results

In this section, we evaluate both the efficiency and accuracy of our approximate aggregation approach using both real-world and synthetic datasets. We designed the experiments with two goals. The first goal is to compare both the efficiency and accuracy of our method against sampling-based and histogram-based methods. Particularly, we used stratified random sampling [58], as well as multi-dimensional equi-depth histogram [162]. The second goal is to demonstrate the efficacy of $v$-optimized binning strategy (including both unbiased and weighted schemes), by comparing the aggregation accuracy and the indexing costs against other two conventional binning strategies – equi-width and equi-depth.

4.4.1 Experimental Setup

Our experiments were conducted using both real-world and synthetic datasets stored in HDF5, one of the popular array formats. The real-world dataset is referred to as aerosol, which has two dimensions – level and time, and it is obtained from ACOS.
GOSAT/TANSO-FTS Level 2 Full Physics Standard Product Set [1]. Because the dataset size we could obtain was less than 5 GB, we created a larger dataset, which had a size of 60 GB. The synthetic dataset freq simulates a series of frequencies based on the Zipf distribution [265] with a skew parameter of 0.5, and has a size of 30 GB. It has three dimensions—lat, long, and time.

We used all the aggregate operators defined in Table 4.1 in our experiments. It turns out that the accuracy trends of COUNT, SQUARED SUM, and LOG SUM are mostly similar to SUM (because the estimations mainly depend on the accuracy of COUNT). In comparison, the accuracy of MIN and MAX is different, but similar to each other. Thus, we only report the average relative error of SUM aggregation in most of our experiments, and add results from MAX aggregation in Section 4.4.2.

In terms of the data skew of the two datasets, we divided the value domain into two ranges: 1) dense range, which takes up less than 5% of the entire value domain but contains at least 90% of the data elements, and 2) sparse range, which contains fewer than 10% elements but takes up over 95% of the entire value domain. Thus, we could perform aggregations over dense and sparse range separately. Note that the division of ‘dense’ and ‘sparse’ can be adjusted based on the data skew, and it is easy to check if a given value range is dense or sparse, e.g., by histogram.

Moreover, recall that we had categorized all the subsetting predicates into three types in Section 4.1.1. Because we can apply different predicate types over dense and sparse ranges separately, we divided all the queries into five types: 1) with dimension-based predicate, 2) with value-based predicates over dense range, 3) with value-based predicates over sparse range, 4) with combined predicates over dense range, and 5) with combined predicates over sparse range. For brevity, we denote the above five query types as ‘DB’, ‘VBD’, ‘VBS’, ‘CD’, and...
Table 4.2: Query Examples of Five Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Query Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td><code>select sum(freq) from freq.h5 where long &lt; 200;</code>&lt;br&gt;<code>select sum(aerosol) from aerosol.h5 where time &gt; 4000000;</code></td>
</tr>
<tr>
<td>VBD</td>
<td><code>select sum(freq) from freq.h5 where freq &lt; 22.1;</code>&lt;br&gt;<code>select sum(aerosol) from aerosol.h5 where aerosol &lt; 0.06;</code></td>
</tr>
<tr>
<td>VBS</td>
<td><code>select sum(freq) from freq.h5 where freq &gt; 312;</code>&lt;br&gt;<code>select sum(aerosol) from aerosol.h5 where aerosol &gt; 9;</code></td>
</tr>
<tr>
<td>CD</td>
<td><code>select sum(freq) from freq.h5 where freq &lt; 22.1 and lat &gt; 500;</code>&lt;br&gt;<code>select sum(aerosol) from aerosol.h5 where aerosol &lt; 0.06 and level &gt; 200;</code></td>
</tr>
<tr>
<td>CS</td>
<td><code>select sum(freq) from freq.h5 where freq &gt; 312 and long &lt; 500;</code>&lt;br&gt;<code>select sum(aerosol) from aerosol.h5 where aerosol &gt; 9 and level &lt; 110;</code></td>
</tr>
</tbody>
</table>

and ‘CS’, respectively. The *queried coverage*, which indicates the ratio between the number of selected elements and the total number of elements in the entire dataset, was varied from 10% to 90% over the dense range. Over the sparse range, the queried coverage was varied from 0.01% to 5%, with value-based predicates, and from 0.005% to 2%, with combined predicates. For each query type, we used 90 queries on each dataset. We implemented our method on top of the precursor system [233, 234], which can execute aggregate queries in SQL over HDF5 datasets. Table 4.2 lists example queries of the five types.

When evaluating the weighted v-optimized binning, we assumed that 25% subarea of the real-world dataset and 50% subarea of the synthetic dataset were queried frequently. Particularly, the ratio of querying possibilities over the frequently queried subarea and the infrequently queried subarea was 10:1. When the queried coverage was less than the total coverage of the frequently queried subarea, we assumed that only the frequently queried subarea was queried, with an exception of the queries over the sparse range, as a frequently queried subarea could not be solely formed by the sparse range.
Figure 4.3: SUM Aggregation Accuracy of Different Methods on the Real-World Dataset

Our experiments were conducted on a machine which has an Intel(R) Xeon(R) Processor with 4 dual-core CPUs (8 cores in all). The clock frequency of each core is 2.53 GHz, and the system has a 12 GB main memory.

4.4.2 Aggregation Accuracy of Different Approximate Aggregation Methods

In this section, we experiment with the real-world dataset and compare our approach against other approximate aggregation methods. From the techniques listed in Section 4.1.2, we focused on techniques that could potentially handle both value-based and dimension-based predicates, and did not require any data reorganization. Thus, we chose two techniques to compare our method against – a popular sampling-based method, stratified random sampling [58], and a multi-dimensional equi-depth histogram-based
method [162]. Although we were also aware of a number of more sophisticated histogram-based methods [90, 164], we could not directly compare with them, mostly because the implementations were not available to us. However, we believe that our comparison with the basic histogram-based method still highlights the main differences between the histogram and bitmap based approaches.

Both sampling and histogram involve a trade-off between aggregation times and accuracy, based on sampling rate and the number of histogram buckets, respectively. Thus, to fully explore the capabilities of these two methods, we chose two different versions as follows. For the sampling-based method, we used two different sampling rates – 2% and 20%. In the implementation, a spatial region of size 128 MB is viewed as a stratum, and random sampling is applied to each such stratum, choosing the same fraction of data from each stratum. For the histogram-based method, the two dimensions, level and time, were treated in the same fashion as the (value-based) attribute aerosol. We also used two histograms of different sizes – the first (coarse-grained) histogram, which we referred to as HIST1, used 200, 18,000, and 200 buckets to partition the attributes level, time, and aerosol, respectively. The second, fine-grained, histogram, which we referred to as HIST2, used 200, 72,000, and 800 buckets to partition the same attributes. Particularly, HIST2 was even larger than the original dataset, due to the space consumed in storing dimension-based attributes.

As we will show later, equi-width binning is not a viable binning strategy for skewed data, and is not used in the set of experiments reported in this subsection. 400 bins were used for bitmap indexing. We evaluated the accuracy of SUM and MAX aggregations, as described below.
Figure 4.4: SUM Aggregation Times of Different Methods on the Real-World Dataset
Table 4.3: Average Relative Error(%) of MAX Aggregation of Different Methods on the Real-World Dataset

<table>
<thead>
<tr>
<th>Method</th>
<th>Query Type</th>
<th>DB</th>
<th>VBD</th>
<th>VBS</th>
<th>CD</th>
<th>CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling 2%</td>
<td>DB</td>
<td>67.874</td>
<td>2.492</td>
<td>58.476</td>
<td>0.000</td>
<td>86.321</td>
</tr>
<tr>
<td>Sampling 20%</td>
<td>DB</td>
<td>28.972</td>
<td>0.453</td>
<td>42.565</td>
<td>0.000</td>
<td>54.084</td>
</tr>
<tr>
<td>Hist1</td>
<td>DB</td>
<td>22.012</td>
<td>0.000</td>
<td>0.000</td>
<td>1.672</td>
<td>46.895</td>
</tr>
<tr>
<td>Hist2</td>
<td>DB</td>
<td>19.478</td>
<td>0.000</td>
<td>0.000</td>
<td>0.692</td>
<td>36.185</td>
</tr>
<tr>
<td>Equi-Depth</td>
<td>DB</td>
<td>24.241</td>
<td>0.000</td>
<td>0.000</td>
<td>1.941</td>
<td>42.168</td>
</tr>
<tr>
<td>V-Optimized</td>
<td>DB</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.694</td>
<td>0.000</td>
</tr>
</tbody>
</table>

**Accuracy of SUM Aggregations:** The results are shown in Figure 4.3. We first explain why results have not been shown for all methods for each case.

In applying sampling, one important observation was as follows. Above a certain threshold, a higher sampling rate does not lead to a noticeably better approximation of the spatial distribution and thus a higher accuracy, when only dimension-based predicates are applied (DB case). Moreover, when combined predicates are involved (CD and CS cases), the overall sampling inaccuracy is still dominated by the error caused by dimension-based predicates. Thus, it turns out that for the DB, CD, and CS cases, accuracy with 2% sampling is only at most 3% worse than with 20% sampling. As a result, in Figures 4.3(a), 4.3(d) and 4.3(e), only 20% sampling results are included.

Another note with respect to Figure 4.3 is that, there is no difference between unbiased and weighted v-optimized schemes when only valued-based predicates are involved (VBD and VBS cases). Thus, a single curve for v-optimized binning is shown for these cases.

Now, focusing on comparing our method against sampling and considering Figures 4.3(a),
4.3(d), and 4.3(e), we can see that the sampling is very inaccurate most of the times. In comparison, we can see that our bitmap-based method, with either equi-depth binning or v-optimized binning, can be significantly more accurate, because of a better approximation of the spatial distribution. Specifically, v-optimized binning gives a higher accuracy (measured by average relative error) by an average of 4,000,000x, 17x, and 80x for the DB, CD, and CS cases, respectively. However, the situation is quite different when only value-based predicates are involved. As Figures 4.3(b) and 4.3(c) show, sampling-based method, especially with 20% sampling rate, can achieve a higher accuracy in most cases. Note that the accuracy of our bitmap-based method, especially with v-optimized binning, is also relatively high – the relative error is normally less than 3%. This small error is caused by the edge bin(s) that overlap with the queried value range. The result is not surprising, since our method has been designed for handling dimension-based or combined predicates for array data, and cannot be expected to outperform existing methods when dimension-based predicates are not involved.

Next, we compare our method against the histogram-based method. As Figure 4.3(a) shows, with only dimension-based predicates, v-optimized binning is more accurate than HIST1 and HIST2 by an average of 50,000x and 25,000x, respectively. There are two reasons for this. First, the histograms can only provide an approximate estimation of the queried subset size, whereas our method can obtain the precise queried subset size after the dimension-based filtering based on bitmaps. Second, the implementation of histograms we used partitions the value-based attribute in an equi-depth manner, whereas v-optimized binning can give a better approximation of the value distribution. As shown in Figure 4.3(b), HIST2 is very effective in dealing with value-based predicates over the
dense range, but this high accuracy should be put in the context of its high storage cost, which is even greater than the original dataset.

Finally, in Figures 4.3(d) and 4.3(e), when combined predicates are involved, the accuracy of HIST1 is lower than equi-depth binning by an average of 189% and 101%, respectively, because of its coarse-grained partitioning over attributes. Similarly, since HIST2 has a finer partitioning than equi-depth binning, its accuracy is higher by an average of 2% and 36%, respectively. Compared with v-optimized binning, the histograms (especially HIST2) are more accurate over the dense range, but much less accurate over the sparse range. This is because the histograms consume more buckets than v-optimized binning for partitioning the dense range, leading to a better approximation of the dense range. Similarly, v-optimized binning involves more bins over the sparse range, and hence gives a finer partitioning over the sparse range. The similar conclusion can also be drawn from the comparison between equi-depth binning and (unbiased) v-optimized binning.

**Accuracy of MAX Aggregations:** Table 4.3 shows the accuracy results of the MAX aggregations. Since it turns out that the results of unbiased and weighted v-optimized binning are not noticeably different, we do not report the results of the weighted scheme here. Like the previous set of experiments, we only report an average relative error for each query type. Clearly, the accuracy of the sampling is the worst, with an exception of the CD case, which we will discuss below. We can see that v-optimized binning provides the highest accuracy in all the cases except for CD, because of the high-quality approximation of both the value and spatial distribution. Additionally, we can also see that the histograms are generally more accurate than equi-depth binning, because the former partitions all the dimension-based attributes, whereas the latter does not involve any priori partitioning along any bitvector.
Now, we explain why sampling is better for the CD case. The sampling-based method is conservative – it always uses the maximum value in the sampled data that has satisfied all the predicates, and hence it is likely to provide more accuracy over the dense range, where the distribution is more uniform. In contrast, as illustrated by the aggregation interface for MAX operator in Table 4.1, our bitmap-based method is aggressive, i.e., it always uses the maximum bin boundary value as the answer, which might be filtered out by dimension-based predicates. The histograms also work in the same aggressive manner, and have lower accuracy than the sampling only for the CD case.

Figure 4.5: SUM Aggregation Accuracy of Different Binning Strategies on the Synthetic Dataset
4.4.3 Aggregation Costs of Different Aggregation Methods

Our next experiment evaluated the aggregation costs of different approximate aggregation methods, including sampling-based, histogram-based, and our bitmap-based methods, as well as accurate aggregation (using all data elements and thus providing 100% accuracy). We experimented on the real-world dataset with the same queries used in Section 4.4.2. The results are shown in Figure 4.4, and do not include any preprocessing costs, i.e., the costs of obtaining sampled data, histogram construction, or bitmap indexing. It turns out that, with an exception of the queries with combined predicates over the dense range, there is no noticeable difference between the two v-optimized schemes. Thus, we only report the results of the weighted scheme for the CD case (Figure 4.4(d)).

We first compare our method against the sampling with two different rates. It can be seen that our method can outperform sampling in almost all the cases. As discussed in Section 4.2.3, if only value-based predicates are involved, only the pre-aggregation statistics stored in metadata are processed with our method, and when the sparse range is queried, only a small subset of bitmap is involved in the aggregation. Thus, the only case our method has higher execution times over sampling with a rate of 2% is in Figures 4.4(a) and 4.4(d). This is because when dimension-based predicates are involved over the dense range, a substantial fraction of bitvectors are processed with our method, whereas the size of queried subset in the sampled data can be much smaller, e.g., 0.2% of the original dataset when the queried coverage is 10%. However, as noted earlier, sampling method with 2% sampling rate has poor accuracy for these cases.

Next, we compare our method against the histogram-based method. Since HIST2 is even larger than the original dataset, leading to high computation costs, we do not compare with it here. We can make three observations as follows. First, HIST1 can
outperform our method in Figure 4.4(e), because of its coarser partitioning over sparse range. Second, HIST1 is slower than our method when only value-based predicates are involved (Figures 4.4(b) and 4.4(d)), mainly due to the additional costs of processing dimension-based attributes stored in the histogram. Another reason is that, our method extensively uses the bitwise operations over compressed bitvectors, which are also more efficient than the algebraic operations over histogram grids. Third, the performances of HIST1 and our method are close in Figures 4.4(a) and 4.4(d).

Finally, we also compare the three bitmap-based methods, i.e., equi-depth binning, as well as the unbiased and weighted v-optimized binning. As shown in Figures 4.4(b) and 4.4(c), with only value-based predicates, equi-depth binning and v-optimized binning have approximately the same performance, which is also better than other methods. This is because only the pre-aggregation statistics are processed in these two cases. When dimension-based or combined predicates are involved, the aggregation costs are roughly proportional to the number of bins that overlap with the queried value range. In Figure 4.4(a), i.e., the case when only dimension-based predicates are involved, all the bins have to be used for dimension-based filtering, and the performances of equi-depth binning and v-optimized binning are again identical. On the other hand, in Figure 4.4(d), which represents combined predicates over the dense range, equi-depth binning leads to a finer partitioning over the dense range, and thereby has to load more bins, leading to a higher aggregation cost. Additionally, in Figure 4.4(e), the unbiased scheme marginally outperforms the weighted scheme as well as equi-depth binning, though this comes with a reduced accuracy.

Clearly, we can see that our method can lead to the lowest aggregation costs when only value-based predicates are involved. In other cases, the performance of our method is
within the same order of magnitude of the best performance, which is achieved by either sampling with a rate of 2% or HIST1. Thus, our method provides the best overall performance.

Figure 4.6: SUM Aggregation Accuracy of Different Binning Strategies with Varying Number of Bins on the Synthetic Dataset
4.4.4 Aggregation Accuracy of Different Binning Strategies

To further evaluate the aggregation accuracy of different binning strategies, we conducted two sets of experiments on the synthetic dataset, by using both a fixed number of bins and a varying number of bins, respectively. Besides the binning strategies used in earlier experiments, we also experimented with equi-width binning.

**Using a Fixed Number of Bins:** In this experiment, 400 bins were used by all the binning strategies – equi-width, equi-depth, unbiased v-optimized, and weighted v-optimized. Figure 4.5 shows the results. Particularly, it turns out that, when no dimension-based predicate is applied or when only the sparse range is queried, the unbiased and weighted schemes are equally accurate. This is because value-based predicates can only marginally impact the aggregation accuracy (explained in Section 4.3.3), and weighting does not influence binning over the sparse range. Thus, we only show the results of the weighted scheme in Figures 4.5(a) and 4.5(d).

We can make four observations as follows. First, not surprisingly, as the queried coverage increases, the results obtained by all the binning strategies tend to be more accurate in almost all the cases. This is because generally the approximation will be better as more elements are involved in the aggregation. Second, we can see that equi-width binning is quite inaccurate in all the cases, and even more so over the sparse range. This is because equi-width binning is not good at preserving value distribution of the original dataset. Thus, we can conclude that equi-width binning is not a viable solution for aggregating skewed data.

Third, focusing on the comparison between equi-depth binning and (unbiased) v-optimized binning, we find that when only dimension-based predicates are involved, v-optimized binning is more accurate than equi-depth binning by an average of 1,000x.
This is because v-optimized binning can effectively reduce the average error over the entire value domain. In addition, we also observe that when only the sparse range is queried, the accuracy of v-optimized binning is also higher by an average of 10x, while equi-depth is slightly more accurate over the dense range. As also explained earlier in Section 4.4.2 when the histograms and v-optimized binning are compared, this is because v-optimized binning consumes more bins over the sparse range, and fewer bins over the dense range.

Lastly, we compare the two schemes of v-optimized binning. As Figures 4.5(a) and 4.5(d) show, when the queried subareas are mostly covered by the frequently queried subarea, the weighted scheme can achieve better accuracy than the unbiased scheme, especially when the queried coverage is close to the portion of frequently queried subarea in the entire dataset (50% in this experiment). Particularly, we can see that the weighted scheme is consistently more accurate until the queried coverage is over 70%. Once the queried coverage is over 50% and infrequently queried subarea is involved, the accuracy difference between the two schemes begins to narrow down. This is because the weighted scheme has a better approximation over the frequently queried subarea, whereas the unbiased scheme is better at approximating the infrequently queried subarea as well as the entire domain. Overall, the weighted scheme can be taken as a tradeoff of the accuracy over the frequently and infrequently queried subarea.

**Using a Varying Number of Bins:** In the next experiment, we evaluated the aggregation accuracy by varying the number of bins from 50 to 800, with combined predicates over both dense and sparse ranges. Here we only report the results over the sparse range, since it turns out that the results over both sparse and dense ranges draw the same conclusion. Additionally, recall that weighting does not influence binning over the sparse range. Thus,
the performance difference between the unbiased and weighted schemes is not noticeable, and we only report the results of the unbiased scheme.

Figure 4.6 shows the results of the three different binning strategies separately. We can see that, for equi-width binning, the accuracy improvement with the growing number of bins is trivial, because equi-width binning is very unlikely to utilize the increased number of bins to better approximate the value distribution. In contrast, for equi-depth binning and v-optimized binning, increasing the number of bins can lead to noticeable accuracy improvement. Particularly, such improvement is very significant for v-optimized binning before the number of bins reaches 400.

### 4.4.5 Indexing Costs of Different Binning Strategies

![Figure 4.7: Index Creation Times with Different Binning Strategies (Varying Number of Bins, Real-World Dataset)](image)

This experiment evaluated the indexing times and storage costs with different binning strategies using the real-world dataset. The number of bins used was varied from 50 to 800 for all schemes. The results are shown in Figures 4.7 and 4.8. The compression algorithm used was the Word-Aligned Hybrid method [243, 244].
First, as expected, both indexing times and space costs of all the binning strategies increase with a growing number of bins, which is due to the increasing computation and large number of bins, respectively. Second, the indexing times of both equi-depth binning and v-optimized binning are 3x - 7x of equi-width binning. This is because that, equi-width binning only requires a single pass over the dataset once the value domain is known, whereas equi-depth binning involves the overheads of either sorting the data or dynamically adjusting the bin boundaries, and v-optimized binning iteratively refines the output of equi-depth binning. The indexing times of unbiased and weighted v-optimized binning are higher than equi-depth binning with an equal number of bins by only an average of 15% and 18%, respectively. As explained in Section 4.2.3, this is because the iterative refinement only operates on two small priority queues, and is not very expensive. However, the index creation time advantage of equi-width binning should be put in the context of accuracy results reported earlier, which demonstrated that it is not a viable binning strategy for approximate aggregation.

The size of the bitmap after compression depends on not only the number of bins used, but also the compression ratio achieved. It turns out that binning strategy impacts
compression also. The space costs with equi-width binning are much less than the other schemes, because most of bins are empty when equi-width binning is applied to skewed data, allowing better compression. Again, this advantage does not necessarily imply that this scheme should be preferred, because the aggregation accuracy is really poor. For the other three schemes, we believe that a better approximation (i.e., a smaller SSE) can also lead to a better compression ratio, and this is the reason why the space requirements of v-optimized binning are lower, especially with a smaller number of bins. This turns out to be another advantage of v-optimized binning over equi-depth binning. For instance, with 200 bins, the size of the bitmap with unbiased v-optimized binning is only 15% of the size of the original dataset, and only 29% of the size with equi-depth binning.

Compared to bitmap indices, the construction times of HIST1 and HIST2 (in Section 4.4.2) are 1875 and 5972 seconds, respectively. Moreover, the sizes of HIST1 and HIST2 are 5.38 and 86 GB, respectively. As shown by the previous experiments, although HIST1 is relatively small, it cannot provide a satisfying aggregation accuracy, while HIST2 is often impractical due to the extremely large size. In comparison, bitmap indices can not only provide a higher aggregation accuracy, but also have a storage advantage. Although bitmap indexing is more expensive due to the bitvector construction, this disadvantage can often be well justified by frequent query processing (e.g., interactive data analysis).

4.4.6 Discussion

Our evaluation has been with commonly available scientific datasets. As is the case with scientific datasets, these have not been high-dimensional datasets. However, it does not imply that our approach cannot work with high-dimensional arrays. Since we are not
computing distances or correlations across dimensions, a high-dimensional array can be mapped to a 1-dimensional array, and a dimension-based predicate involving an arbitrary number of dimensions can still be represented as a bitvector.

4.5 Related Work

We now compare our work with the existing efforts on approximate query processing in the context of both relational databases and scientific data management, and the efforts closely related to bitmap indexing.

As stated previously, the existing approaches for approximate query processing can be mainly categorized into three types, i.e., the sampling-based, histogram-based, and wavelet-based techniques. The most popular sampling methods include simple random sampling, which randomly selects a certain percent of elements out of original dataset, and stratified random sampling [58], which first divides the dataset into strata and then performs random sampling within each stratum. We have already compared the efficiency and accuracy of our method over stratified random sampling, treating different spatial blocks as strata. Methods for improving sampling accuracy over skewed data have been extensively studied in the context of relational databases [29, 46–48, 105, 106, 110].

Another set of more recent efforts has concentrated on sampling in the context of scientific data management [171, 222, 239]. However, to the best of our knowledge, no existing sampling-based method is effective at preserving both spatial and value distribution of the original dataset, or alternatively (like KD-tree based stratified sampling [239]) requires reorganizing the data, which can be prohibitively expensive for large-scale datasets.

Various histogram construction algorithms have been proposed in the context of relational databases [41, 86–88, 101, 103, 175, 211]. As mentioned earlier, when no
dimension-based predicate is involved, our approach produces approximate answers solely based on the pre-aggregation statistics stored in the metadata, and the structure of these statistics is equivalent to a 1-dimensional histogram. Such histograms, however, are not capable of preserving spatial distribution. Multi-dimensional histograms [90, 162] may serve as an alternative solution, and can be applied to array data by treating dimensions as additional attributes. Our experiments have shown that this method is less efficient and less accurate than our method. Also, multi-dimensional histograms have primarily been used for selectivity estimation [135, 176] (an internal step in query optimization) as opposed to user aggregations in the past. Buccafurri et al. [41] focused on the problem of improving the frequency estimation inside each histogram bucket with fixed boundaries. Lastly, wavelets [45, 153, 220, 221] have been devised mainly for processing approximate range-sum queries over OLAP data cubes. This approach cannot efficiently process value-based predicates in an array view, and there are also limits on the aggregate operators that can be supported when wavelets are used to summarize data.

Online aggregation [30, 94, 134] is another closely related area. The idea is to provide a series of approximate answers for aggregate queries and incrementally improve the accuracy as the data scan proceeds. Again, we are not aware of online aggregation for array data. Among the techniques used, MRA-tree [134] is a generic tree structure with aggregation statistics for all its indexed nodes, and can be built on top of any tree index. Thus, MRA-tree can preserve either spatial distribution (i.e., using a quadtree or a K-D-B tree) or value distribution (e.g., based on the R-tree), but expensive data reorganization is required in each case. More recent works have been focused on online aggregation in the context of MapReduce [59, 60, 170].
Although bitmap indexing was initially proposed in the context of data warehouses [169, 247, 248], recently it has been widely applied in the area of scientific data management [56, 194, 195, 202–204, 206, 242]. To improve query efficiency, different binning strategies [78, 182, 197, 246] and encoding methods [122] have been proposed over time. Unlike our v-optimized binning strategy, which is better at reducing the variance between the real data distribution and its bitmap-based approximation, the most recent binning strategies are mainly designed for reducing the cost of candidate check [182] and accelerating selection queries. In other recent work from our group, we have applied our aggregation approach to a number of scientific analytics tasks, including correlation mining [205], subgroup discovery [236], and contrast set mining [264] – however, the underlying algorithms are different in each case.

### 4.6 Summary

This work has described a novel approximate aggregation approach over array data, by leveraging the fact that, bitmap indices can serve as a summary structure that well preserves both spatial and value distribution of data. Moreover, existing compression algorithms for bitvectors make bitmap a very compact structure, and fast bitwise operations allow efficient processing of queries involving dimension-based predicates. Our aggregation method, in conjunction with a novel binning strategy, enables high accuracy and low processing costs.
In the previous chapter, we have shown that flexible aggregations over array data can be approximated based on bitmap indexing and without involving any raw data. Motivated by this work, we find that bitmap indices also have the potential in handling a number of data mining tasks that involve frequent aggregations. In this chapter, we focus on a data mining task that can be broadly applied to scientific exploration – subgroup discovery. Particularly, we develop an efficient subgroup discovery algorithm, which directly operates on the compact bitmap indices instead of the raw datasets, and utilizes fast bitwise operations on them, allowing processing of larger datasets.

5.1 Subgroup Discovery and Array Data

We first introduce the formal definition of subgroup discovery. Next, we introduce an example involving arrays, and extend the definition of subgroups to apply to array data. Afterwards, to process numeric attributes in array data, we discuss the quality function we choose and subgroup combination.
5.1.1 Preliminaries

A comprehensive overview of subgroup discovery can be found in [95]. Subgroup discovery is a combination of predictive and descriptive induction, which focuses on the extraction of relations, with respect to the property of interest given by a target variable. Unlike classification techniques for addressing classification and prediction issues, subgroup discovery describes the local patterns of data in the form of individual rules. As also formulated by Gamberger et al. [73] and Lavrač et al. [133], a rule (R) that consists of an induced subgroup description can be formally defined as:

\[ R : \text{Cond} \rightarrow \text{Target}_{\text{val}} \]  

(5.1)

where \( \text{Target}_{\text{val}} \) is a value with respect to the property of interest given by a target variable, and \( \text{Cond} \) is commonly a conjunction of attribute-value pairs given by several explaining variables. A subgroup discovery task is mainly comprised of four elements: subgroup description language, quality function, target variable, and search strategy, described in the following paragraphs. In our description, \( \Omega_A \) denotes the set of all attributes, and for each attribute \( a \in \Omega_A \), a value domain \( \text{dom}(a) \) is defined.

Description 1 (Subgroup Description). A subgroup description is defined as a conjunction of attribute-value pairs, where for each attribute-value pair \( a_i = v_i, a_i \in \Omega_A \) and \( v_i \in \text{dom}(a_i) \).

An important measure associated with a subgroup is the quality, which measures the “interestingness” of the subgroup, and is used for extracting and ranking the rules. There is no consensus on the best quality function to be used for evaluating the interestingness of a subgroup.
Description 2 (Quality Function). Let $\Omega_{sd}$ denote the universal set of all the possible subgroup descriptions, and given a particular target variable $t \in \Omega_A$, a quality function $q$ is a mapping -- $q : \Omega_{sd} \times \text{dom}(t) \rightarrow R$, where $R$ is the space of real numbers denoting quality score.

The types of attributes including both explaining variables and target variables can be binary, categorical, or numeric. For processing numeric attributes, discretization is often utilized to divide a continuous value range into multiple discrete intervals.

5.1.2 An Example Involving Arrays

Our goal focuses on subgroup discovery over array-based datasets. We use an example to illustrate the idea, which is shown through Figure 5.1. This example involves three 2-dimensional arrays $A$, $B$, and $T$, where $T$ indicates the target variable, and $A$ and $B$ are the two explaining variables. Two dimensions are indexed from 0 to 4, and denoted as $i$ and $j$, respectively. One can view this dataset as a relational table forming 25 records, where each record has three attributes.

Our goal is to identify interesting subsets (or subgroups) of $T$ such that each subset is significantly different from the entire array $T$, which has a size of 25 and a mean value of 3.6. Each subgroup is evaluated by a quality score that is based on a combination of two metrics: subgroup size (or support) and the difference with respect to the mean between the subgroup and the entire array. One can identify several subgroups of interest from the target array $T$.

The identified subgroups can be described by: 1) dimensional ranges of $i$ and/or $j$, 2) value ranges of $A$ and/or $B$, and 3) both value ranges and dimensional ranges. The Subgroup 1 and Subgroup 2, which belong to the first type of subgroups, simply select
the first two rows and the first two columns of the array T, respectively. As an example of the second type of subgroup, the Subgroup 4 comprises the elements of T where the corresponding elements of A have a value between 1 and 2. The Subgroup 3 exemplifies the third type of subgroup, by selecting the elements of T within the second row and the third row, where the corresponding elements of B have a value between 6 and 7. In each of the subgroups of interest, the mean of the value of T within the subgroup is either significantly higher or lower than the average of the entire array, and is associated with either a positive or negative quality score.
5.1.3 Extended Subgroup Description

Compared with the relational data model that conventional subgroup discovery algorithms operate on, one key difference of array-based scientific datasets is that, every array element is explicitly indexed by an array subscript or coordinate value in each dimension. In practice, an interesting subgroup (i.e., data subset) may also correspond to a particular spatial subarea. For example, within a 3-dimensional space modeled by longitude, latitude and depth, given by a target variable salinity, perhaps it is not only influenced by certain value-based attributes like pressure and temperature, which correspond to different arrays in the same dimensional layout, but is also correlated with the dimension-based attributes like latitude and depth that indicate array dimensions.

Formally, we categorize all the attributes possibly involved in the subgroup description into two types: dimension-based and value-based. Let $\Omega^D_A$ and $\Omega^V_A$ denote the set of all dimension-based and value-based attributes, respectively. Each dimension-based attribute $a^d$ indicates an array dimension, and a corresponding dimensional domain $\text{dom}(a^d)$ is defined (based on dimension indices or coordinate values). Each value-based attribute $a^v$ indicates an array name, and a corresponding value domain $\text{dom}(a^v)$ is defined based on the element values in the array. For a dimension-based attribute $a^d_i$, from $\text{dom}(a^d_i)$, we can obtain a set of all the possible dimensional intervals $R^d_i$ - each interval $r^d_i$ comprises a set of contiguous values from $\text{dom}(a^d_i)$. Similarly, for a value-based attribute $a^v_i$, from $\text{dom}(a^v_i)$, we can obtain a set of all the possible value intervals $R^v_i$, where each interval $r^v_i$ comprises a set of contiguous values from $\text{dom}(a^v_i)$. Let $R^d$ and $R^v$ be the universal set of all the possible dimensional intervals and value intervals, respectively.
Description 3 (Extended Subgroup Description). A subgroup description is defined by a conjunction of attribute-range (or attribute-interval) pairs, where each attribute can be either dimension-based or value-based. Each dimension-based attribute-range pair is of the form $a_i^d = r_i^d$, where $a_i^d \in \Omega_A^d$, and $r_i^d \in R_i^d$, and each value-based attribute-range pair is of the form $a_i^v = r_i^v$, where $a_i^v \in \Omega_A^v$, and $r_i^v \in R_i^v$.

5.1.4 Quality Function and Subgroup Combination

The initial subgroup discovery algorithms [119, 240] were proposed to facilitate the data exploration in the medical domain, and most existing subgroup discovery algorithms [64, 82, 115, 133, 254] mainly consider binary and categorical attributes. Some of these algorithms process numeric attributes by a ‘straight-forward’ discretization, i.e., creating a few intervals, and then applying quality function and search strategy designed for categorical attributes. Our initial attempts in applying existing algorithms for scientific simulation datasets show that such strategies are not adequate. Thus, to process numeric attributes, two major steps besides discretization are taken.

**Quality Function:** Most quality functions used for subgroup discovery, such as interest, novelty, significance, specificity, and Weighted Relative Accuracy (WRAcc) [95], are only applicable to binary or categorical attributes. By contrast, the quality function applicable to numeric attributes usually involves weighing and aggregations over a subgroup. In our work, we consider the *mean* with respect to the target variable as the property of interest, although other statistics can also be considered. As also used in [27], the quality function we use is referred to as *Continuous Weighted Relative Accuracy (CWRAcc)*:

$$q_{CWRAcc} = \frac{n}{N} \times (m - M)$$ (5.2)
where \( n \) and \( N \) denote the subgroup size and general population size, respectively, and \( m \) and \( M \) denote the mean with respect to the target variable in the subgroup and the general population, respectively. Note that the value of the quality computed by either of these two functions can be either positive or negative, indicating \( m \) is either greater than or less than \( M \).

**Subgroup Combination:** Unlike the cases where one is processing only binary or categorical attributes, it is often reasonable to combine two subgroups concerning the same attribute but with adjacent value ranges, based on a set of combination rules. Otherwise, it is very likely that we will have a massive number of subgroups, of which the majority can be further merged into the ones of greater generality. For example, two adjacent ranges \( 11 \leq \text{depth} \leq 30 \) and \( 31 \leq \text{depth} \leq 50 \) may be combined as a larger range \( 11 \leq \text{depth} \leq 50 \). The combination rules we use will be discussed in Section 5.3.2 in details.

### 5.2 SciSD Algorithm

This section describes the design of our SciSD algorithm. Our algorithm extensively uses bitmap indices as a summary and efficient representation of data, instead of the actual array data. However, to simplify the presentation in this section, the algorithm will be presented as if it is operating on the array data. Use of bitmap indices to accelerate the algorithm will be described in Section 5.4.

#### 5.2.1 Tight Optimistic Estimates

Before discussing the search strategy used in the algorithm, we first introduce the notion of optimistic estimates. Optimistic estimates [82] aim to safely prune the search space, especially when monotonicity with respect to the property of interest cannot be applied.
An optimistic estimate of a given subgroup computes the upper bound of the quality, i.e., the maximal quality that can never be exceeded by any subset of this subgroup. A tight optimistic estimate means that, there exists a subset of the given subgroup such that the quality of this subset can reach the estimated value. Note that such a subset with the maximal quality does not necessarily correspond to a subgroup description [82].

Given a subgroup $sg$, we denote the quality of the subgroup as $q(\text{sg})$. Further, we denote by $sg_{pos}$ the subset comprising all elements whose values are greater than the mean of $sg$. Similarly, we denote by $sg_{neg}$ the subset comprising all elements whose values are lower than the mean of $sg$.

**Theorem 2.** For the Continuous Weighted Relative Accuracy quality function and a given subgroup $sg$, the tight optimistic estimate $oe(\text{sg})$ can be formulated as:

$$oe(\text{sg}) = \begin{cases} q(\text{sg}_{pos}), & \text{if } q(\text{sg}) > 0 \\ q(\text{sg}_{neg}), & \text{if } q(\text{sg}) < 0 \end{cases}$$  \hfill (5.3)

Theorem 2 has been proved by Atzmüller et al. [27]. Our search strategy will leverage optimistic estimates to safely prune the search space, as described in the next subsection.

### 5.2.2 Search Strategy

Our SciSD algorithm performs an exhaustive search, i.e., the space of all the possible subgroups needs to be explored. Clearly, the search space is exponential in the number of attribute-range pairs. Therefore, we use certain pruning measures as well as binning of both dimensional and value ranges to reduce the computation costs. The search for subgroups is based on set-enumeration tree [34], which ensures that every node is either visited only once or not visited at all (if is pruned). The equivalence between set-enumeration trees and canonical orderings has been proved in the literature [181]. Thus, a simple rule for creating the tree is to generate the children nodes by appending only those attributes that follow all
existing attributes in a given ordering. Both dimension-based and value-based attributes can be equally treated in such an ordering.

Figure 5.2: An Example of Set-Enumeration Tree

Figure 5.2 shows an example of set-enumeration tree corresponding to Figure 5.1 in Section 5.1. To keep the illustration simple, subgroup combination is not considered here. Moreover, let us assume the dimensional space of input arrays is only $2 \times 2$. Thus, both attributes $i$ and $j$ have two possible distinct values 0 and 1. In this tree, each node indicates not one, but a set of subgroup candidates. The set associated with each node comprises elements with conjunction over all distinct ranges of the attributes the node is associated with. For example, the node $i \land j$ indicates the subgroup candidates $i = 0 \land j = 0$, $i = 0 \land j = 1$, $i = 1 \land j = 0$, and $i = 1 \land j = 1$, and candidates generated by any
combination of these. Ordering is maintained while generating child nodes in the tree, e.g., the node $B$ can have children nodes $B \land i$ and $B \land j$, but not $B \land A$. For our explanation, we use the terms parent subgroup and child subgroup – for illustration, $A = 1$ is the parent subgroup of $A = 1 \land B = 2$, which is the child subgroup here.

**Algorithm 7**: SciSD(dim-based attr. universal set $\Omega^D_A$, val-based attr. universal set $\Omega^V_A$, dim-range universal set $R^D$, val-range universal set $R^V$)

1: Let $Q$ be a queue for searching subgroup candidates
2: Let $Q'$ be a queue for output subgroups
3: Push the root node (represents the general population) into $Q$ {This node will be pruned later}
4: while $Q$ is not empty do
5:   Let $sg_f$ be the first subgroup in $Q$
6:   for each attribute $a_i \in \Omega^D_A$ or $\Omega^V_A$, where $a_i$ follows the last attribute in the description of $sg_f$ in the given ordering do
7:     for each range $r_j \in R^D_i$ or $R^V_i$ do
8:       Let $sg_l$ be the last subgroup in $Q$ 
9:       Let $sg_k$ be the child subgroup candidate of $sg_f$, by adding a new attribute-range pair $a_i = r_j$
10:      if $sg_l$ can be combined with $sg_k$ then
11:        Update $sg_l$ by combining it with $sg_k$
12:      Let $sg'_l$ be the last subgroup in $Q'$
13:      $sg'_l \leftarrow sg_l$
14:      else if oe$(sg_k)$ is not pruned then
15:         Push $sg_k$ into $Q$
16:         if $sg_k$ is not pruned then
17:             Push $sg_k$ into $Q'$
18:         end if
19:      end if
20:   end for
21:   end for
22: end while
23: return $Q'$

Algorithm 7 shows our search strategy. We use two queues – subgroup candidate queue $Q$ and the output subgroup queue $Q'$ (lines 1 and 2). $Q$ stores the subgroups that can potentially be ‘interesting’ but require further validation, and $Q'$ stores ‘interesting’
subgroups after validation, i.e., the ones that are going to be a part of the output. Our search traverses the set-enumeration tree \textit{level by level}, starting from the root node that represents the general population without any subsetting (line 3). To identify attribute-range pairs that can contribute to ‘interesting’ subgroups, lines 6 and 7 iterate over each attribute in a given ordering (e.g., canonical ordering), and each range in order, respectively. For each subgroup candidate in subgroup candidate queue $Q$, it can produce a set of child subgroups by appending another attribute-value pair, where the appended attribute follows the last attribute in its parent subgroup (line 9).

For each produced child subgroup, we first consider \textit{subgroup combination} (lines 10 to 13). Line 10 attempts to combine adjacent subgroup candidates at the same level according to certain combination rules, which will be discussed in Section 5.3.2. Such subgroup combination is only considered between the last subgroup candidate in the subgroup candidate queue $Q$ and the most recently produced child subgroup.

Note that to prevent an over-pruning situation, i.e., a situation where two subgroups should not be pruned but their combined subgroup is pruned, we need to ensure that once two subgroups are combined, the combined subgroup will not be pruned. Lines 11 to 13 update the last subgroup in both queues $Q$ and $Q'$ after combination.

\textit{Subgroup pruning} occurs if the tentative subgroup combination fails (lines 14 to 19). Similar to subgroup combination, a \textit{pruning test} is applied based on certain pruning measures, which will be discussed in Section 5.3.1. The pruning process includes two steps. First, we check if the optimistic estimate of the subgroup can pass the pruning test (line 14). If so, the subgroup is viewed as a subgroup candidate, which is added to the subgroup candidate queue $Q$. By adding it to the queue $Q$, we can check if any of its child subgroups can be a qualified subgroup as the search proceeds (line 15). Otherwise, this
subgroup can be safely pruned because of the property of the optimistic estimate, and its child subgroups are not evaluated any further. If a subgroup qualifies because of the value of the optimistic estimate, we apply the same pruning test to the subgroup itself (line 16). Only if the subgroup still qualifies, it is added to the output subgroup queue $Q'$ as an output subgroup (line 17). In practice, all the output subgroups can be ranked according to the quality function.

5.3 Subgroup Pruning and Combination

In our algorithm, efficiency is maintained by subgroup pruning based on certain pruning measures, as well as subgroup combination rules based on a set of combination rules, which are explained in this section. Due to the space limit, all the theorem proofs can be found in an extended version [236].

5.3.1 Pruning Measures

Our algorithm uses three pruning measures to determine if a subgroup should be pruned or not. Two of them are relatively straight-forward - a minimum support (or size) of the subgroup and a minimum threshold quality, since a subgroup of interest should be large enough and significantly different from the general population.

**Pruning Measure 1** (Minimum Support). Given a subgroup $sg$, with $\text{sup}(sg) = \frac{n}{N}$, where $n$ and $N$ are the size of $sg$ and the general population, respectively, if $\text{sup}(sg) < \text{sup}_{\text{min}}$, $sg$ is pruned.

**Pruning Measure 2** (Minimum Absolute Quality). Given a subgroup $sg$, if $|q(sg)| < q_{\text{min}}$, $sg$ is pruned.
The third pruning measure we use is as follows – compared with the parent subgroup, a child subgroup should be considered only if it has a higher quality than the parent subgroup. The motivation is that if such a pruning rule is not used, a large number of small and redundant subgroups will be generated by the algorithm.

**Pruning Measure 3** (Minimum Relative Quality). Given a subgroup $sg_{\text{parent}}$ and its child subgroup $sg_{\text{child}}$, $sg_{\text{child}}$ is of interest only if

$$
\begin{align*}
q(sg_{\text{child}}) - q(sg_{\text{parent}}) &\geq q^r_{\text{min}}, & \text{if } q(sg_{\text{parent}}) > 0 \\
q(sg_{\text{parent}}) - q(sg_{\text{child}}) &\geq q^r_{\text{min}}, & \text{if } q(sg_{\text{parent}}) < 0
\end{align*}
$$

According to Algorithm 7, for the subgroup candidates that involve multiple attribute-range pairs, their non-root parent subgroups must have already satisfied Pruning Measure 2. By using Theorem 3, Pruning Measure 3 here can be considered as a sufficient condition of Pruning Measure 2 for these subgroup candidates.

**Theorem 3.** Given a parent subgroup $sg_{\text{parent}}$ that satisfies Pruning Measure 2, and its child subgroup $sg_{\text{child}}$, if both $sg_{\text{parent}}$ and $sg_{\text{child}}$ satisfy Pruning Measure 3, then $sg_{\text{child}}$ also satisfies Pruning Measure 2.

### 5.3.2 Combination Rules

Three combination rules are defined to determine if two subgroups can be combined or not – all three conditions must be met for two subgroups to be combined. First, intuitively, only the two subgroups, which belong to the same parent subgroup and are adjacent at the same level in the set-enumeration tree, can be combined. In other words, one of the prerequisites of subgroup combinability is the existence of adjacency.

**Combination Rule 1** (Existence of Adjacency). Given two subgroups described by two conjunctions of attribute-range pairs $a_1 = r_1 \land a_2 = r_2 \ldots \land a_n = r_n$ and $a'_1 = r'_1 \land a'_2 = r'_2 \ldots \land a'_n = r'_n$
\( r'_2 \ldots \land a'_n = r'_n \), respectively, then the two subgroups are considered adjacent and can be combined only if \( \forall i, 1 \leq i \leq n : a_i = a'_i, \forall i, 1 \leq i < n : r_i = r'_i \), and, \( r_n \) and \( r'_n \) are two adjacent dimensional ranges in \( R^D_n \) or value ranges in \( R^V_n \).

Second, as also mentioned earlier in Section 5.2.2, if any two adjacent subgroups can pass the pruning test, they may still be pruned after combination, due to the low quality of the combined subgroup. This should be avoided. Therefore, it is necessary to set a combination rule to ensure that the combined subgroup should pass the pruning test, i.e., one combination rule should be a sufficient condition of satisfying all the three pruning measures defined in Section 5.3.1. Note that as Algorithm 7 shows, one of the subgroups used for combination is from the output subgroup queue, and it has already passed the pruning test.

Here, we propose an easy-to-compute criterion referred to as difference homogeneity, which is defined as Combination Rule 2. A positive value of difference homogeneity indicates that the means with respect to the target variable in these two subgroups are homogeneous, i.e., either greater than or less than the mean of the general population at the same time. This combination rule can ensure that a combined subgroup pass the pruning test, and the proof can be found in the extended version.

**Combination Rule 2** (Difference Homogeneity). Let \( m_1 \) and \( m_2 \) be the mean with respect to the target variable in two given subgroups, and let \( M \) be the mean of the general population, then the two subgroups can be combined only if \( (m_1 - M) \times (m_2 - M) > 0 \).

Lastly, in practice it is helpful for the users that a subgroup does not spread too broadly. Therefore, it is necessary to evaluate the distribution purity of a combined subgroup. Before introducing the next combination rule, we need to propose another
criterion referred to as *Continuous Weighted Entropy* (CWE), which is adapted from the entropy used for evaluating the purity of classification:

\[
CWE = -\sum_{i=1}^{n} p(i) \times \log p(i) \times \frac{|v_i - M|}{v_n - v_1}
\]  

(5.4)

where \( n \) is the number of intervals for the target variable, \( p(i) \) and \( v_i \) denote the probability and representative value of the \( i \)th interval, respectively, and \( M \) represents the mean of the general population. Since \( v_1 \) and \( v_n \) are the minimum and maximum representative value, respectively, \( \frac{v_i - M}{v_n - v_1} \) here represents the *weight*, which implies the *normalized distance* from the \( i \)th interval to the mean of the general population. Therefore, for two subgroups of equal size, the \( CWE \) of a subgroup in which most elements are within the intervals remote from the mean of the general population, tends to be greater than, the other subgroup in which most elements are within the intervals close to the mean of the general population.

The third combination rule aims to restrict the extent to which adjacent subgroups can be combined, by introducing a user-specified parameter – *maximum continuous weighted entropy* \( CWE_{\text{max}} \).

**Combination Rule 3** (Distribution Purity). *If two given subgroups are combined into a subgroup \( s_{\text{combined}} \) and let its continuous weighted entropy be \( CWE(s_{\text{combined}}) \), then the two subgroups can be combined only if \( CWE(s_{\text{comb}}) \leq CWE_{\text{max}} \).*

This combination rule can effectively control the *granularity* of output subgroups and hence help the user to control the number of output subgroups at a manageable level. Generally, a larger maximum CWE value tends to generate fewer subgroups yet of greater generality and higher quality.
### 5.4 Algorithm Optimization Using Bitmaps

In this section, we discuss the algorithm acceleration using bitmaps. Note that our actual algorithm implementation is based entirely on bitmaps. For clarity in our write-up, we are presenting bitmaps as an optimization mechanism. Since our approach can involve search over all enumerations, it will require multiple passes on the entire dataset if bitmaps are not used. This is going to be prohibitively expensive on any disk-resident dataset.

![Diagram of Bitmap Usage](image)

**Figure 5.3: Use of Bitmaps in SciSD**

A key characteristic of our algorithm is that, the input is compact bitmap indices instead of the original raw datasets. Note that bitmap (indices) are not used in the conventional way indices are used, i.e., for querying actual data more efficiently. Instead, bitmaps are used as a summary representation of the data. Also note that bitmaps lose a certain level of precision because of binning for value-based attributes. However, to control costs, our subgroup discovery method must use binning of value-based and dimension-based attributes.
A bitmap example is shown in Figure 5.3. Besides the normal representation of the data with bitmaps, our algorithm uses two other types of bitmaps. Returning to our running example, assume that within a $2 \times 2$ space, 2 dimensions (i.e., dimension-based attributes) row and column, are denoted as $i$ and $j$, respectively. The subgroup discovery involves 3 arrays (i.e., value-based attributes) $T$, $A$, and $B$, where $T$ is the target variable. All the bitmap indices here are generated in a row-major fashion. Three types of bitmap indices are generated in this example: 1) (conventional) indices for value-based attributes, e.g., $T$, $A$ and $B$; 2) indices for dimension-based attributes, e.g., $i$ and $j$; and 3) two sign bitvectors – both a positive bitvector and a negative bitvector.

These bitvectors are used in the following fashion. With the first two types of indices, each attribute-range pair can be represented by a bitvector in the bitmap, whether the attribute is dimension-based or value-based. Particularly, the second type of indices are created in the same way as value-based attributes are indexed, i.e., by treating each coordinate value as an array element value. Equi-depth binning [26], which partitions the entire domain in such a way that each bin contains an approximately the same number of elements, is used for indexing both value-based and dimension-based attributes. Use of binning restricts the ranges (of both values and dimensions) that can be used in describing a subgroup. However, this has a very significant impact on the execution time of the algorithm, as both the size of bitvectors and number of potential subgroups to be considered is now restricted to a manageable level.

The two sign bitvectors are used for optimistic estimates. Particularly, the positive bitvector indicates all the elements greater than the mean with respect to target variable in the original dataset (2.5 in this example), and leads to a positive quality. Similarly, the negative bitvector results in a negative quality.
Compared with conventional bitmap indexing that only outputs type-1 indices, our approach generates more indices. However, it turns out that both the bitvectors for dimension-based attributes and two individual sign bitvectors do not add much space or time complexity. This is because of the regularity of array elements within the same dimensional range, leading to very high compression ratios (e.g., 1%).

Now, we explain how efficiency is achieved in our algorithm using bitmaps. First, since the size of bitmap indices is often only 15%-30% of the original data size, I/O performance can be clearly improved. In addition, specific algorithm steps are accelerated as follows. First, a conjunction of multiple attribute-range pairs can be calculated by bitwise AND operations over different bitvectors. As each attribute-range pair corresponds to a bitvector in our bitmaps, a subgroup described by a conjunction of multiple attribute-range pairs can now be represented by a single bitvector after bitwise AND operations. This property is particularly useful every time a parent subgroup produces a child subgroup by appending another attribute-range pair (see line 9 in Algorithm 7). In our running example, $A = 1 \land i = 2$ can be derived by $1001 \land 0011 = 0001$.

Next, a disjunction of two attribute-range pairs can be obtained by bitwise OR operation between two bitvectors. This can speed up the subgroup combination process in our algorithm (see line 11 in Algorithm 7), since the combination of two attribute-range pairs can be viewed as a disjunction. In the above example, $B \leq 2$ can be derived from $1001 \lor 0100 = 1101$.

Another benefit of bitmaps is that, both subgroup size and the mean value with respect to the target variable in a subgroup can be computed efficiently with bitmaps. This property can accelerate all the computations involved in our quality evaluations, pruning measures, and combination rules. As the membership of a subgroup can be described by a single
bitvector, the subgroup size can be easily obtained by counting the number of 1s in the bitvector. Moreover, if we denote the bitmap that corresponds to the target variable as the target bitmap, associated with the target bitmap, the mean with respect to the target variable in a subgroup can be calculated as follows:

\[ m = \frac{\sum_{i=1}^{n} v_i \times \text{COUNT}(b_i \land b)}{\sum_{i=1}^{n} \text{COUNT}(b_i \land b)} \]  

(5.5)

where \( m \) is the mean with respect to the target variable in the subgroup, \( n \) is the number of bitvectors in the target bitmap, \( v_i \) is the representative value of the bitvector \( b_i \) in the target bitmap, and \( b \) is the bitvector that represents the membership of the subgroup. It turns out that the mean value can be calculated efficiently and with reasonably high accuracy based on the above process. In the above example, as the subgroup \( A = 1 \) can be represented by the bitvector \( 1001 \), the mean with respect to \( T \) in this subgroup can be computed by

\[
\frac{1 \times \text{COUNT}(1000 \land 1001) + 4 \times \text{COUNT}(0001 \land 1001)}{\text{COUNT}(1000 \land 1001) + \text{COUNT}(0001 \land 1001)} = 2.5.
\]

Finally, the two sign bitvectors can facilitate the optimistic estimates discussed in Section 5.2.1 (as used in line 14 of Algorithm 7). In Equation 5.3, given a bitvector that indicates all the elements in a given subgroup \( sg \), a simple bitwise AND operation between this bitvector and the positive bitvector can result in the bitvector corresponding to \( sg_{pos} \). Similarly, \( sg_{neg} \) can be efficiently calculated with the negative bitvector. In the above example, to obtain the subset with element values greater than the mean in the subgroup \( A = 1 \) represented by the bitvector \( 1001 \), we can perform a bitwise AND between this bitvector and the positive bitvector, \( 1001 \land 0011 = 0001 \).

### 5.5 Experimental Results

In this section, we evaluate the performance (execution efficiency) of our algorithm as well as the quality of the output subgroups. We designed the experiments with the
following goals: 1) to compare our algorithm with SD-Map* [27], which is a popular subgroup discovery algorithm (applicable to relational data involving numerical attributes) – it has been implemented in an open-source software VIKAMINE [14], and 2) to demonstrate both performance of our algorithm and the quality of the output subgroups, by varying both the maximum continuous weighted entropy (CWE) and the number of bins used in bitmap indexing.

### 5.5.1 Experimental Setup

Our experiments were conducted using four real-life scientific datasets, which are all stored in NetCDF, one of the popular array formats. The first two datasets are downloaded from the World Ocean Atlas 2009 (WOA09) [15] monthly compositing data. Because these two datasets are generated on 5° and 1° grids, we refer to them as WOA09_5DEG and WOA09_1DEG, respectively. Each dataset comprises five attributes including apparent oxygen utilization (AOU), temperature, dissolved oxygen (DO), salinity, and oxygen saturation (OS), in a 4-dimensional space, which is modeled by longitude, latitude, depth, and time. The sizes of WOA09_5DEG and WOA09_1DEG datasets are 14 MB and 373 MB, respectively. The third dataset is obtained from World Ocean Atlas 2013 (WOA13) [16] annual compositing data, which is generated on 5° grids. Compared with the first two datasets, this dataset is modeled with the same four dimensions, but it comprises three more attributes – silicate, phosphate, and nitrate. We refer to this dataset as WOA13. The fourth dataset is generated by Parallel Ocean Program (POP) [112]. 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. The dataset mainly comprises four attributes, salinity, temperature, UVEL and VVEL, where UVEL and VVEL observe the velocity of ocean current in the grid-x and grid-y directions, respectively. POP generates 1.4 GB data for each attribute per time-slice, and each attribute is modeled with three dimensions: latitude, longitude, and depth. We only used one time-slice which is referred to as the POP dataset in our experiments, and thus, the total data size of the fourth dataset was 5.6 GB. The number of attributes involved in our experiments is up to 8, which we believe is a large number compared to the number of attributes likely of interest to any given scientist. Our experiments were conducted on a machine with 8 GB of main memory and Intel(R) Xeon(R) 2.53 GHz CPU. The version of VIKAMINE we used was 2.2.

5.5.2 Comparison with SD-Map*

Our first set of experiments compared the performance of our algorithm and the quality of its output subgroups against SD-Map*, a popular subgroup discovery algorithm over numeric attributes. SD-Map* has been implemented in an open-source software VIKAMINE, and it uses the same quality function – Continuous Weighted Relative Accuracy (CWRAcc), which is defined by Equation 5.2. Although we are also aware of a small number of other algorithms that can work with numeric attributes, we cannot directly compare with them, because they either use different quality functions, and/or their implementations were not available to us. Since VIKAMINE can only support subgroup discovery over small relational datasets, we only used WOA09_5DEG and WOA13 these two datasets, which have a size of 14 MB and 8 MB, respectively. To match
the format expected by VIKAMINE, we converted the data from NetCDF to the CSV format. In the process, the four dimension-based attributes were also added as additional columns, since VIKAMINE can only process relational data. Note that treating array dimensions as additional relational attributes is not a practical approach for large datasets, due to the high data reorganization and storage costs.

First, among the five value-based attributes in WOA09_5DEG dataset, we set $\sigma_S$ as the target variable – its mean value is 71.03. We used 100 bins for indexing each value-based attribute, and 12 bins for indexing each dimension-based attribute. The minimum support, minimum absolute quality, and minimum relative quality were set as 0.01, 0.09, and 0.001, respectively. The maximum CWE was varied from 0.25 to 1.5.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Subgroup</th>
<th>CWRAcc</th>
<th>Support</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AOU &lt; 0.08</td>
<td>6.24</td>
<td>0.20</td>
<td>102.24</td>
</tr>
<tr>
<td>2</td>
<td>depth &lt; 40</td>
<td>6.08</td>
<td>0.21</td>
<td>99.81</td>
</tr>
<tr>
<td>3</td>
<td>AOU &lt; 0.08 AND depth &lt; 40</td>
<td>4.65</td>
<td>0.15</td>
<td>102.45</td>
</tr>
<tr>
<td>4</td>
<td>DO $\geq$ 6.14</td>
<td>4.63</td>
<td>0.20</td>
<td>94.18</td>
</tr>
<tr>
<td>5</td>
<td>0.08 $\leq$ AOU &lt; 0.75</td>
<td>4.62</td>
<td>0.20</td>
<td>94.13</td>
</tr>
<tr>
<td>6</td>
<td>temperature $&gt; 18.41$</td>
<td>4.55</td>
<td>0.20</td>
<td>93.79</td>
</tr>
<tr>
<td>7</td>
<td>4.98 $\leq$ DO &lt; 6.14</td>
<td>3.85</td>
<td>0.20</td>
<td>90.29</td>
</tr>
<tr>
<td>8</td>
<td>temperature $&gt; 18.41$ AND AOU &lt; 0.08</td>
<td>3.56</td>
<td>0.11</td>
<td>102.14</td>
</tr>
</tbody>
</table>

Tables 5.1 and 5.2 show the best 8 subgroups discovered by SD-Map* and our algorithm on WOA09_5DEG dataset, respectively, with the maximum CWE of 1.5. The subgroups within each table are in descending order of absolute quality measured by CWRAcc.
Table 5.2: The Best 8 Subgroups Discovered on WOA09_5DEG by SciSD with Maximum CWE of 1.5

<table>
<thead>
<tr>
<th>Rank</th>
<th>Subgroup</th>
<th>CWRAcc</th>
<th>Support</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.43 ≤ AOU ≤ 1.95</td>
<td>12.26</td>
<td>0.58</td>
<td>92.17</td>
</tr>
<tr>
<td>2</td>
<td>2.68 ≤ AOU ≤ 7.99</td>
<td>-12.25</td>
<td>0.34</td>
<td>35.00</td>
</tr>
<tr>
<td>3</td>
<td>0.01 ≤ DO ≤ 4.29</td>
<td>-11.61</td>
<td>0.36</td>
<td>38.78</td>
</tr>
<tr>
<td>4</td>
<td>4.43 ≤ DO ≤ 10.57</td>
<td>11.44</td>
<td>0.60</td>
<td>90.10</td>
</tr>
<tr>
<td>5</td>
<td>0 ≤ depth ≤ 200</td>
<td>9.30</td>
<td>0.51</td>
<td>89.34</td>
</tr>
<tr>
<td>6</td>
<td>250 ≤ depth ≤ 1500</td>
<td>-9.30</td>
<td>0.49</td>
<td>52.14</td>
</tr>
<tr>
<td>7</td>
<td>2.36 ≤ temperature ≤ 12.97</td>
<td>-6.57</td>
<td>0.50</td>
<td>57.89</td>
</tr>
<tr>
<td>8</td>
<td>16.16 ≤ temperature ≤ 32.68</td>
<td>5.15</td>
<td>0.25</td>
<td>91.64</td>
</tr>
</tbody>
</table>

We can make the following four observations about the two algorithms. First, our algorithm can discover more interesting subgroups (i.e., subgroups of higher quality). For example, the quality of the best 4 subgroups discovered by our algorithm is almost twice the best subgroup discovered by SD-Map*. This is because Combination Rule 2 leads to subgroups that are larger and of higher quality. Second, SD-Map* is only able to discover the subgroups with the mean value significantly greater than the general population, i.e., the subgroups with a positive CWRAcc. Our algorithm can also discover interesting subgroups with a negative CWRAcc. This is because our Pruning Measure 2 is also able to capture interesting subgroups with significantly smaller means. Third, as Table 5.1 shows, a nontrivial fraction of subgroups generated by SD-Map* are redundant. Subgroup 3 is really the intersection of Subgroup 1 and Subgroup 2, and in fact, has a lower CWRAcc than either of its two super-subgroups. Similarly, Subgroup 8 is an analogous intersection of Subgroup 1 and Subgroup 6. In contrast, our algorithm does not result in any subgroup that is redundant. This is because our Pruning Measure 3 can...
prevent any subgroup specialization that does not lead to quality gain. Finally, the subgroups discovered by our algorithm tend to have greater generality, i.e., larger subgroup size, while we observed that most subgroups discovered by SD-Map* could have been merged into larger subgroups. For example, Subgroup 1 and Subgroup 5, as well as Subgroup 4 and Subgroup 7 in Table 5.1, could have been combined into a larger subgroup. In comparison, our algorithm can combine as many subgroups as possible, under the user-specified maximum CWE. Therefore, our algorithm can ensure that any two of the output subgroups cannot be further combined, facilitating better exploration from scientists.

Table 5.3: Comparison between SD-Map* and SciSD on WOA09_5DEG with Varying Maximum CWE

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># of Subgroups</th>
<th>Exe Times (secs)</th>
<th>CWRAcc</th>
<th>Support</th>
<th># of Attr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SciSD_0.25</td>
<td>111</td>
<td>58.61</td>
<td>0.75</td>
<td>0.04</td>
<td>1.05</td>
</tr>
<tr>
<td>SciSD_0.5</td>
<td>88</td>
<td>48.53</td>
<td>1.03</td>
<td>0.06</td>
<td>1.06</td>
</tr>
<tr>
<td>SciSD_1</td>
<td>53</td>
<td>32.67</td>
<td>1.90</td>
<td>0.12</td>
<td>1.10</td>
</tr>
<tr>
<td>SciSD_1.5</td>
<td>26</td>
<td>17.37</td>
<td>4.17</td>
<td>0.27</td>
<td>1.19</td>
</tr>
<tr>
<td>SD-Map*</td>
<td>1792</td>
<td>2116</td>
<td>0.48</td>
<td>0.02</td>
<td>3.06</td>
</tr>
</tbody>
</table>

Table 5.3 summarizes the statistics of the subgroups discovered by SD-Map* and our algorithm with varying maximum CWE on WOA09_5DEG dataset. ‘SciSD_0.25’, ‘SciSD_0.5’, ‘SciSD_1’, and ‘SciSD_1.5’ are denoted as the algorithm with the maximum CWE of 0.25, 0.5, 1, and 1.5, respectively. These statistics include the number of output subgroups, execution times, average support, average CWRAcc, and the number of
attributes involved per subgroup. Note that the reported execution times of SD-Map* does not include the data preprocessing time, i.e., the data conversion time.

We can make three observations. First, we can see that, our algorithm not only produces subgroups of greater generality and higher quality than SD-Map*, but the execution times are also lower by a factor of up to two orders of magnitude. Second, our algorithm manages to restrict the number of subgroups within a manageable level, whereas SD-Map* reports a large number of subgroups, which tend to be too specialized. A prominent example is that, with respect to the attribute $AOU$, our algorithm with the maximum CWE of 1.5 only discovered two subgroups, with positive and negative quality, respectively. In comparison, SD-Map* discovered 766 subgroups, and all of them turn out to be overlapping subsets of the only subgroup with the positive quality discovered by our algorithm (this subgroup is the $Subgroup$ 1 in Table 5.2). Note that VIKAMINE can support a post-processing module named ‘minimal improvement filter’, which aims to filter out redundant subgroups and minimize the final output, but such post-processing currently is not supported over numeric attributes. Lastly, we can see that maximum CWE can be used to effectively control the granularity of the output subgroups. Generally, a larger maximum CWE can allow more subgroups to be combined, leading to fewer output subgroups, which are of greater generality and higher quality. A larger maximum CWE results in lower execution times also.

Next, we experimented on another dataset WOA13 by setting nitrate as the target variable – its mean value is 6.32. We used 100 bins for indexing each value-based attribute, and 12 bins for indexing each dimension-based attribute, with an exception of time dimension. Only 1 bin was used for indexing time, because this annual compositing dataset only comprises a single time slice. Our method set the minimum
support, minimum absolute quality, and minimum relative quality as 0.002, 0.14, and 0.001, respectively. The maximum CWE was varied from 0.25 to 1.5.

Table 5.4: The Best 8 Subgroups Discovered on WOA13 by SD-Map*

<table>
<thead>
<tr>
<th>Rank</th>
<th>Subgroup</th>
<th>CWRAcc</th>
<th>Support</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.16 ≤ phosphate</td>
<td>3.22</td>
<td>0.20</td>
<td>22.40</td>
</tr>
<tr>
<td>2</td>
<td>1.16 ≤ phosphate AND time = 1</td>
<td>3.22</td>
<td>0.20</td>
<td>22.40</td>
</tr>
<tr>
<td>3</td>
<td>12.75 ≤ silicate AND 1.16 ≤ phosphate</td>
<td>2.66</td>
<td>0.15</td>
<td>23.74</td>
</tr>
<tr>
<td>4</td>
<td>12.75 ≤ silicate AND 1.16 ≤ phosphate AND time = 1</td>
<td>2.66</td>
<td>0.15</td>
<td>23.74</td>
</tr>
<tr>
<td>5</td>
<td>12.75 ≤ silicate</td>
<td>2.65</td>
<td>0.20</td>
<td>19.61</td>
</tr>
<tr>
<td>6</td>
<td>12.75 ≤ silicate AND time = 1</td>
<td>2.65</td>
<td>0.20</td>
<td>19.61</td>
</tr>
<tr>
<td>7</td>
<td>1.16 ≤ phosphate AND 15.50 ≤ depth</td>
<td>0.84</td>
<td>0.05</td>
<td>23.30</td>
</tr>
<tr>
<td>8</td>
<td>1.16 ≤ phosphate AND 15.50 ≤ depth AND time = 1</td>
<td>0.84</td>
<td>0.05</td>
<td>23.30</td>
</tr>
</tbody>
</table>

Table 5.5: The Best 8 Subgroups Discovered on WOA13 by SciSD with Maximum CWE of 1.5

<table>
<thead>
<tr>
<th>Rank</th>
<th>Subgroup</th>
<th>CWRAcc</th>
<th>Support</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>phosphate &lt; 0.84</td>
<td>-3.41</td>
<td>0.74</td>
<td>1.71</td>
</tr>
<tr>
<td>2</td>
<td>0.88 ≤ phosphate &lt; 3.39</td>
<td>3.40</td>
<td>0.25</td>
<td>19.90</td>
</tr>
<tr>
<td>3</td>
<td>silicate &lt; 8.45</td>
<td>-2.66</td>
<td>0.75</td>
<td>2.76</td>
</tr>
<tr>
<td>4</td>
<td>13.90 ≤ silicate &lt; 185.23</td>
<td>2.65</td>
<td>0.19</td>
<td>20.20</td>
</tr>
<tr>
<td>5</td>
<td>40 ≤ depth &lt; 175</td>
<td>-0.98</td>
<td>0.25</td>
<td>2.35</td>
</tr>
<tr>
<td>6</td>
<td>400 ≤ depth &lt; 1050</td>
<td>-0.81</td>
<td>0.20</td>
<td>2.24</td>
</tr>
<tr>
<td>7</td>
<td>0 ≤ depth &lt; 35</td>
<td>0.80</td>
<td>0.08</td>
<td>16.17</td>
</tr>
<tr>
<td>8</td>
<td>200 ≤ depth &lt; 375</td>
<td>0.76</td>
<td>0.06</td>
<td>19.39</td>
</tr>
</tbody>
</table>

Tables 5.4 and 5.5 show the best 8 subgroups discovered by SD-Map* and our algorithm (with the maximum CWE of 1.5) on WOA13 dataset, respectively. We can still draw
Table 5.6: Comparison between SD-Map* and SciSD on WOA13 with Varying Maximum CWE

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># of Subgroups</th>
<th>Exe Times (secs)</th>
<th>CWRAcc</th>
<th>Support</th>
<th># of Attr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SciSD_0.25</td>
<td>37</td>
<td>17.69</td>
<td>0.39</td>
<td>0.059</td>
<td>1</td>
</tr>
<tr>
<td>SciSD_0.5</td>
<td>33</td>
<td>14.64</td>
<td>0.69</td>
<td>0.180</td>
<td>1</td>
</tr>
<tr>
<td>SciSD_1</td>
<td>19</td>
<td>10.27</td>
<td>1.00</td>
<td>0.189</td>
<td>1.16</td>
</tr>
<tr>
<td>SciSD_1.5</td>
<td>16</td>
<td>6.12</td>
<td>1.06</td>
<td>0.140</td>
<td>1.30</td>
</tr>
<tr>
<td>SD-Map*</td>
<td>3036</td>
<td>508</td>
<td>0.24</td>
<td>0.017</td>
<td>4.28</td>
</tr>
</tbody>
</table>

the same conclusions as those from the previous experiment. Particularly, in Table 5.4, we can see that the best 8 subgroups discovered by SD-Map* are actually all covered by either Subgroup 1 or Subgroup 5. The other 6 subgroups are all subsets of these two subgroups. In comparison, among the best 8 subgroups discovery by our algorithm, if we only consider the subgroups with a positive CWRAcc, Subgroup 2 in Table 5.5 has higher quality than Subgroup 1 in Table 5.4, and Subgroup 4 in Table 5.5 has no lower quality than Subgroup 5 in Table 5.4.

Similar to Table 5.3, Table 5.6 summarizes the comparison results on WOA13 dataset. It turns out that we can make the similar observations as those from the previous experiment. First, the average quality of the output subgroups discovered by our algorithm is 1.6x - 4.4x of SD-Map*'s output. Second, our execution time is only 1.2% of SD-Map*, when a large maximum CWE (1.5) is used. This is because lenient subgroup combination can effectively reduce the processing costs. Third, compared with SD-Map*, which produces a large number of redundant subgroups, our output subgroups are much more compact. For instance, among the total 3036 output subgroups, the attribute phosphate occurs 2187 times, but only in the attribute-range pair ‘1.16 ≤ phosphate’, which
corresponds to Subgroup 1 in Table 5.4. This implies that the other 2186 subgroups (or 72% of output) involving phosphate are all redundant. In addition, 50% of the output subgroups discovered by SD-Map* are simply created by involving one more attribute-range pair time = 1, which does not help improve the quality.

5.5.3 Evaluation over Larger Datasets

We next experimented on the two larger datasets – WOA09_1DEG and POP datasets. In this set of experiments we varied the maximum CWE and the number of bins used for indexing value-based attributes. One of our main goals was to understand how binning (a critical step in using bitvectors) impacts the execution time and the output quality of the algorithm. We set the minimum support, minimum absolute quality, and minimum relative quality as 0.01, 0.04, and 0.001, respectively. For WOA09_1DEG dataset, the maximum CWE was varied to be 0.25, 0.5, 1, and 1.5. The number of bins used for indexing each of the value-based attributes was varied between 20, 40, 60, 80, and 100. The number of bins used for indexing dimension-based attributes – latitude, longitude, depth, and time, was fixed throughout all the experiments, and was 18, 36, 24, and 12, respectively, for these four dimensions. For POP dataset, the maximum CWE was varied from 0.25 to 1. The number of bins used for indexing each of the value-based attributes was varied from 20 to 60. The number of bins used for indexing the dimension-based attributes latitude, longitude, and depth was also fixed – 24, 36, and 21, respectively, for these three dimensions.

Figures 5.4 and 5.5 show the evaluation results on WOA09_1DEG and POP datasets. First, we can see that using a larger maximum CWE and fewer bins can result in fewer subgroups being output. This is because a larger maximum CWE allows more small
Figure 5.4: Evaluation on WOA09_1DEG Dataset: Impact of Varying Maximum CWE and # of Bins on # of Subgroups, Execution Times, and Average CWRAcc
Figure 5.5: Evaluation on POP Dataset: Impact of Varying Maximum CWE and # of Bins on # of Subgroups, Execution Times, and Average CWRAcc
subgroups to be combined, and fewer bins lead to more coarse-grained binning and fewer attribute-range pairs. Second, in most cases, a larger maximum CWE and use of fewer bins also lead to lower processing times. The reason is two-fold. On one hand, more lenient subgroup combination can help eliminate more search space where multiple attributes are involved. On the other hand, use of fewer bins also implies fewer attribute-range pairs involved in the subgroup discovery. Finally, for both datasets, we can see higher average quality with larger maximum CWE and smaller number of bins. However, note that the subgroup quality cannot always be improved by increasing maximum CWE or decreasing the number of bins, since such quality is also highly dependent on the dataset itself. Two exceptions can be found in Figure 5.4(c). The first exception is that, when the number of bins is 20, using the maximum CWE of 1.5 cannot result in a subgroup of quality higher than using the maximum CWE of 1. The other exception is that, when the number of bins is greater than 20 and the maximum CWE is 1.5, the subgroup quality cannot be further improved with more bins. However, overall, high-quality results and reduced execution times with 20 bins point to the practicality of our algorithm, i.e., it can be effective on large datasets and can provide reasonable response times.

5.5.4 Effectiveness of Search Strategy

Lastly, we evaluated the search efficiency on WOA09_1DEG dataset, by using almost the same parameters as the previous experiment (exception being that a fixed maximum CWE of 1.5 was used). In Table 5.7, we report the size of search space, the number of visited subgroups during our search, the number of visited subgroups eliminated by
Table 5.7: Search Efficiency Evaluation on WOA09_1DEG with Varying # of Bins and Maximum CWE of 1.5

<table>
<thead>
<tr>
<th># of Bins</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Search Space</td>
<td>3.0E+10</td>
<td>4.8E+11</td>
<td>2.4E+12</td>
<td>7.6E+12</td>
<td>1.9E+13</td>
</tr>
<tr>
<td># of Visited Subgroups</td>
<td>1626</td>
<td>2216</td>
<td>2676</td>
<td>3136</td>
<td>3596</td>
</tr>
<tr>
<td># of Pruned Subgroups</td>
<td>1470 (90.4%)</td>
<td>1984 (89.5%)</td>
<td>2366 (88.4%)</td>
<td>2752 (87.8%)</td>
<td>3139 (87.3%)</td>
</tr>
<tr>
<td># of Combined Subgroups</td>
<td>136 (8.4%)</td>
<td>211 (9.5%)</td>
<td>289 (10.8%)</td>
<td>363 (11.6%)</td>
<td>436 (12.1%)</td>
</tr>
</tbody>
</table>

pruning, and the number of visited subgroups eliminated by combination. Additionally, we also report the proportion of eliminated subgroups among all the visited ones.

The size of search space equals the product of the number of bins over each attribute. The number of eliminated subgroups reported only includes the ones that are directly pruned or combined based on our pruning measures and combination rules, and excludes the child subgroups of those. We can make the following observations. First, the search space increases exponentially as more bins are used, and even with a small number of bins, the search space is still huge. This shows that a brute-force algorithm will have an unacceptable cost. However, the number of visited subgroups is at least 7 orders of magnitude smaller, and does not increase very rapidly with increasing number of bins. This is because our level-wise search strategy can eliminate unqualified or small subgroups by pruning or combination as early as possible, and hence unnecessary exploration over their child subgroups can be avoided. As a result, the depth of our search tree is relatively small, mostly less than 3. Second, we can see that around 90% of visited subgroups are pruned based on our pruning measures, and around 10% visited subgroups are combined to provide a higher quality. Therefore, generally only less than 1% of visited subgroups are finally qualified, leading to subgroups of most significance to the users.
5.6 Related Work

As an important data mining technique that is often applied to data exploration and descriptive induction, subgroup discovery has been extensively studied in recent years. The existing algorithms can be broadly classified into three types: *extensions of classification algorithms* [73, 119, 130, 133, 240], *extensions of association algorithms* [28, 81, 82, 115], and *extensions of evolutionary fuzzy algorithms* [35, 43, 64]. However, only a small number of subgroup discovery algorithms are capable of processing numeric attributes. As an extension of SD-Map [28], SD-Map* is designed based on FP-growth, and it uses frequent pattern tree as the search tree. In contrast, our algorithm uses a set-enumeration tree based search strategy, with pruning and combination methods we have introduced. We have extensively compared our algorithm with SD-Map*, and have shown that our algorithm can lead to the subgroups of greater generality and higher quality, as well as up to two orders of magnitude lower execution times.

Other efforts have focused on discretization methods – for example, TargetCluster [161] discretizes target variable based on clustering, instead of the simple equi-width/equi-depth binning that is more common, whereas MergeSD [95] involves discretization with overlapping intervals, by adjusting interval bounds with a bound table. Moreover, subgroup discovery techniques have also been applied in the context of spatial databases [120, 155]. Additionally, the objective of our subgroup discovery is similar to *bump hunting* [72], which also aims to identify subsets that are considerably greater or smaller than the average of the general population. Mehta *et al.* [158] propose a correlation preserving discretization method by taking into account the inherent correlations among multiple variable, and this method can be potentially used to optimize
the binning process in our algorithm. However, no existing work is designed for processing large-scale array-based scientific datasets. Compared with all the existing subgroup discovery algorithms, a key difference in our approach is that we directly operate on the compact bitmap indices, instead of the raw datasets, and utilize fast bitwise operations on them, allowing processing of larger datasets.

Reducing the relevancy or redundancy of the output is also an important issue in subgroup discovery. As a modified version of the classical subgroup discovery, relevant subgroup discovery is proposed to eliminate irrelevant subgroups, based on the theory of relevancy [131, 132]. Relevant subgroup discovery algorithms are devised mainly based on either optimistic estimates or closed sets. Most algorithms [131, 136] based on optimistic estimates can lead to a dramatic reduction of the search space as well as the execution time, but it may not be able to guarantee correct results [80]. Closed-sets-based algorithms [74, 80] require quadratic time complexity, which is clearly not applicable to large scientific datasets. Additionally, Chen et al. [54] applied a sequential coverage approach where the formal relevancy criteria is not followed. Other non-redundant subgroup discovery algorithms [36, 215] proposed different redundancy criteria to follow. However, to the best of our knowledge, none of these algorithms is designed for the subgroup discovery over numeric array data. In the future, our method can be combined with a separate post-processing step of relevancy check.

Apart from subgroup discovery, other important descriptive data mining tasks that involve various differential analysis include contrast set mining [33], emerging pattern mining [67], differential rule mining [142], and anomaly identification [83]. Again, the distinctive aspect of our algorithm is efficient processing of large-scale array data.
5.7 Summary

This work has presented a novel algorithm for subgroup discovery over array-based datasets. This algorithm has extended the subgroup description to processing of array data, and is capable of effectively processing numeric attributes. Moreover, our algorithm directly operates on the compact bitmap indices instead of the raw datasets, and utilizes fast bitwise operations on them, allowing processing of larger datasets. We have extensively evaluated our algorithm by using multiple real-life datasets, and compared it with a popular subgroup discovery algorithm – SD-Map*. We demonstrate both high efficiency and effectiveness of our algorithm.
Chapter 6: SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats

In practice, scientific data analysis can often be more complex than subsetting and aggregations, and MapReduce can be adopted for simplifying the implementations. However, it often requires reloading scientific data in another file system or format (for offline processing), which can be often prohibitively expensive. In this chapter, we present a framework that can facilitate ad-hoc analysis in many scientific applications, by allowing scientific data in different formats to be processed with a MapReduce-like API.

6.1 Background

This section provides background information on the MapReduce model and the MATE system, as well as the popular scientific data formats, including NetCDF and HDF5.

6.1.1 MapReduce and MATE

MapReduce [63] was proposed by Google for scalable application development for data-centers. With a simple interface of two functions, map and reduce, this model has a great suitability for the parallel implementations of a variety of applications, including search engine support and machine learning [57], [75]. The map function takes a set of input instances and generates a set of corresponding intermediate output (key, value) pairs.
The MapReduce library groups together all of the intermediate values associated with the same key and shuffles them to the reduce function. The reduce function, also written by the users, accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values.

In recent years, our research group also designed a system we refer to as MapReduce with AlternaTE API (MATE) [109], [108]. The distinctive characteristic of MATE is that it enables the users to explicitly declare a reduction object, and uses this to overcome a key inefficiency in MapReduce, which is of storing and shuffling a large number of key-value pairs.

6.1.2 Scientific Data Formats

For efficient storage and I/O on large-scale scientific data, a number of scientific data formats have been proposed, including CDF (Common Data Format), FITS (Flexible Image Transport System), GRIB (GRid In Binary), HDF and HDF5 (Hierarchical Data Format), HDF-EOS (Earth Observing System extensions to HDF), and NetCDF (Network Common Data Format). Among these scientific data formats, NetCDF and HDF5 seem to be the most popular ones at the current time.

NetCDF [8] comprises software libraries and a self-describing, portable data format, with the goal of supporting creation, access, and sharing of scientific data. It is commonly used in climatology, meteorology, and GIS applications. A NetCDF dataset comprises of dimensions, variables, and attributes, which all have both a name and an ID number by which they are identified.

HDF5 [3] is another widely used data model and file format for storing and managing data. It has been introduced in Section 2.1.
6.2 System Design

In this section, we discuss the design and implementation of the SciMATE system. After giving a brief overview of the system, we describe an API for integrating a new data format into the system.

![Execution Overview of SciMATE](image)

Figure 6.1: Execution Overview of SciMATE

6.2.1 System Overview

SciMATE’s execution is closely based on the precursor system, MATE, which has been described in our earlier publications [108, 109]. The key new feature is the scientific data processing module, which is responsible for partitioning input scientific datasets, loading partitioned data into memory blocks, and possible data restructuring while loading non-contiguous data.
Figure 6.1 gives an overview of the execution flow of a typical application using SciMATE in a distributed environment. First, given an input scientific dataset, the scientific data processing module will invoke a specific partitioning function, which corresponds to the data format involved. This function divides the input into a number of partitions. The partitioning function needs to be implemented for each specific data format, and it interacts with the corresponding library to help retrieve dataset information such as dataset dimensionality/rank, the length of each dimension, the number of units, and the total size of dataset. Additionally, while performing partitioning, data locality is considered so that each node can load most of the input partitioned data from its own local disk.

Figure 6.2 shows additional details of the scientific data processing module. Since NetCDF and HDF5 are two of the most popular scientific data formats, we have implemented data processing support for these two formats in the current version of SciMATE. Although SciMATE is able to process large NetCDF/HDF5 datasets, all the nitty-gritty NetCDF/HDF5 APIs details are abstracted within the data processing module and transparent to application developers. Thus, the users are allowed to develop applications on the on-disk scientific datasets as if they are ordinary arrays located in the memory.

The most important component of the scientific data processing module is a block loader, which is scheduled by the runtime to load partitioned data into memory blocks. As Figure 6.2 illustrates, the block loader is connected with both a data format selector and an access strategy selector so that the module can tackle two options specified by the users: the data format of input datasets and access strategy. The current data format selector provides three system-defined options: NetCDF, HDF5 and flat-file, i.e. binary data. Once
the users select one of the three options, the runtime will dynamically bind a corresponding adapter through the data format selector. The launched adapter translates calls to the generic interface of block loader, into calls to the original scientific file format libraries (NetCDF/HDF5) or the standard file I/O library. On the other hand, the current access strategy selector provides two options: full read and partial read, which will be introduced later in Section 6.3.
6.2.2 API for Integrating a New Data Format

Besides supporting NetCDF, HDF5, and flat-files, our system allows for enhanced reusability and extensibility by allowing a new data format adapter to be easily added. Thus, the same data analysis application can be executed even on a dataset which has a different data format. The three system-defined data format adapters can be viewed as instances of a generic adapter interface. Given a new data format, a third-party adapter can be developed by implementing the block loading prototypes that interact with the corresponding library.

The API set for scientific data processing is summarized in Table 6.1. To integrate a new data format, a customized partition function is required, which computes both the starting location and the size of each partition. The same function then sends these results to assigned computing nodes. While some critical information like the number of nodes can be retrieved using certain MPI functions, other important information is obtained via data-format specific functions. These functions include get\_dimensionality, get\_dataset\_size, get\_dimlens and get\_unit\_num, which help obtain dataset dimensionality/rank, dataset size, the length of each dimension and the number of units, respectively.

Moreover, the API set also allows the use of data-format specific libraries to implement different access strategies, including full read and partial read. The full\_read function reads complete data records. The partial\_read function specifies how to load a single contiguous subregion. Additionally, partial read can also be implemented via certain access patterns: the partial\_read\_by\_block function is for the strided pattern which will be discussed later in Section 6.3, the partial\_read\_by\_column function allows the users to select required columns that are mapped to certain dimensions, regardless of
column contiguity, and finally, the \textit{partial-read-by-list} function serves to retrieve a list of discrete elements in the input datasets.

To show how this API is used for different data formats, we take the \textit{full-read} function as an example, and demonstrate how NetCDF and HDF5 calls are implemented. For double precision data, the steps involved are: 1) open the file and retrieve the file id by file name; 2) open the dataset and retrieve the dataset id by file id and dataset name; 3) check if the data type is double; 4) retrieve the dataset rank by dataset id; 5) retrieve the length of each dimension; 6) define the data space and memory space with the same rank and the same length of each dimension; 7) load the entire dataset into read buffer; and 8) release the file id and other descriptors if necessary. The NetCDF and HDF5 calls implementing these steps are listed in Table 6.2. Note that because of a slight difference between NetCDF and HDF5 programming models, steps 4-6 can be skipped for the NetCDF adapter. Based on this, we expect that an adapter for any other array-based scientific data format can be implemented in a similar fashion.

6.3 System Optimization

In this section, we discuss optimizations that have been implemented in SciMATE system to help improve the performance of reading data.

6.3.1 Data Access Strategies and Patterns

In our framework, we have considered two different types of access strategies: \textit{Full Read} and \textit{Partial Read}. In terms of the partial read strategy, it can be implemented by different access patterns.
**Full Read:** Full read involves loading the entire original dataset for processing. It is the simplest case during the block loading, where all the block data is required for computations. Therefore, no subsetting or merge operations are required.

**Partial Read:** Partial read refers to loading non-contiguous subsets of input dataset. The need for such a read operation can arise because of many reasons. First, not all data records may be analyzed by a particular application. Similarly, not all dimensions may be needed. There is also a possibility that the data elements needed to construct a single record may not be stored contiguously. The key issue with a partial read is that a simple read of all required data elements may lead to poor performance, due to frequent non-contiguous and/or small I/O requests. Such non-contiguous and small reads may be replaced by contiguous or large reads. SciMATE performs this replacement after input data is partitioned and partitions are assigned to each node, but before each partition is loaded into a memory block. Note that all the details of such restructuring are transparent to the users.

To perform partial read, the users only need to specify the layout of a high-level view of the required data subsets, which usually can be described by a certain high-level access pattern. In other words, partial read is performed based on a specific high-level access pattern explicitly determined by the users beforehand. The different access patterns we support are as follows:

- **Strided Pattern:** If the required non-contiguous regions have a regular pattern, a *strided pattern* can be specified, via starting dimension/record, the access stride vector within each dimension/record subset, the number of dimension/record subsets, as well as the number of dimensions/records within each subset. All the details related to the low-level data layout are mapped from the high-level view of datasets related to record/dimension and hence hidden from the users. The strided
patterns can be categorized into a *simple-strided pattern* and a *nested-strided pattern*. *Simple-strided pattern* refers to a series of I/O requests where each request is for a segment that contains the same number of elements, and each segment starting location is incremented by the same factor in a single dimension/record growth direction. *Nested-strided pattern* is more complex. It is composed of strided segments separated by regular strides, i.e., a nested-strided pattern is defined by two or more strides in different dimensions and/or record growth direction, instead of one stride as in the simple-strided access pattern.

- **Column Pattern**: If the required data consists of a set of arbitrary columns, a *column pattern* can be specified, using that set of columns. This is because in many HPC data analysis applications, only a subset of dataset dimensions may be needed for a certain type of data analysis, and SciMATE can map each dimension to a single column during computation.

- **Discrete Point Pattern**: If the required regions cannot be described by a regular pattern, a *discrete point pattern* can be specified, using a vector of element locations and the number of required elements.

### 6.3.2 Optimizing Different Types of Accesses

The following optimizations are performed in our framework. For strided patterns, appropriate strided reads are invoked from NetCDF/HDF5 libraries. We appropriately map the high-level parameters such as dimension/record index, to the low-level layout like column/row index. The optimization for column pattern is described in details below. For discrete point pattern, no optimization has been implemented currently.
When only a subset of columns are needed, the first question is whether to use full read or partial read. It turns out that when a contiguous subset of columns are needed, partial read can be beneficial. However, when a non-contiguous subset is needed, the performance can depend upon the particular layout being used.

It also turns out that, NetCDF has a better tolerance for column non-contiguity than HDF5. If the loaded columns are composed of multiple discrete regions, the performance of HDF5 partial read is likely to get worse. In comparison, the loading time of any NetCDF partial read, including reading a number of discrete column regions, will take at most twice as long as the full read (while reading the same amount of data). Thus, if a small subset of data is to be loaded, partial read is advantageous for NetCDF.

The second issue is to optimize partial read with column accesses. We choose from two approaches for reading a set of columns, which are fixed-size column read and contiguous column read.

**Fixed-size Column Read:** Fixed-size column read is a naive strategy. This strategy allows only a fixed number of columns to be read at a time, and the system default value of this fixed number is 1. Thus, the data loading amount for each time will be the same, resulting in a balanced workload distribution.

**Contiguous Column Read:** Contiguous column read strategy reads a contiguous column set at a time. Instead of a fixed number, the number of loaded columns in this strategy is determined by column contiguity at runtime. Contiguous column read takes advantage of the column contiguity so that the number of column reads equals the number of contiguous regions. This strategy can minimize the number of reads and significantly reduce the overhead resulting from frequent small I/O requests.
The choice of strategy in our system is guided by the observations from a number of experiments we performed. Here, we use k-means clustering to illustrate the influence of column contiguity in the performance of column read. We executed k-means application on 8 GB 10-dimensional NetCDF and HDF5 datasets. We considered several different possible application scenarios, where different numbers of contiguous/discrete columns were read from the datasets. The results are shown in Figure 6.3. In the horizontal axis, different scenarios are shown. Particularly, “1” means only 1 single column was read, “2C” indicates that 2 contiguous columns were required, “3C” stands for 3 contiguous columns, “2D” denotes 2 discrete columns to be read, “2C+2C” represents 2 discrete column sets, where each consists of 2 contiguous columns, and finally, “3D” refers to 3 discrete columns.

First, we observed that the data loading performance could be categorized into 3 levels: with 1, 2C, and 3C being the first level, 2D and 2C+2C being the second level, and 3D being the third level. To simplify the analysis, we ignored the influence brought by the number of loaded columns, especially between “1” and “2C” in the experiments on the HDF5 datasets. Although the numbers of loaded columns differed in the first 3 scenarios,
their data loading time were close, because the column reads were performed only once. Similarly, it took 2 column reads to complete data loading in “2D” and “2C+2C”, so the data loading time was longer than the first three, while being shorter than “3D”. Overall, we can conclude that the performance primarily depends upon the number of separate reads, and not on the volume of data read.

Second, the results showed NetCDF had a better column non-contiguity tolerance than HDF5. For instance, sometimes reading 3 discrete HDF5 columns could be quite expensive, even over 3 times slower than reading a single column, while reading discrete NetCDF columns caused a relatively small overhead. Additionally, we also observed that the strides among all the discrete regions also had some impact on the performance.

To summarize, we use the observations made from our experiments to guide the selection of appropriate column access approach, i.e., contiguous column read. Note, however, we expect a future development to improve the attractiveness of fixed-size column read. Once NetCDF and/or HDF5 libraries allow parallel data accesses from multiple threads in a single process, the performance of fixed-size column read may improve, since a better load balance can be achieved.

6.4 Experimental Results

In this section, we evaluate the functionality and scalability of SciMATE system on a cluster of multi-core machines. We show the performance achieved with three different data formats: NetCDF, HDF5, and flat-file. Last, we present the results from optimization evaluation, including comparing the performance of partial read against full read, as well as the performance of fixed-size column read against contiguous column read.
6.4.1 Applications and Environmental Setup

We used three popular data analysis algorithms, which are k-means clustering (k-means), principal components analysis (PCA), and k-nearest neighbor search (kNN). K-means is one of the most popular data mining algorithms. PCA is a popular dimensionality reduction method that was developed by Pearson in 1901. Its goal is to compute the mean vector and the covariance matrix for a set of data points that are represented by a matrix. KNN is a type of instance-based learning method [20] for classifying objects based on closest training examples in the feature space, which has been successfully applied in diverse domains like protein function prediction and image de-noising [178].

The datasets and the application parameters we used are as follows. The sizes of all the datasets for the evaluation in Section 6.4.2 and 6.4.3 were 16 GB and 8 GB, respectively. With k-means, we used 10-dimensional points, and the number of clusters, $k$, was set to be 100. For both PCA and kNN, the points were 1000-dimensional. In kNN, the number of clusters, $k$, was set to be 100, and the number of nearest neighbors, was set to be 30.

Our experiments were conducted on a cluster of multi-core machines. The system uses AMD Opteron(TM) Processor 8218 with 4 dual-core CPUs (8 cores in all). The clock frequency of each core is 2.6 GHz and the system has an 8 GB main memory. We have used up to 128 cores (16 nodes) for our study. The NetCDF version is 4.1.1, and the HDF5 version is 1.8.7.

6.4.2 Functionality Evaluation

In this section, we evaluate the data processing performance and the data loading performance of SciMATE, by executing the three data analysis algorithms in a way that
involved full reads. For each of the applications, we use 16 GB datasets that are stored in three different data formats: NetCDF, HDF5 and flat-file (FLAT).

**Data Processing Times:** Figures 6.4 through 6.6 show the comparison results for three applications as we scale the number of threads used on a single node. Figures 6.7 through 6.9 show the results as we scale the number of nodes used with 8 threads on each node. We can see that the data processing performance on these three different data formats is quite similar, and moreover, scales with both number of threads and nodes. This shows that
SciMATE is capable of supporting different data formats, and scaling the performance of data analysis applications.

**Data Loading Times:** Figures 6.10 and 6.11 show the data loading times for k-means and PCA, respectively. The results indicate that loading NetCDF/HDF5 datasets is faster than loading flat-file datasets, demonstrating up to 62.6% throughput improvement. We believe this is because the highly structured nature of scientific data can facilitate the parallel I/O in distributed environments. Compared with the unstructured flat-file datasets, the data
layout of NetCDF/HDF5 datasets is described in the header information, so certain unnecessary scan for each partition or split can be avoided. Moreover, the results also show that loading NetCDF datasets is slightly faster than loading HDF5 datasets. This is related to the performance of the libraries, and mostly unrelated to SciMATE. Specifically, NetCDF has a linear data layout, i.e. the data arrays are either stored in contiguous space (with a predefined order) or interleaved with a regular pattern. Such regularity in the layout helps improve MPI-IO performance. In comparison, HDF5 chooses hierarchical
data layout, which is more flexible but leads to more irregular accesses. Particularly, HDF5 uses dataspace and hyperslabs to map and transfer data between the memory and the file system. This causes buffer packing/unpacking in a recursive way, and results in performance loss to a certain degree. Another factor is that NetCDF has a smaller header I/O overhead than HDF5. In NetCDF, only one header is stored, while in HDF5 header metadata is dispersed in separate header blocks for each object.
6.4.3 Optimization Experiments

In this section, we evaluate two of the optimizations proposed in Section 6.3, i.e. use of partial reads, and use of contiguous column reads. Our third optimization, which is the use of APIs for strided patterns, is not evaluated here, since it is directly supported in NetCDF and HDF5.

Comparing Partial Read with Full Read: To evaluate the benefits of partial read, we designed the following experiment. We executed PCA application with 4 nodes and 1 thread per node, on both NetCDF and HDF5 datasets (8 GB each). We considered the following cases: 200, 400, 600, and 800 contiguous columns were required by the data analysis applications, when the dataset had a total of 1000 columns. For these four cases, we compared the performance of partial read against full read.

![Figure 6.12: PCA: Comparison between Partial Read and Full Read](image)

The results reported in Figure 6.12 show that if the number of contiguous columns to be loaded is no more than half of the total number, partial read can outperform full read (shown as right-most charts, i.e. when all 1000 columns are read). Otherwise, full read
may be a better option. This is because that both NetCDF and HDF5 datasets have row-major ordering, and therefore, the data corresponding to a subset of contiguous columns actually is not stored contiguously in the dataset files. Thus, a partial read involves multiple I/O requests for loading segments from disk files to the read buffer. Therefore, unless the volume of data to be read is quite small, the overhead of separate I/O requests slows down the read operation.

Moreover, compared with the full read, the partial read only loads required data into memory, leading to a better data locality and lower memory consumption. Therefore, sometimes partial read can also accelerate the later reduction process because of the better data locality.

Comparing Fixed-Size Column Read with Contiguous Column Read: We compare the performance of fixed-size column read against that of contiguous column read, by executing kNN application with 2 threads on one node, on both NetCDF and HDF5 8 GB datasets. In the experiments we designed, the SciMATE reads three contiguous column sets with the same number of columns in the 1000-dimensional datasets. Each contiguous column set contains 100, 200, and 300 columns respectively in three different comparison experiments.

Figure 6.13 and Figure 6.14 show that contiguous column read outperforms fixed-size column read, even the number of contiguous regions is quite limited. Contiguous column read can gain a speedup of about 3.0 on NetCDF datasets, and a speedup of about 30.0 on HDF5 datasets.
6.5 Related Work

The topics of data-intensive computing and MapReduce have attracted a lot of attention in the past few years. Because of the large volume of work in this area, we will restrict ourselves to work specific to scientific data processing.

At Indiana, CGL-MapReduce [70] has been developed as an extension to MapReduce. It uses streaming for all the communications, and thus improves the performance to some extent. More recently, a runtime system, Twister [69], has been developed to optimize iterative applications. Somewhat similar optimizations are already implemented in MATE.

Integrating MapReduce systems with scientific data has been a topic of much interest recently [42, 89, 107, 146, 225, 229, 260], as also summarized by Buck et al. [42]. The Kepler+Hadoop project [225], as the name suggests, combines MapReduce processing with Kepler, a scientific workflow platform. Its limitation is that it cannot support processing of data in different scientific formats. In another system [260] NetCDF processing using Hadoop is allowed, but the data has to be converted into text, causing high overheads.
Figure 6.14: KNN: Comparison between Fixed-size Column Read and Contiguous Column Read on HDF5 Datasets

SciHadoop [42] integrates Hadoop with NetCDF library support to allow processing of NetCDF data with MapReduce API. We expect to compare the performance of our system with SciHadoop in the future. We will like to note, however, that our underlying system, MATE, has been shown to outperform Hadoop by a large factor in previous studies [108, 109]. Moreover, their current implementation is restricted to handling NetCDF. The authors plan on extending their work to support HDF5 (and list several challenges in accomplishing this), whereas our current implementation already handles HDF5. MARP [185] is another MapReduce-based framework to support HPC analytical applications, which optimizes for certain access patterns. However, it cannot directly operate on scientific data formats.

Research has been conducted towards processing and querying data in formats like NetCDF, with other APIs (i.e. besides MapReduce) as well. The NetCDF Operator (NCO) library has been extended to support parallel processing of NetCDF files [223]. However, a specialized API has to be used which is format-specific. To enable querying of large-scale scientific data, FastQuery framework has been developed [107], but their focus is querying (subsetting) of data rather than processing of data.
6.6 Summary

With growing importance of analysis of large-scale data in various scientific areas, two trends have become clear. First, data collected from instruments and simulations is going to be stored in certain specialized data formats, depending upon the specific domain (e.g., climate data is almost always stored in NetCDF). Second, with the rapid increase in dataset sizes, it is not feasible to reformat the data, or load it in a different file system or a database. Thus, we need an ability to develop scalable data analysis applications that can work with the popular scientific data formats.

This work has presented such a framework. Our system, SciMATE, has an API which allows it to be customized for any new scientific data format. We have also developed several optimizations, particularly driven by the observations that most scientific datasets have a large number of attributes, but only a subset of those are needed by any particular application. We have created three instances of the system, for NetCDF, HDF5, and flat-files, and have evaluated them using three popular data analysis tasks. Our results show that each of the three instances of the system scales well on a multi-core cluster.
Table 6.1: Descriptions of the functions in the Scientific Data Processing API

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int partition (char*, char*, int, size_t**, size_t**)</td>
<td>Partitions the input dataset for each computing node by computing the starting location and size of each partition. The arguments are file name, dataset name, number of nodes, starting location of each partition, and the size of each partition.</td>
</tr>
<tr>
<td>int get_dimensionality (char*, char*)</td>
<td>Retrieves dataset dimensionality/rank. The arguments are file name and dataset name.</td>
</tr>
<tr>
<td>size_t get_dataset_size (char* char*)</td>
<td>Retrieves dataset size. The arguments are file name and dataset name.</td>
</tr>
<tr>
<td>int get_dimlens (char*, char*, size_t*)</td>
<td>Retrieves the length of each dataset dimension. The arguments are file name, dataset name, and dataset dimensionality.</td>
</tr>
<tr>
<td>unsigned long get_unit_num (char*, char*)</td>
<td>Retrieves the number of units in the dataset. The arguments are file name and dataset name.</td>
</tr>
<tr>
<td>int full_read (char*, char*, void*)</td>
<td>Reads the entire input dataset. The arguments are file name, dataset name, and read buffer.</td>
</tr>
<tr>
<td>int partial_read (char*, char*, void*, size_t*, size_t*)</td>
<td>Reads a contiguous selection of the input dataset. The arguments are file name, dataset name, read buffer, starting location, and buffer size.</td>
</tr>
<tr>
<td>int partial_read_by_block (char*, char*, void*, size_t*, size_t*, size_t*)</td>
<td>Reads a non-contiguous selection of the input dataset by regular block. The arguments are file name, dataset name, read buffer, block starting location, stride vector among blocks, block count, and block size.</td>
</tr>
<tr>
<td>int partial_read_by_column (char*, char*, void*, size_t*, size_t*, int*, int)</td>
<td>Reads a non-contiguous selection of the input dataset by column. The arguments are file name, dataset name, read buffer, starting location, buffer size, the indexes of loaded columns, and number of loaded columns.</td>
</tr>
<tr>
<td>int partial_read_by_list (char*, char*, void*, size_t**, int)</td>
<td>Reads a non-contiguous selection of the input dataset by specifying a list of discrete elements. The arguments are file name, dataset name, read buffer, a vector of element locations, and element count.</td>
</tr>
</tbody>
</table>
Table 6.2: Invoked Calls in NetCDF/HDF5 Lib for full read Implementation

<table>
<thead>
<tr>
<th>Step#</th>
<th>Invoked NetCDF Call</th>
<th>Invoked HDF5 Call(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>nc_open</code></td>
<td><code>H5Fopen</code></td>
</tr>
<tr>
<td>2</td>
<td><code>nc_inq_varid</code></td>
<td><code>H5Dopen</code></td>
</tr>
<tr>
<td>3</td>
<td><code>nc_inq_vartype</code></td>
<td><code>H5Dget_type</code></td>
</tr>
<tr>
<td>4</td>
<td><code>(nc_inq_varndims)</code></td>
<td><code>H5Sget_simple_extent_ndims</code></td>
</tr>
<tr>
<td>5</td>
<td><code>(nc_inq_dimlen)</code></td>
<td><code>H5Sget_simple_extent_dims</code></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td><code>H5Dget_space, H5Screate_simple</code></td>
</tr>
<tr>
<td>7</td>
<td><code>nc_get_var_double</code></td>
<td><code>H5Dread</code></td>
</tr>
<tr>
<td>8</td>
<td><code>nc_close</code></td>
<td><code>H5Dclose, H5Tclose, H5Sclose, H5Fclose</code></td>
</tr>
</tbody>
</table>
Chapter 7: Smart: A MapReduce-Like Framework for In-Situ Scientific Analytics

In the previous chapter, we focused on processing disk-resident scientific datasets, which involves handling multiple scientific data formats. In this chapter, we discuss adapting MapReduce for in-situ analytics, which processes scientific data from (distributed) memory generated by simulation program.

7.1 MapReduce and In-Situ Analytics

In this section, we first argue why the MapReduce API is suitable for in-situ analytics, and then focus on the challenges in applying the MapReduce idea for efficient in-situ scientific analytics.

7.1.1 Opportunity and Feasibility

MapReduce [63] has been one of most widely adopted programming model for developing data analytics implementations – though it is perhaps not as widely accepted for science areas as it is for commercial areas. MapReduce API not only simplifies parallelization, but the framework implementation handles much of scheduling, task management, and data movement. However, none of its current implementations is directly suitable for in-situ scientific analytics.
We posit that MapReduce API is indeed suitable for a large set of analytics tasks one might perform in-situ on a scientific simulation. Now we give some specific use cases of in-situ analytics reported in various studies, and how these cases can potentially fit the MapReduce paradigm: 1) visualization algorithms [114, 250], where most steps are embarrassingly parallel and others involve reductions; 2) statistical analytics [266] and similarity analytics [205] where statistics like averages, histogram, and mutual information need to be calculated – steps that are very well suited for MapReduce [42], or even higher-level frameworks built on top of MapReduce, e.g., Pig [168] and Hive [212]; and 3) feature analytics [129] and clustering analytics [257], which have been efficiently implemented in MapReduce (though in an offline fashion), e.g., logistic regression and k-means clustering through Spark [252].

Besides the match between the target applications and the choice of programming model, another important issue tends to be that of programmer’s expertise. In this respect, we argue that as MapReduce has been gaining great popularity in recent years, many scientists are now well-trained for writing MapReduce-style code for scientific analytics [42, 146, 154, 185, 213, 225, 237, 260]. Therefore, an in-situ MapReduce-like framework can be a promising approach to improve both productivity and maintainability of scientific analytics.

### 7.1.2 Challenges

We next discuss the challenges of bridging the gap between in-situ scientific analytics and MapReduce, which we summarize as *four mismatches*.
Data Loading Mismatch

As the name ‘in-situ’ implies, the downstream analytics program is required to take the input directly from (distributed) memory rather than from a file system, as soon as the simulated data becomes available. However, existing MapReduce implementations are not designed for such a scenario. To further elaborate on this data loading mismatch, we first categorize all the MapReduce implementations into four types according to the data loading mechanism.

1. **Loading Data from Distributed File Systems**: A prominent example is Hadoop, as well as its variants like M3R [192] and SciHadoop [42], which load data from Hadoop Distributed File System (HDFS). Moreover, Hadoop actually mimics Google’s MapReduce [63], which loads data from the Google File System (GFS). Additionally, Disco [2], a MapReduce implementation in Erlang, loads data from Disco Distributed File System (DDFS).

2. **Loading Data from Shared and/or Local File Systems**: Systems like MARIANE [71] and CGL-MapReduce [70] have adapted MapReduce to scientific analytics environment by loading data from a shared file system. Moreover, other MPI-based implementations like MapReduce-MPI [173] and MRO-MPI [160] can load data from shared file system and/or local disk.

3. **Loading Data from Memory**: Pthread-based MapReduce prototypes like Phoenix [179], Phoenix++ [208], and MATE [109], can load data from memory. However, these prototypes are restricted to shared-memory environment, and hence currently they are not available for distributed computing.
4. **Loading Data from a Data Stream:** Though MapReduce was originally designed for batch processing, systems like HOP [59], M3 [23], and iMR [147] have focused on stream processing.

Clearly, the first two categories, where data is loaded from file systems cannot support in-situ analytics. Similarly, the third class lacks native support for global synchronization required in a distributed environment. The fourth group seems more suitable, as one might consider the possibility of wrapping simulation output as a data stream. However, this approach is still problematic, as simulation outputs are sporadic, and more critically, stream systems do not normally support iterative processing [230].

However, among all the MapReduce implementations we have examined, we find Spark [252] as an exception here. Its input data layout is defined as Resilient Distributed Dataset (RDD) [251], which can be derived from all the above data source options. However, Spark still has functionality and performance limitations, which will be demonstrated through a series of experiments we report in Section 7.4.

**Programming View Mismatch**

Simulations are usually implemented in MPI (or a PGAS language) that is suitable for distributed memory environments (possibly in conjunction with a shared memory API like OpenMP/OpenCL). With these low-level parallel programming libraries, the programmers explicitly express parallelism in a *parallel programming view*. On the other hand, the simplified interface of MapReduce presents a *sequential programming view*, which hides all the parallelization complexities. Thus, traditional MapReduce implementations cannot explicitly take partitioned simulation output as the input, or launch the execution of
analytics from an SPMD region. Without any change at the downstream MapReduce side, this mismatch cannot be addressed in a realistic way.

An elegant option will be to develop a new MapReduce implementation, which can present a \textit{hybrid programming view}. Particularly, at the beginning, a parallel programming view should be presented, to allow the programmers to be aware of all the partitions during the parallel execution. After the partitioned data are input, a sequential programming view should follow, so parallelism details are hidden.

\textbf{Memory Constraint Mismatch}

As simulation programs normally execute with problem sizes that require all or almost all available main memory on each node, the \textit{in-situ} analytics program can only take a very small amount of memory. However, nearly all existing MapReduce implementations are memory-intensive. This is primarily because in the mapping phase, each element results in intermediate data in the form of one or more key-value pairs, which can have an even greater size than the original input data. Note that although a combiner function at the mapper side can significantly reduce the size of intermediate data in the shuffling phase, it will not help reduce the peak memory consumption in the mapping phase. The memory constraint mismatch cannot be addressed unless we redesign the MapReduce execution flow – particularly, we need to avoid the intermediate key-value pairs.

\textbf{Programming Language Mismatch}

The last mismatch is from the programming languages that are used to implement simulation and analytics programs with MapReduce. Almost all of the HPC simulations in use are written in Fortran or C/C++, whereas both Hadoop and Spark, which are the most widely adopted MapReduce implementations (though Spark also provides other
functionality), cannot natively support Fortran or C/C++. Although this mismatch can be alleviated by using alternate C/C++ based MapReduce implementations [71, 160, 173, 179], these systems are not widely adopted.

7.2 System Design

In this section, we discuss the design and implementation of our system. Overall, Smart design addresses all the challenges we described in the last section, specifically: 1) to address the data loading mismatch, Smart supports processing data from memory generated by the simulation program – and in one of the in-situ modes (time sharing), does so without requiring an extra data copy; 2) to address the programming view mismatch, Smart offers a hybrid programming view – this exposes the data partitions to the analytics while launching the data processing, and can still hide parallelism during the execution; 3) to address the memory constraint mismatch, Smart achieves high memory efficiency by modifying the original MapReduce API (while still keeping programming effort very low), and more specifically, avoids the large number of key value pairs or the need for shuffling; and 4) to address the programming language mismatch, Smart is implemented in C++11, in conjunction with OpenMP and MPI.

7.2.1 System Overview

Figure 7.1 gives an overview of the execution flow of a typical application using Smart in a distributed environment. First, given a simulation program, each compute node generates a data partition at each time-step. Instead of the data being output to the disk, the memory resident data partitions are immediately taken as the input by the downstream Smart analytics job(s). Since the data partitions are generated from the SPMD region of the simulation program, the Smart jobs are also launched from the same code region.
Unlike most distributed data processing systems, Smart can directly expose these partitions to the subsequent processing, rather than involve any explicit data partitioning among the compute nodes.

Next, the Smart runtime scheduler processes partitioned data block by block. For each data block, the Smart runtime scheduler equally divides it into multiple \textit{splits}, where each split is assigned to a thread for processing. Additionally, Smart binds each thread to a specific CPU core to maximize the performance.

In processing elements within a split, there are two key operations, \textit{reduction} and \textit{combination}, which are carried out on two core map structures, \textit{reduction map} and \textit{combination map}, respectively. To support these operations, the programmers need to define a \textit{reduction object}, which represents the data structure of value in the key-value pairs of the two maps. This data structure maintains the accumulated (or reduced) value across all key-value pairs that have the same key. In the reduction operation, a key is first
generated for each element in the split. With this key, the runtime next locates a reduction object in the reduction map, and then the corresponding element is accumulated on this reduction object. In the combination process, all the reduction maps are combined into a single combination map locally, and then all the combination maps on each node are further merged on the master node.

The above execution flow modifies the original MapReduce processing, but it is also the key to the high memory efficiency of Smart. Specifically, explicit declaration of the reduction object eliminates the shuffling phase of MapReduce. We will show the specifics of API through examples later in this section, but the key point is that besides the declaration of reduction object, the programming effort is not any higher than the one involved in using the original MapReduce API.

### 7.2.2 Two In-Situ Modes

To maximize the performance in different scenarios, our system provides two in-situ modes – *time sharing* and *space sharing*. More specifically, we observe that: 1) for certain simulations and/or architectures, memory can be a significant constraint, and we must avoid unnecessary data copying, and 2) in many-core architectures, simulations may not be able to use all available cores effectively, and dedicating a certain number of cores for data analytics can be feasible and desirable. The two situations described above (which may not necessarily be exclusive), lead to the *time sharing* and *space sharing* modes.

**Time Sharing Mode:** Time sharing mode aims to minimize the memory consumption of analytics, by avoiding extra data copy of simulation output. Note that although the memory copy itself is likely not an expensive operation, it can increase the total memory requirements, which can lead to performance degradation in certain cases.
As shown in Figure 7.2(a), to avoid an extra data copy, Smart sets a read pointer on the memory space corresponding to the output from a particular time-step (when the data is ready). Thus, this data can be now shared by both simulation and analytics programs. However, because this memory space is subject to being overwritten by the simulation program, the analytics logic must execute before the simulation resumes. As a result, in this mode simulation and analytics run in turns, and each makes full use of all the cores of each node (and hence the name time-sharing).

**Space Sharing Mode:** Consider a cluster where every node is an Intel Xeon Phi. Since each coprocessor has a much larger number of cores than the CPU, a simulation program written for a standard multi-core cluster is unlikely to use all cores of the Xeon Phi effectively. In this case, instead of stopping the progress of simulation periodically and performing the analytics, one can easily dedicate a certain number of the available cores for the analytics.
As shown in Figure 7.2(b), Smart maintains a circular buffer internally, in which each cell can allocate memory on demand and be used for caching the output from a time-step. In this mode, one can view simulation program and Smart as the producer and the consumer, respectively. Once a time-step’s output is generated, if the circular buffer is not full, then this data can be fed to the Smart middleware by copying it to an empty cell. Otherwise, simulation program will be blocked until a cell in circular buffer becomes available.

7.2.3 Runtime System

Launching Smart Runtime

Smart is written in C++11, using OpenMP and MPI to achieve parallelism and to also be compatible with a scientific simulation environment. Thus, launching Smart does not require installing additional libraries (e.g., HDFS). Now we show how to launch Smart in two different in-situ analytics modes.

**Listing 7.1: Launching Smart in Time Sharing Mode**

```cpp
void simulate(Out* out, size_t out_len, const Param& p) {
   // Each process simulates an output partition of data type In and length inLen. */
   // Launch Smart after simulation in the parallel code region.
   SchedArgs args(num_threads, chunk_size, extra_data, num_iters);
   unique_ptr<Scheduler<In, Out>> smart(new DerivedScheduler<In, Out>(args));
   smart->run(partition, in_len, out, out_len);
}
```

**Launching Smart in Time Sharing Mode:** As demonstrated in Listing 7.1, to run Smart in this mode, only 3 lines (lines 4 - 6) need to be added to the simulation program. The example code shows the execution of processing a single time-step. Lines 4 and 5 construct a derived Smart scheduler by specifying the number of threads, the size of a unit data chunk (i.e., unit element), the extra data besides the input array (e.g., the initial k centroids are required in k-means clustering), and the number of iterations. Note that Smart scheduler class is defined as a template class, and hence Smart can be utilized for taking any array type as input or output, without complicating the application code. In line 6, Smart launches
analytics by taking the partitioned data as the input, and the final result will be output to the
given destination. Note that the definition of reduction object, as well as the derived Smart
scheduler class, are implemented in a separate file based on another API set, which does
not add any complexity of the original simulation code.

Listing 7.2: Launching Smart in Space Sharing Mode

```c
void simulate(Out * out, size_t out_len, const Param & p) {
  /* Initialize both simulation and Smart. */
  #pragma omp parallel num_threads(2)
  #pragma omp single
  {
    #pragma omp task // Simulation task.
    {
      omp_set_num_threads(num_sim_threads);
      for (int i = 0; i < num_steps; ++i) {
        /* Each process simulates an output partition of length in_len. */
        smart->feed(partition, in_len);
      }
    }
    #pragma omp task // Analytics task.
    for (int i = 0; i < num_steps; ++i)
      smart->run(out, out_len);
  }
}
```

Launching Smart in Space Sharing Mode: As shown in Listing 7.2, space sharing
mode requires somewhat larger amount of work than the time sharing mode, since an
extra task-level parallelism has to be deployed. Particularly, two OpenMP tasks are
created for concurrent execution. After the initialization of both simulation and Smart, one
task encapsulates the simulation code and then feeds its output to Smart (lines 6 - 13), and
the other task runs analytics (lines 14 - 16). The number of threads used for simulation is
specified within the simulation task, and the number of threads used for analytics is
specified when Smart is initialized. Note that MPI codes are hidden in both simulation
task and analytics task, and in this mode MPI functions may be called concurrently by
different threads. Thus, to avoid the potential data race, the level of thread support should
be upgraded to `MPI_THREAD_MULTIPLE` when MPI environment is initialized.

Data Processing Mechanism

194
Algorithm 8: run(const In* in, size_t in_len, Out* out, size_t out_len)

1: process_extra_data(extra_data, combination_map) {Process the extra data if needed}
2: for each iteration iter do
3:   if iter ≠ 1 then
4:     Distribute the global combination map to each local combination map
5:   end if
6: Distribute the local combination map to each reduction map
7: for each processing unit chunk ∈ in do
8:   key ← gen_key(chunk, data, combination_map)
9:   accumulate(chunk, data, reduction_map[key])
10: end for {Reduction}
11: for each (key, red_obj) ∈ reduction_map do
12:   if key exists in combination_map then
13:     merge(red_obj, combination_map[key])
14:   else
15:     move red_obj to combination_map[key]
16:   end if
17: end for {Local combination and global combination}
18: post_combine(combination_map) {Perform post-combine operations if needed}
19: end for
20: if out ≠ NULL and out_len ≠ 0 then
21:   for each (key, red_obj) ∈ combination_map do
22:     convert(red_obj, out[key])
23: end for
24: end if {Output results from the combination map}

Algorithm 8 shows the data processing mechanism in Smart. Note that all the functions
called in the pseudo-code are from our API and either have to be or can be implemented
by the programmers. Due to the space limit, the full description of API is available in a
separate extended version [230]. At the beginning, some extra input can be processed to
help initialize the combination maps as well as reduction maps (lines 1 - 6). In the reduction
phase (lines 7 - 10), as a data block is divided into multiple splits, each thread processes
a data split one unit element at a time. In line 8, a key is generated for the unit element.
Line 9 accumulates the derived data from the element into a reduction object, which can
be located in the reduction map by the generated key. The reduction object is updated in
place – no intermediate key-value pair is emitted or stored, and thus, no shuffling phase is needed during the reduction. This is a key difference between our alternate API and the conventional MapReduce paradigm.

Lines 11 - 17 show the combination phase consisting of two steps – local combination and global combination. In the local combination, the reduction maps maintained by all the threads on a process are combined into a local combination map. Particularly, the two reduction objects associated with the same key are merged into one. In the global combination, the local combination maps on all compute nodes are further combined into a global combination map that holds the global result. This global combination leverages the same merge operation used for the local combination. Line 18 can update reduction objects after the combination phase for each iteration, e.g., computing average based on sum and count. Finally, lines 20 - 23 convert all the reduction objects in the global combination map into the desired output.

A Smart Analytics Example: Histogram

Listing 7.3: Histogram as an Example Application

```cpp
Derive a reduction object:
struct Bucket : public RedObj {
  size_t count = 0;
};

Derive a system scheduler:
template<class In>
class Histogram : public Scheduler<In, size_t> {
  // Compute the bucket ID as the key.
  int gen_key(const Chunk& chunk, const In* data, const map<int, unique_ptr<RedObj>>& combination_map) const override {
    // Each chunk has a single element.
    return (data[chunk.start] - MIN)) / BUCKET_WIDTH;
  }

  // Accumulate chunk on red_obj.
  void accumulate(const Chunk& chunk, const In* data, unique_ptr<RedObj>& red_obj) override {
    if (red_obj == nullptr) red_obj.reset(new Bucket);
    red_obj->count++;
  }

  // Merge red_obj into com_obj.
  void merge(const RedObj& red_obj, unique_ptr<RedObj>& com_obj) override {
    com_obj->count += red_obj->count;
  }
};
```
We now illustrate how to develop an analytics program by using Smart. The analytics code can be reused in any (even offline) analytics mode. As an example, Listing 7.3 shows equi-width histogram construction. Two major steps are taken. To begin with, the user needs to define a derived reduction object class. Here the class *Bucket* represents a histogram bucket, consisting of a single field *count*.

In the second step, a derived system scheduler class should be defined, e.g., *Histogram* here. Note that to facilitate the manipulation on the datasets of different types, in our system the derived class can be defined as either a template class or a class specific to an input and/or output array type. For this kind of non-iterative application, the user usually only needs to implement three functions – *gen_key*, *accumulate*, and *merge*. First, the *gen_key* function computes the bucket ID based on the element value in the input data *chunk*, and the bucket ID serves as the returned key. For example, if the element value is located within the value range of the first bucket, then 0 will be returned. For simplicity, we assume that the minimum element value can be taken as priori knowledge or be retrieved by an earlier Smart analytics job. Note that in this application, since each element should be examined individually, each *chunk* as a processing unit only contains a single array element. Next in the reduction phase, the *accumulate* function accumulates *count* of the bucket that corresponds to the key returned by the *gen_key* function. Lastly, given two reduction objects, where the first one *red_obj* is from the reduction map, and the second one *com_obj* is from the combination map, the *merge* function merges *count* on *com_obj* in the combination phase.

From the above example, we can see that Smart provides a sequential programming view for application development, and the user only needs to write some sequential code based on the declared reduction object. Thus, like traditional MapReduce framework, our
system makes parallelism entirely transparent to the application code. Note that unlike a MapReduce job optimized by a combiner function, our application code does not emit any key-value pair as intermediate result. More code examples can be found in our extended technical report [230].

7.3 System Optimization for Window-Based Analytics

7.3.1 Motivation

In practice, simulation output may contain some short-term volatility or undesired fine-scale structures. In such cases, it is important to perform analytics for specific ranges of time-steps, also referred to as sliding windows. In some other cases, in-situ analytics can involve certain preprocessing steps like denoising [98] and smoothing [93, 156], which also execute on a sliding window basis. A simple example of such window-based analytics is moving average, where the average of the elements within every window snapshot is computed.

A critical challenge in the implementation of such window-based analytics is that of high memory consumption. More broadly, suppose we are calculating a derived quantity corresponding to each separate window snapshot centered by each input element (e.g., average for each distinct window of length $W$). The space complexity using Smart’s processing is $\Theta(R \times N)$, where $R$ and $N$ denote two factors – the size of each reduction object and the maximal number of reduction objects maintained by Smart, respectively. On one hand, the size of the reduction object is dependent on the specific application, and it is typically varied from $\Theta(1)$ to $\Theta(W)$. For example, since average is algebraic and can be computed by sum and count, the size of reduction object for moving average will only be $\Theta(1)$, while median is holistic and can only be computed by preserving all elements,
the size of reduction object for moving median is $\Theta(W)$. Another example is $K$-nearest neighbor smoother, where the size of reduction object is $\Theta(K)$, $1 \leq K \leq W$.

On the other hand, the total number of reduction objects in window-based analytics is equal to the input data size, since each input element corresponds to a window snapshot in a form of reduction object. Irrespective of the application, since $N$ can often be too large to meet the memory constraints of in-situ scenarios, it is very desirable to reduce the space complexity, especially by reducing the maximal number of reduction objects.

7.3.2 Optimization: Early Emission of Reduction Objects

Algorithm 9: reduce(Split split)

1. for each data chunk ∈ split do
2.     for each key k generated by chunk do
3.         Let the reduction object red_obj be reduction_map[key]
4.         accumulate(chunk, data_, red_obj)
5.         if red_obj.trigger() then
6.             convert(red_obj, out_[key])
7.             reduction_map.erase(key)
8.         end if {Optimization for early emission}
9.     end for
10. end for

We develop the optimization based on the following observation. For most elements, all the associated window snapshots are entirely covered by their respective local split of data. As a result, most reduction object values have been finalized in in the (local) reduction phase, and they will not be involved in the subsequent combination phase. By capturing this observation, we design a mechanism that can support early emission of reduction objects in the reduction phase, which is in contrast to the original design that holds all the reduction objects until the combination phase ends.
Our optimization is implemented as follows. First, we extend the reduction object class by adding a trigger function. This trigger evaluates a self-defined emission condition, and determines if the reduction object should be emitted early from the reduction map. By default, the function returns false, and hence no early emission is triggered. Second, we extend the implementation of reduce operation, which is an internal step in Smart scheduling. Lines 5 - 7 in Algorithm 9 show the extension. Once a data element is accumulated on a reduction object (line 4), the added trigger function evaluates a user-defined emission condition (line 5). If this condition is satisfied, the reduction object will be immediately converted into an output result, and then be erased from the reduction map (lines 6 and 7). With such an optimization, the maximal number of reduction objects need to be maintained is reduced from the input data size to the window size.

To support such an optimization, the user only needs to overwrite the trigger function when deriving the reduction object class. Particularly for window-based applications, the emission condition can be the number of elements that have so far contributed to a reduction object equal to the window size, which indicates the reduction object value has been finalized. It should be noted that, this optimization is not only specific to in-situ window-based analytics, but also can be broadly applied to other applications, even for offline analytics [145, 148–150, 187–189, 224, 226, 227, 231]. A simple example can be matrix multiplication, where the number of element-wise multiplications that contribute to a single output element is a fixed number.

7.4 Experimental Results

In this section, we evaluate both efficiency and scalability of our system on both multi-core and many-core clusters. First, we compare with Spark [252] – a popular
MapReduce implementation (while also providing other functionality), which has been shown to outperform Hadoop by up to 100x. Second, we compare with analytics programs written with lower-level APIs (MPI and OpenMP), to measure both the programmability and overheads of our middleware approach. Third, we evaluate the scalability of Smart as the number of nodes and cores is increased. Next, we focus on understanding and comparing performance for time sharing and space sharing modes. Lastly, we evaluate the effect of the optimization for window-based analytics, by comparing the performance with an implementation that disables the trigger mechanism.

### 7.4.1 Applications and Environmental Setup

We experimented with nine applications that represent six different classes of in-situ analytics – these classes were previously described as in-situ use cases from the literature in Section 7.1.1. The classes of analytics and specific applications are: 1) **visualization**: grid aggregation [233] groups the elements within a grid into a single element for multi-resolution visualization, 2) **statistical analytics**: histogram renders data distribution with equi-width buckets, 3) **similarity analytics**: mutual information reflects the similarity or correlation between two variables, 4) **feature analytics**: logistic regression measures the relationship between a dependent variable and multiple independent variables; 5) **clustering analytics**: k-means tracks the movement of centroids in different time-steps [257]; and 6) **window-based analytics**: moving average and moving median compute average and median in a sliding window, respectively, Gaussian kernel density estimation plots data density with the Gaussian kernel, and Savitzky-Golay filter [184] is a well-known smoothing filter.

The above analytics programs can be applied on a variety of simulation programs. However, from a performance view-point, only two aspects of the simulation program are...
important for us – the memory requirements for the simulation, and relative to it, the amount of data that is either output or needs to be analyzed every time-step. Thus, we choose two open-source simulation programs that have very different amounts of output. Specifically, for every time-step in our experimental setup, Heat3D [5] generates large volumes of data, e.g., 400 MB per node, whereas Lulesh [7] has a moderate amount of output, which is typically smaller than 100 MB on each node.

Our experiments were conducted on two different clusters. The first cluster is a more traditional cluster with multi-core nodes – specifically, each node is an Intel(R) Xeon(R) Processor with 4 dual-core CPUs (8 cores in all). The clock frequency of each core is 2.53 GHz, and the system has a 12 GB main memory. We experiment with time sharing mode only on this cluster, as the simulation program can be expected to scale with all available cores. We have used up to 512 cores (64 nodes). The second cluster has a many-core accelerator on each node, and both time sharing and space sharing modes are used and compared. Each node on this cluster has an Intel Xeon Phi SE10P coprocessor, with 61 cores and a clock frequency of 1.1 GHz (488 cores in total). The memory size of coprocessor is 8 GB.

7.4.2 Performance Comparison with Spark

Although Spark can directly load data from memory and hence can address the data loading mismatch, it cannot overcome the other three mismatches mentioned in Section 7.1.2. Thus, to make a fair comparison, we let Spark bypass all the other mismatches with the following setup: 1) to bypass the programming view mismatch, the simulation program was replaced by a simple emulator – a sequential program that outputs double precision array elements that follow a normal distribution, and in addition, the
Figure 7.3: Performance Comparison with Spark
experiments were only conducted on a single node with 8 cores, 2) the memory constraint mismatch was also addressed by the use of the emulator which hardly consumed any extra memory, and thus there was no tight memory bound for the analytics programs; and 3) to bypass the programming language mismatch, the emulator used by Spark was written in Java. 40 GB data was output from the simulation, over 800 time-steps, and the number of threads used for analytics was varied from 1 to 8. The version of Spark used was 1.1.1.

We used three applications for comparison, with the following parameters – 1) logistic regression: the number of iterations and the number of dimensions were 10 and 15, respectively; 2) k-means: the number of centroids, the number of iterations, and the number of dimensions were 8, 10, and 64, respectively; and 3) histogram: 100 buckets were generated. Particularly, both logistic regression and k-means were implemented based on the example codes provided by Spark. Since the emulation code was not parallelized, here we only report the computation times of analytics.

The comparison results are shown in Figure 7.3. Smart can outperform Spark by up to 21x, 62x, and 92x, on logistic regression, k-means, and histogram, respectively. The reason for such a large performance difference is three-fold. First, like other MapReduce implementations, Spark emits massive amounts of intermediate data after the map operation, and grouping is required before reduction. By contrast, Smart performs all reduction in place of reduction maps, avoids emitting any key-value pairs, and thus, completely eliminates the need for grouping. Moreover, every Spark transformation operation makes a new RDD (Resilient Distributed Dataset) [251] due to its immutability. In comparison, all Smart operations are carried out on reduction maps and combination maps, and these maps can be reused even for iterative processing. Further, Spark serializes RDDs and send them through network even in local mode, whereas Smart avoids copying
any reduction object from reduction map to combination map, by taking advantage of the 
shared-memory environment within each compute node.

Besides the efficiency advantage, we can also see that Smart scales much better than 
Spark, at least in the shared-memory environment. Particularly, Smart can achieve a 
speedup of 7.95, 7.71, and 7.96, by using 8 threads on logistic regression, k-means, and 
histogram, respectively. This is because that, Spark can only allow the number of worker 
threads to be controlled by the user, while it still launches extra threads for other tasks, 
e.g., communication and driver’s user interface. Particularly, we can see that, when 8 
worker threads were used for Spark execution, the speedup becomes relatively small, 
because not all 8 cores are being used for computation. By contrast, Smart does not 
launch any extra threads, and the analytics is efficiently parallelized on all threads.

In addition, Smart can also achieve a much higher memory efficiency than Spark. It 
turns out that for all the three applications, Spark takes up constantly over 90% of the 
total memory (12 GB) whereas the memory consumption of Smart is only 4.3% (528 MB). 
Since the time-step size is already 512 MB, the analytics program run by Smart actually 
consumes only around 16 MB memory. Note that the time-step size is much smaller than 
the memory capacity, and hence Spark is very unlikely to spill the input to the disk.

7.4.3 Performance Comparison with Low-level Analytics Programs

In the second experiment, we compared both the programmability and performance of 
analytics programs written using Smart against the ones that were manually implemented 
in OpenMP and MPI. We used logistic regression and k-means with the same parameters 
as in Section 7.4.2. 1 TB data were processed on a varying number of nodes, ranging from 
8 to 64.
First, it turns out that Smart is effective in simplifying application development, by saving the efforts on implementing and debugging low-level parallelization details. Specifically, for k-means and logistic regression, 55% and 69%, respectively, of the lines of OpenMP/MPI codes in the low-level implementations are either eliminated or converted into sequential code by Smart. Note that these low-level codes are usually the most error-prone part for the programmers.

Second, we will like to understand performance overheads that arise as well. Figure 7.4 shows the results. First, we find that the low-level codes for k-means can outperform Smart version by up to 9%. Such performance difference is mainly due to the extra overheads involved in the global combination of Smart. In the manual implementation, the synchronized data is stored in contiguous arrays, and the global synchronization can be done by a single MPI function call (MPI_Allreduce). By comparison, Smart stores reduction objects in a map structure noncontiguously, and hence an extra serialization of these objects is required by global combination. Note that we
follow such a design for a better applicability and flexibility – the keys do not have to be continuous integers on each node, and early emission of reduction objects can be supported. Second, it turns out that the performance difference on logistic regression is unnoticeable, because only a single key-value pair is maintained in this application and trivial serialization is needed. Overall, since in practice the total processing cost is mostly dominated by the simulation program, we do not expect noticeable overheads from our framework over hand-written low-level code.

7.4.4 Scalability Evaluation

The next set of experiments evaluate the scalability of Smart, by using both Heat3D and Lulesh simulations, and nine analytics programs: 1) grid aggregation: the grid size was 1,000; 2) histogram: the number of buckets was 1,200; 3) mutual information: the number of buckets for each variable was 100, and hence the 2-dimensional space was divided into up to 10,000 cells; 4) logistic regression: the number of iterations and the number of dimensions were 3 and 15, respectively; 5) k-means: the number of centroids, the number of iterations, and the number of dimensions were 8, 10, and 4, respectively; and 6) the four window-based applications, including moving average, moving median, (Gaussian) kernel density estimation, as well as Savitzky-Golay filter: the window sizes were all 25.

First, we evaluate the total processing times on Heat3D, as we scale the number of compute nodes from 4 to 32, with 8 threads on each node being used for both simulation and analytics. 1 TB data was output by Heat3D over 100 time-steps. As Figure 7.5 shows, Smart can achieve 93% parallel efficiency on average for all the applications. Particularly, we can even see that, for some cases where 16 nodes are used, a super linear scalability can
be achieved. Such an extra speedup is caused by the reduction in memory requirements per node as more compute nodes are used.

Second, we evaluate the performance of scaling the number of threads on 64 nodes by using Lulesh. Lulesh output 1 TB data over 93 time-steps. The number of threads used for both simulation and analytics per node was up to 8. Figure 7.6 shows the results. Smart can achieve 59% and 79% parallel efficiency on average for the first five applications, and the other four window-based applications, respectively. The difference in parallel efficiency is related to the nature of these applications. For example, compared with the first five applications, the window-based applications are more compute-intensive, and the synchronization overheads weigh much less in the total processing cost, leading to a better scalability.
7.4.5 Evaluating Memory Efficiency

We next demonstrate a key advantage of Smart design (its time sharing mode implementation) – in-situ analytics can be supported even without involving an extra copy of the simulation output. Many simulation programs practically use almost all available memory on the machine, and unnecessary copying of data can lead to severe performance degradation – this is even more important as memory to flop ratio has decreased with many recent systems. We evaluate such impact by comparing the performance with an implementation involving data copy.

In this set of experiments, 1 TB data was output by Heat3D on 4 nodes, and by Lulesh on 64 nodes. Logistic regression and mutual information were used as analytics programs on Heat3D and Lulesh, respectively, with the same parameters as for the experiments in

Figure 7.6: In-Situ Processing Times with Varying # of Threads on Lulesh (Using 64 Nodes)
(a) In-Situ Processing Times of Logistic Regression on Heat3D (Using 4 Nodes)

(b) In-Situ Processing Times of Mutual Information on Lulesh (Using 64 Nodes)

Figure 7.7: Evaluating the Efficiency of Time Sharing Mode
Section 7.4.4. To vary the memory consumption in the simulation program, we varied the time-step size for each simulation run as follows. For Heat3D, we could vary the length of one dimension of the 3D problem size, and hence we varied the time-step size as well as the memory consumption linearly. Particularly, the time-step size was varied from 0.6 GB to 1.8 GB. For Lulesh, we could vary the edge size of a 3D array cube simulated on each node, and hence by varying the edge size linearly, we could result in a cubic growth of memory consumption. Particularly, the edge size was varied from 100 to 233, and the corresponding time-step size was varied from 1.5 GB to 18.3 GB.

As shown in Figure 7.7, we can see that when our system does not involve any data copy, there can be a notable performance improvement. For Heat3D, with a time-step greater than 1.4 GB, our system can outperform the other implementation by up to 11%. Note that a time-step of 1.8 GB makes the system reach the memory bound in our setup, since a time-step of 2 GB can result in a crash. For Lulesh, with an edge size smaller than 220, only a performance gain of up to 7% is achieved. This is because the size of simulated data on each node is only 247 MB, which is very small compared with the memory capacity (12 GB). However, when the edge size reaches 233, the memory consumption of the implementation involving data copy becomes very close to the physical capacity, and hence its processing time increases substantially. For this case, our system can achieve a speedup of 5x.

7.4.6 Comparing Time Sharing and Space Sharing Modes

Recall that in the space sharing mode, both simulation and analytics run concurrently, using two separate groups of cores on each node. All of our experiments so far have considered the time sharing mode only. Now we evaluate the efficiency of space sharing
Figure 7.8: Evaluating the Efficiency of Space Sharing Mode
mode, by comparing against the performance in time sharing mode, as well as the performance of pure simulation as a baseline, on a many-core cluster.

In this set of experiments, 1 TB data was output by Lulesh on 8 Xeon Phi nodes. Since it turns out that the simulation could not benefit from hyperthreading on the coprocessors, we only used 60 threads for computation in this mode, and 1 core was reserved for scheduling and communication. Histogram, k-means, and moving median were used as analytics programs, with the same parameters as for the experiments in Section 7.4.4. Besides time sharing and ‘simulation-only’ versions, to vary the number of cores used for simulation and analytics in space sharing mode, we used 5 different versions, in which the number of cores used for simulation was varied from 50 to 10, and the remaining cores were used for analytics.

The results are shown in Figure 7.8. Here, “n,m” denotes a space sharing scheme with n threads for simulation and m threads for analytics. First, although the best performance in space sharing mode is achieved by different schemes for different applications, it does not incur too much overhead compared with the ‘simulation-only’ performance, even with a moderate amount of computation (as shown in Figure 7.8(c)). Second, the best performances in space sharing mode for k-means and moving median are achieved by “50,10” and “30,30”, and reflect a performance improvement of over the time sharing mode by 10% and 48%, respectively. This is because space sharing mode can make better use of some cores, when simulation reaches its scalability bottleneck. In addition, we also notice that not all applications can benefit from space sharing mode – the best performance of histogram in space sharing mode (achieved by “50,10”) is 4.4% lower than the performance from the time sharing mode. This is because the synchronization (or message passing) cost in histogram is relatively higher that in the other two applications,
and space sharing mode can only execute the message passing in simulation and analytics sequentially, to avoid the potential data race in MPI, i.e., only a single thread can call MPI function at a time during concurrent execution. Thus, we conclude that space sharing mode can be advantageous when a simulation program does not scale well with increasing number of cores, but it is not a good fit for the applications involving frequent synchronization.

7.4.7 Evaluating the Optimization for Window-Based Analytics

![Graph](image)

(a) In-Situ Processing Times of Moving Average on Heat3D (Using 4 Nodes)

![Graph](image)

(b) In-Situ Processing Times of Moving Median on Lulesh (Using 64 Nodes)

Figure 7.9: Evaluating the Effect of Optimization for Window-Based Analytics
The last set of experiments evaluate the effect of optimization for window-based analytics. Specifically, we compare the optimized version against an implementation that does not set a trigger function and hence cannot support early emission of the reduction object. In the first experiment, we used Heat3D to simulate 300 GB data, and used moving average as the analytics program on 4 nodes. Similar to the previous experiment, we varied the time-step size from 0.5 to 1 GB in Heat3D, and the window size of moving average was 7. In the second experiment, 1 TB data was output by Lulesh, and then analyzed by moving median on 64 nodes. We also varied the size of simulated data on each node from 5.2 to 186 MB in Lulesh, by varying the edge size of array cube from 60 to 200, and the window size of moving median was 11.

The results are shown by Figure 7.9. We can see that the optimization can lead to a speedup of up to 5.6 and 5.2 in the two experiments, respectively, which is because of the memory efficiency. For instance, with such an optimization, it turns out that the maximal number of reduction objects maintained by Smart can be decreased by 1,000,000 times for the case of the moving average application. Moreover, a time-step of 1 GB in Heat3D, or an edge size of 200 in Lulesh, can result in a crash from the implementation without the trigger mechanism, due to the extremely large memory consumption – we have not even reported the results for these two cases.

7.5 Related Work

As recent years have witnessed an increasing performance gap between I/O and compute capabilities, in-situ scientific analytics [118, 127, 255, 267] has attracted much attention. The research on in-situ scientific analytics has been mainly focused on two areas – applications and platforms. In-situ applications and algorithms have been
extensively studied, with work on topics including indexing [117, 127], compression [128, 268], visualization [114, 250, 266], and other analytics like object tracking [257], feature extraction [129], and fractal dimension analysis [219]. On the other hand, in-situ resource scheduling research that offers platforms can be classified into time sharing and space sharing categories. An example of time sharing platform is GoldRush [262], which runs analytics on the same simulation cores. Since simulation and analytics are tightly coupled, cycle stealing becomes critical for performance optimization. For the case of space sharing platforms, CPU utilization of simulation and analytics are decoupled, while the memory bound on analytics still holds. Examples of efforts include Functional Partitioning [139], and the systems Damaris [68] and CoDS [256]. By contrast, our work explores the opportunities in in-situ scientific analytics at the programming model level. Broadly, in-situ applications can benefit from Smart by adapting the system API and abstracting parallelization, while Smart can be deployed on top of any of the in-situ resource scheduling platforms.

In a broader context of online resource scheduling platforms, another two processing modes have been studied in addition to in-situ processing. The first is in-transit processing, where by leveraging extra resources, online analytics can be moved to dedicated staging nodes that are different from the nodes where simulation runs. Platforms supporting this mode include PreDatA [261], GLEAN [219], JITStager [18], and NESSIE [167]. Based on the observation that in-situ and in-transit modes can complement each other, the second mode is that of hybrid processing. This mode is supported on many platforms, including ActiveSpaces [65], DataSpace [66], FlexIO [263], and others [37]. Our system can be incorporated into these platforms to support in-transit or hybrid processing.
We had earlier compared the limitations of various MapReduce implementations for possible in-situ analytics, and have extensively compared our work against Spark. In addition, iMR [147] is specifically designed for in-situ log stream processing. To meet the in-situ resource constraints, iMR focuses on lossy processing and load shedding. Smart, in comparison, is based on a distinct API that reduces memory requirements. Further, integrating MapReduce with scientific analytics has been a topic of much interest recently [42, 146, 154, 185, 213, 225, 237, 260]. SciHadoop [42] integrates Hadoop with NetCDF library support to allow processing of NetCDF data with MapReduce API. SciMATE [237] is a MapReduce variant that can transparently process scientific data in multiple scientific formats. Zhao et al. [260] implement a parallel storage and access method for NetCDF data based on MapReduce. The Kepler+Hadoop project [225] integrates MapReduce with Kepler, which is a scientific workflow platform. Other MapReduce frameworks [154, 185, 213] are also developed for scientific analytics. In contrast, Smart is designed for in-situ processing, and accordingly, the focus is on addressing the data loading mismatch, memory constraint, and other similar issues. Moreover, Smart is not bound to any specific scientific data format, since its input is considered to be resident in (distributed) memory.

Additionally, the topic of accelerating applications using GPUs, Intel MIC and heterogeneous resources have also been studied in our group [49–53]. These works have been focusing on different aspects including scheduling, memory hierarchy, and pattern-specific optimizations, and they can be potentially integrated into our framework on different architectures.
7.6 Summary

In this work, we have developed and evaluated a system that applies MapReduce-style API for developing in-situ analytics programs. Our work has addressed a number of challenges in creating data analytics programs from a high-level API that is efficient and can share resources with an ongoing simulation program.

We have extensively evaluated our framework. Performance comparison with Spark shows that our system can achieve high efficiency in in-situ analytics, by outperforming Spark by at least an order of magnitude. We also show that our middleware does not add much overhead (typically less than 10%) compared with low-level implementations. Moreover, we have demonstrated both the functionality and scalability of our system by running different simulation and analytics programs in different in-situ modes on clusters with multi-core and many-core nodes. We can achieve 93% parallel efficiency on average. Finally, we show that our optimization for in-situ window-based analytics can achieve a speedup of up to 5.6. Smart is an open-source software, and the source code can be accessed at https://github.com/SciPioneer/Smart.git.
Chapter 8: Conclusions and Future Work

In this chapter, we first summarize the contributions of this dissertation, and then we discuss the future work.

8.1 Contributions

Our main objective is to provide high-performance data management and data processing support on array-based scientific data, targeting data-intensive applications and various scientific array storages. Specifically, we have developed a number of high-performance data management and data processing utilities, which can significantly reduce costs of data translation, data transfer, data ingestion, data integration, data processing, and data storage involved in many scientific applications. Our contributions can be summarized as follows.

- Our initial efforts towards scientific data management have built an SQL interface over HDF5 datasets, with the support for parallel server-side subsetting and aggregation. The queries can involve different predicates based on dimensional and/or value ranges. Our implementation has been shown to be able to significantly outperform OPeNDAP by saving both data translation and data transfer costs.

- Motivated by the data model mismatch between array-based scientific data and relational data for which SQL is originally designed, we have designed a system to
support a number of structural aggregations, which are unique in array data model and unable to be supported by SQL. Extensive evaluations have shown that our system not only can effectively avoid the data ingestion cost of loading data into a database, but also can significantly outperform SciDB even without considering its data ingestion cost.

- We have further improved the efficiency of scientific data management through the paradigm of approximate query processing. With the insight that bitmap indices can be a good summary structure to preserve both spatial and value distribution of the array data, we propose a novel approximate aggregation approach over arrays. The key feature is the capability of operating entirely on the compact bitmap indices rather than the original raw datasets. A comparison with both histogram and sampling methods has shown high efficiency of our implementation.

- We posit that bitmap indexing can also be applied to more complex data mining tasks over array data, especially for the ones involving frequent aggregations. Particularly, we have designed a novel subgroup discovery algorithm over arrays. We have demonstrated both high efficiency and effectiveness of our algorithm by comparing against VIKAMINE, which implements a different subgroup discovery algorithm.

- We investigate scientific data processing mainly in the context of MapReduce. Based on the observation that scientific data is often stored in different data formats, we have developed a novel MapReduce-like framework that can be customized to support transparent processing on any of the scientific data formats. In contrast to the traditional MapReduce implementations that require integrating and/or
reloading scientific data, our system can avoid the data integration and/or data reloading costs. We have evaluated the performance using three popular data analysis tasks, and each instance has processed multiple data formats, including NetCDF, HDF5, and flat-files.

- We then shift the focus from offline analytics to in-situ analytics, which can avoid expensive data movement of simulation output data by co-locating both simulation and analytics program. We have developed a MapReduce-like framework which can support efficient in-situ scientific analytics in two different modes – time sharing and space sharing. This system can substantially reduce both data transfer and data storage costs. Performance comparison with Spark has demonstrated that our system can outperform it by at least an order of magnitude. We have also shown high efficiency and scalability of our middleware, by using nine applications on both multi-core and many-core clusters.

In the future, we will like to extend our work in both data management and data mining directions. On one hand, we can apply bitmap indexing to some data mining tasks such as exception rule mining [99, 209] or anomaly detection [84, 85], and also develop some middlewares for bitmap-based analysis tasks. On the other hand, we can adapt our MapReduce implementations for bioinformatics applications and approximate algorithms.

8.2 Future Work

In the future, we plan to extend our works to provide richer data management and data processing support. The future work is mainly motivated by two observations. First, we find bitmap indexing has a huge potential in assisting many data analysis tasks over array-based scientific datasets, especially for the ones involve some sort of associations that are
described by both dimensional and value-based predicates. Second, it is also desirable to adapt our MapReduce frameworks for other applications.

8.2.1 Bitmap-Based Data Analysis

Bitmap-Based Exception Rule Mining

Given a set of predefined rules, which may come from priori knowledge or a certain rule mining application (e.g., correlation mining or subgroup discovery), exception rule mining [99] aims to discover interesting exceptions that contradict some of these rules. Unlike the common sense rules that often have high support and confidence, exception rules can represent some critical phonomania that are either unknown or omitted, and these rules often have low support but high confidence. A motivating example in scientific data can be as follows. For oceanic data, temperature can often be correlated with the depth to some extent – temperature decreases as the depth increases in common sense. However, in a certain area there may exist an exception that contradicts the correlation. This can be of great interestingness, indicating a heat source, e.g., an undersea volcano.

While there is clearly a need to apply exception rule mining to scientific discovery over array-based scientific data, to the best of our knowledge, none of existing algorithms has considered this problem. In the future, we plan to design a bitmap-based exception rule mining algorithm. Like the subgroup discovery algorithm presented in Chapter 7, this algorithm can entirely operate on bitmap indices, because each rule can still be described in a form of association, i.e., a conjunction of dimension-based or value-based attribute-range pairs.

Moreover, there can also be a very straightforward solution from the viewpoint of subgroup discovery. Intuitively, an exception rule can be considered as a special subgroup in the context of one or multiple given rules. Based on this intuition, our exception rule
mining can be carried out in two steps. First, for each given rule, we can extract a data subset from the input by applying the rule as a filter. This step can be translated into a series of bitwise operations, like what we did in bitmap-based approximate aggregation method. Second, we can treat the data subset as the entire dataset in the view of subgroup discovery, and then run our subgroup discovery algorithm with customized quality function, to identify interesting subgroups that should be equivalent to the output of exception rule mining.

**Bitmap-Based Middleware**

Besides subgroup discovery, our group has also designed a number of bitmap-based data analysis tasks, including contrast set mining [264], correlation analysis [202], correlation mining [205], and missing value imputation [193]. However, all the above implementations are built in low-level programming language (C/C++), and the designs are neither fault-tolerant nor sufficiently parallelized.

We plan to build a middleware with the goals as follows: 1) to abstract the execution flow with generic data structures and operations; 2) to make fault tolerance and parallelization transparent to the programmers. The first goal can not only facilitate the maintenance of the existing implementations, but also simplify the development of other similar bitmap-based applications. The second goal can help scale out the implementations.

We can fulfill the above goals on Spark [11], since Spark has been widely adopted and can well meet our needs. The design can be implemented in two steps. First, we define a bitmap-based resilient distributed dataset (RDD) [251], which can represent a summary structure for an entire input array or its subset. Particularly, it can be in a form of a vector or map, where each entry encapsulates a bitvector and its associated pre-aggregation statistics.
for a variable. The pre-aggregation statistics should be customizable. Second, we define a set of bitmap-based operations for supporting RDD transformations. These operations should at least include: 1) index operation to produce an RDD given an array; 2) filter operation based on a dimension-based or value-based predicate; 3) merge operation over two bitmap-based RDDs; 4) histogram operation to count the number of 1s in each bitvector and then produce a histogram; and 5) a number of aggregate operations over the produced histogram to produce an aggregation result.

8.2.2 Smart Extension

Supporting Bioinformatics Applications

Despite that MapReduce is originally designed to process text files, it has been adapted for processing datasets in a variety of disciplines. For example, many MapReduce variants (e.g., GATK [157], Biodoop [137], and Hadoop-BAM [165]) have demonstrated that MapReduce already has a substantial base in Next-Generation sequencing analysis as well as other bioinformatics domains.

Motivated by this trend, we plan to customize our Smart system for bioinformatics analytics. Although the current input data type of our system is array, it is not hard to adapt it for taking sequencing data as the input. Specifically, we can mimic our precursor system [231] for partitioning sequencing data and implementing some analysis tasks.

Supporting Online Analytics for Approximate Algorithms

Recent years has witnessed a rapid growth in dataset sizes in a variety of disciplines, from business to the sciences, and large-scale data analytics is one of the key ‘big data’ challenges today. Specifically, both time and resource constraints are often imposed in many use cases. First, to obtain crucial and timely insights for decision making and
planning, the analysis over large datasets is often expected to be *interactive*, and hence high delay should be avoided. Consequently, early results from runtime *approximation* techniques like online sampling has been gaining great popularity, by giving a better control over the entire analytics plan. Besides the requirements of responsiveness, another major trend in both scientific and commercial sectors is that, nowadays a growing number of large-scale analytics tasks have been executed in the ‘pay-as-you-go’ cloud environments. In response to this trend, online analytics based on runtime approximation has been shown to be a cost-effective paradigm for increasingly data-intensive analytics [113]. Particularly, *early termination* of analytics with satisfying estimates can be translated to saving in execution times, power, and machines.

We plan to extend our Smart system to supporting online analytics. Specifically, it can incrementally process the input through online sampling. Given the partial results of the data already processed, fast estimates can be updated at runtime, based on user-defined *estimation method*. Once the estimates satisfy the user’s requirement, which is described as a *termination condition*, early termination is carried out to reduce latency and save resources. We plan to extend the API by allowing the user to customize both estimation method and termination condition. Initial efforts have been made in [232].
Bibliography


226


232


238


