© 2007 by Rishi Rakesh Sinha. All rights reserved.

INDEXING SCIENTIFIC DATA

BY

RISHI RAKESH SINHA

B.S., State University of New York at Stony Brook, 2002
M.S., University of Illinois at Urbana-Champaign, 2004

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

INDEXING SCIENTIFIC DATA

Rishi Rakesh Sinha, Ph.D.

Department of Computer Science

University of Illinois at Urbana-Champaign, 2007

Marianne Winslett, Advisor

The ability to extract information from collected data has always driven science. Today's large computers and automated sensing technologies collect terabytes of data in a few weeks. Extracting information from such large amounts of data is like trying to find a needle in a haystack. For efficient information extraction, we need disk-based indexing schemes that can efficiently handle queries restricting ranges on dozens of attributes. Unfortunately, the unique characteristics of scientific data and queries cause traditional indexing techniques to have poor performance on scientific workloads, occupy excessive space, or both.

Bitmap indexes were proposed as a solution to these problems. However, in experiments with scientific data and queries, we found that previously proposed variations of bitmap indexes either were quite slow or required excessive storage for processing the large-range query conditions our scientists used. Scientists also told us that bitmap indexes, though smaller than traditional indexes, were too large for scientific data warehouses. Our scientists also wanted an efficient method to consolidate the data points returned by the indexes into larger, more meaningful regions of interest.

To address these three problems, we introduced multi-resolution bitmap indexes, which group data into bins at multiple granularities. We achieved a query performance which is 10 times faster

than traditional bitmap indexes by using bitmap indexes built at these multiple granularities. To address the issue of size, we introduced an adaptive version of multi-resolution bitmap indexes. The adaptive index adds and drops auxiliary indexes as needed for the query workload and is a fraction of the size of the data being indexed. We achieved a performance improvement of a factor of 6, compared to an ordinary multi-resolution bitmap index of the same size. We also introduced a novel algorithm to consolidate data points into regions of interest. By exploiting the special properties of compressed bitmap indexes and scientific meshes we achieved sublinear running times, with respect to the number of points in the query result, for both the index lookup and region consolidation.

। विज्ञानं त्वेव विजिज्ञासितव्यमिति ।

- Chhāndogya Upanishad 7.17.1

Search for knowledge should be perpetual.

To Ma, Papa and Swati

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Marianne Winslett, for her continuous guidance and inspiration throughout my thesis study. I benefited tremendously from her help and insights in many aspects of my life as a graduate student, from literature study to technical writing, from preparing presentations to reviewing papers, from traveling to job hunting. I feel blessed to have an advisor like her. I am also very thankful to my thesis committee members, Professor Geneva Belford, Professor Jiwaei Han, Professor YuanYuan Zhou and Dr. Mike Folk, for their precious support and assistance during my thesis research and for their career advice.

Secondly, I am grateful to the team at The HDF Group for their continual support, both financial and moral, during my graduate life. Mike, Elena, Albert, Quincey, Peter: thanks for your continued support. I would also like to thank the Computer Science and Engineering program at the University of Illinois for providing me with a fellowship for a year and for providing conference travel support. I also appreciate the computing facilities and user support provided by the Department of Computer Science and the National Center for Supercomputing Applications (NCSA) at the University of Illinois. In particular, I am deeply thankful to the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois, for its wonderful research program, facilities and research members. Much of my thesis research was motivated by CSAR's applications and these applications provided me with excellent test cases. I thank my many collaborators at CSAR: Dr Mike Heath, Bob Fiedler, and John Norris, for their tremendous help in getting me oriented with their data sets. I thank the Data Management Group at Lawrence Berkeley National Laboratory for hosting me for a year and providing an environment that fosters innovation. I thank my collaborators at Lawrence Berkeley Lab for their help and guidance: Arie Shoshani, John Wu,

Ekow Otoo, and Kurt Stockinger.

Inspiration for many of the problems addressed in my thesis was drawn from talking to people working in various scientific domains. I would like to thank Pratyush Sinha and Vikas Mehra from the Civil and Environmental Engineering Department for helping me in understanding the earth sciences data sets and providing me with interesting queries in the domain. I would also like to thank Joseph Moore at the Astronomy Department at University of Illinois for his insights into managing large astronomy data warehouses.

My thesis could not have been completed without assistance from the Scientific Data Management group members, Soumyadeb Mitra and Arash Termehchy. I am especially thankful to Soumyadeb for all the wonderful discussions and the help he gave me at the starting of the Maitri project. I would also like to thank my co-researchers in the security group for listening through my various talks and helping me in honing my presentations. In addition, I thank the faculty members and fellow students in the Database and Information Systems Laboratory at the University of Illinois, for their warm help during my stay at the database group.

I thank my many friends at the University of Illinois, who helped make my study in Urbana-Champaign such a wonderful time. I am grateful to Sayantan Chakrobarty, Abhishek Tiwari, Robin Dhamankar, Sruthi Bandhakavi, Rob McCann, Warren Shen, Yoonkyong Lee, Vikas Mehta, and Amit Sharma. Many of them also helped my research in one way or another. I owe special thanks to Smruti Sarangi, my roommate and closest friend during my graduate life. His advice at critical junctures has immensely helped me.

Finally I am deeply thankful to my parents, Manorama and Navin Sinha, for their endless support, patience and encouragement. I would also like to thank my brothers Pushkar Sinha and Rajeev Sinha and my sisters-in-law Shakti Kamal and Bishakha Sinha for their continued support. Last but not the least I would like to thank my fiancée Swati, who never fails to bring a smile to my face.

Table of Contents

List of Tables xii

List of Figures xiii

Chapter 1 Introduction 1
- 1.1 State of the Art in Scientific Data Management 2
- 1.2 Maitri Scientific Data Management System 4
- 1.3 Indexing Scientific Data 5
- 1.4 Research Issues in Bitmap Indexes 8
- 1.5 Contributions of This Thesis 9
- 1.6 Thesis Roadmap 12

Chapter 2 Scientific Data Management: The Big Picture 13
- 2.1 Need for a Scientific Data Management System 13
- 2.2 Previous Scientific Data Management Solutions 18
- 2.3 The Maitri Framework 19
- 2.4 Challenges in Building Maitri Modules 23
  - 2.4.1 Block Manager 23
  - 2.4.2 Query Manager 25
  - 2.4.3 Metadata Manager 26
  - 2.4.4 Index Manager 28
  - 2.4.5 Buffer Manager 28
  - 2.4.6 Concurrency Manager 30
- 2.5 Realizing the Maitri Vision 31

Chapter 3 Related Work 32
- 3.1 Traditional Indexing Schemes 32
- 3.2 Bitmap Indexes 33
- 3.3 Previous Work on Bitmap Indexes 34
- 3.4 Retrieving Regions of Interest 40

Chapter 4 Multi-resolution Bitmap Indexes 42
- 4.1 Drawbacks of Previous Solutions 42
  - 4.1.1 Interval Encoding 42
  - 4.1.2 Binning 46

4.1.3 Unencoded Bitmap Indexes 47
4.2 Multi-resolution Bitmap Indexes 48
4.2.1 Reducing the Number of Bit Vectors 49
4.2.2 Querying a Multi-resolution Bitmap Index 51
4.2.3 Index Creation 53
4.3 Sizes of Multi-resolution Bitmap Indexes 55
4.4 Experiments 56
4.4.1 Earth Science Domain 57
4.4.2 Rocket Science Domain 62

Chapter 5 Parallelizing Bitmap Indexes 68
5.1 Problems with Previous Solutions 69
5.2 Bitmap Indexes for Clusters 69
5.3 Experiments 74
5.3.1 Earth Science Domain 74
5.3.2 Rocket Science Domain 75

Chapter 6 Adaptive Multi-resolution Bitmap Indexes 77
6.1 Disk Space Constraints in Scientific Data Warehouses 77
6.2 Locally Optimal Bin Boundaries 80
6.3 Exploiting Query Locality 84
6.4 Experimental Data and Queries 87
6.4.1 Real-World Data and Query Log 87
6.4.2 Modeling Query Patterns and Data 87
6.4.3 Generating Synthetic Data 90
6.5 Experiments 91
6.5.1 The Effect of Memory and Disk Cache Size 92
6.5.2 The Effect of the Number of Bins 96
6.5.3 The Effect of the Index Size Limit 99
6.5.4 Performance with Synthetic Query Logs and SDSS Data Set 101
6.5.5 Synthetic Data and Real-World Query Log 102
6.5.6 Multiple Files 104

Chapter 7 Consolidating Query Results into Regions 107
7.1 WAH Compressed Bitmaps 112
7.2 From Bitmaps to Query Lines 115
7.3 Region Growing Phase 118
7.3.1 Union-Find 120
7.3.2 Region Growing 122
7.4 Interesting Mesh Orderings 126
7.5 Experiments 129
7.5.1 Semi-Structured Mesh 130
7.5.2 Unstructured Meshes 133

Chapter 8 Conclusions **139**
8.1 Summary 140
8.2 Future Work 144

References **146**

Appendix A **153**

Author's Biography **155**

List of Tables

4.1 Number of bit vectors operated upon. 51
4.2 Number of bins and bin widths for vegetation attributes on which indexes were created. 58
4.3 Queries used in earth science domain experiments. 59
4.4 Queries used in rocket science domain experiments. 63
4.5 Processor utilization for query Q6′. 67

6.1 SDSS adaptive bitmap index sizes. 93

7.1 Possible linearizations of different kinds of meshes. 114
7.2 Sequences of words that can form a single query line. 117
7.3 Operations on the Union-Find Array. 122

List of Figures

1.1 Architecture of the Maitri framework for scientific data management. 5
1.2 Data distribution for the MODIS Estimated Vegetation Index (EVI) attribute. EVI has low selectivity in queries, as most queries request a wide subrange of [2,000, 6,000]. 6

2.1 Architecture of the Maitri framework for scientific data management. 16
2.2 Maitri application programming interfaces. 21
2.3 The architecture of the buffer manager. 29

3.1 Bitmap index basics. 35
3.2 Using a bitmap index with 8 bins to respond to queries. 37

4.1 The effect of rocket query range width. The Y axis shows the number of iterations of the inner loop of the OR code (upper line), and the number of words read from the original and intermediate-result bit vectors (lower line). The X axis shows the width of the query range. 47
4.2 Answering a query using a 2-resolution bitmap index. The highlighted boxes show which parts of the high and low-resolution indexes are used to answer the query. 49
4.3 The effect of rocket query range width with single- and multi-resolution bitmap indexes. The log-scale Y axis shows the number of bit vector words read and the number of inner loops of the OR code. The X axis shows the query range width. 50
4.4 Range query processing algorithm for a multi-resolution bitmap index with equi-width bins. 52
4.5 Index sizes for vegetation data. 58
4.6 Uniprocessor performance for earth science queries Q1-Q9, excluding removal of false positives. 61
4.7 Index sizes for rocket science data. 64
4.8 Uniprocessor performance for rocket science queries Q1′-Q7′ and Q10, excluding removal of false positives. 65
4.9 Elapsed time to read underlying data files to remove false positives. 66

5.1 Processing a query $[lb, ub]$ in parallel. Here n is the number of processors performing the query and m is the number of index locations (the number of processors involved in index creation). Both are powers of 2. 73
5.2 Multiprocessor performance for index creation and lookup, with earth science data. 75

5.3 Multiprocessor performance for index creation and lookup, with rocket science data. 76

6.1 Answering queries with an unbinned (center), binned (left), or multi-resolution (right) bitmap index. 79
6.2 Architecture of a two-level adaptive bitmap index (ABI). 83
6.3 Algorithm for creation of a LOMBI 84
6.4 Spatial locality of SDSS queries with RA endpoints. 85
6.5 The projection index cache architecture. 86
6.6 The state diagram for scientific queries. 89
6.7 SDSS queries on RA, between query numbers 18K and 20K. 89
6.8 SDSS queries on RA, between query numbers 30K and 32K. 90
6.9 Time to run all RA queries in the 15,000 test query log, using a two level index. The curve represents the percentage of queries answered from cache. 94
6.10 Time to run all DEC queries in the 15,000 test query log, using a one level index, with 25 percent space restriction. The curve represents the percentage of queries answered from cache. 96
6.11 Performance effect of varying the number of level 1 bins, using the 15000 query log from SDSS for the RA attribute with a 25% space constraint. 97
6.12 Performance effect of varying the number of level 2 bins, using the 15000 query log from SDSS for the RA attribute with a 25% space constraint. 98
6.13 Total run time for the RA queries in the SDSS test log, under different size limits. 100
6.14 Total run time for the DEC queries in the SDSS test log, under different size limits. 101
6.15 Total run time for 5,000 synthetic RA queries over real SDSS data. 102
6.16 Total run time for 15000 queries from the SDSS query log, on synthetic RA data. . 105
6.17 False positive removal costs with multiple files, in seconds. 106

7.1 Examples of different types of meshes used in scientific/engineering computations. 108
7.2 A 49-point mesh with two regions of interest. 109
7.3 A possible linearization of the example mesh. The corresponding uncompressed and the compressed bit vectors shown below the mesh. 113
7.4 Converting a bit vector into line segments. 116
7.5 Splitting the line segments into query lines. 119
7.6 The union-find algorithm. 121
7.7 The union-find array. 122
7.8 The connected component labeling algorithm. 123
7.9 The modified connected component labeling algorithm. 123
7.10 The CLLLines algorithm. 124
7.11 One plane in a gyrokinetic mesh, with two regions of interest. 126
7.12 The CLLLines&Ordering algorithm. 127
7.13 Creating large mesh lines using diameter of a graph. 128
7.14 Reordering the mesh using a modified DFS algorithm. 129
7.15 Response time for range queries over the Potential variable in the plasma simulations. 132
7.16 Complete Gyrokinetic Mesh. 133
7.17 Line information for queries over the Potential attribute from the plasma simulation. 134
7.18 Response time for range queries over the Z coordinate, with rocket simulation data. 136

7.19 Response time for range queries over the X and Y coordinates, with rocket simulation data. 137
7.20 Response time for range queries over X, Y, and Z attributes, for rocket simulation data. 138

Chapter 1

Introduction

When Kepler formulated his famous laws of planetary motion, he was using data painstakingly gathered by his mentor Tycho Brahe to advance science. With today's advances in sensor, automation, and computation technologies, gathering scientific data has become very easy. Large data sets produced by observation or simulation allow scientists to formulate new theories and to test existing theories against actual data. This ability has led to the production and consumption of large amounts of data by scientists.

For example, the Sloan Digital Sky Survey currently offers twelve terabytes of data, which is being used by scientists across the world to study astronomical phenomena and discover new heavenly bodies. Looking at the earth instead of the sky, the Earth Observing Satellite system from NASA produces 3 terabytes of data per day. While these are both observational data, simulation data sets do not lag too far behind the observational data in terms of size. Each complex run at the Center for Simulation of Advanced Rockets at the University of Illinois at Urbana-Champaign produces a couple of terabytes of data. Future scientific applications will be even more data-intensive, due to improved observation-gathering technology and more complex and larger-scale simulation codes.

To extract information from such large amounts of data, scientists need to be able to select arbitrary subsets of these large data sets by constraining the values of scientific variables; this task requires efficient data management techniques, including a good indexing scheme. For example, to find all regions of the rockets and moments in time where the temperature was between 1000 and 2000 degrees and the pressure was over 500 Pascals, it is very inefficient to scan all data files to find the relevant points and consolidate them into regions. A good multidimensional index can

cut the processing time for this query from hours to seconds.

1.1 State of the Art in Scientific Data Management

Scientific data management is a much studied field. Some scientists have turned to commercial database management systems (DBMSs) to provide easy and efficient access to their data. Under this approach, scientific tools such as visualization and mining programs submit declarative queries to the DBMS, which optimizes and executes the queries. For example, the USD [JGL^{+}92] and Tioga [SCN^{+}93] systems for scientific data access were built on top of standard relational or object-relational [SR86] DBMSs. These systems were developed specifically for visualization of scientific data and did not try to address the broader aspects of scientific data management. As another example, the bioinformatics community initially standardized on the Oracle relational DBMS, but has started looking for alternatives. More recently, the Sloan Digital Sky Survey (SDSS) has used the SQL Server database engine to provide access to terabytes of astronomy data [SDSa]. However, the success of SDSS is yet to be replicated in big science. Generally, it is well known that traditional DBMSs are not suited to meet the broader needs of scientific data management for the following reasons [Dep04]:

- The array data type, something very important for scientists, is not very well supported in most commercial databases. Traditional DBMS also lack built-in support for easy storage and retrieval of many other data types important to scientists, such as mesh data and graph data.

- Traditional DBMSs require significant performance tuning for scientific applications, a task that scientific programmers are not well prepared or eager to address.

- The pricing structure of traditional DBMSs is ill suited for scientists, who commonly use multiple platforms spread over a wide geographic area, often with different architectures

and operating systems. For the scientists' codes and data to be easily portable to all their platforms, the same DBMS needs to be running on all the platforms, which is very expensive.

- Parallel database software is especially expensive, and is not available for the leading-edge parallel machines popular in scientific computing centers. While commercial vendors like SQL Server and Oracle have cluster versions of their DBMSs, their processor count based pricing structure makes them too expensive for large supercomputers. Moreover, ports for these systems are not available for all leading-edge parallel machines.

- Since most scientific data are written once and never updated, traditional concurrency control and logging mechanisms are an expensive liability.

Since commercial systems did not meet their needs, scientists banded together with software engineers to create their own data management software. Today most scientific data is stored in binary files whose formats were developed specifically for scientific data, and accessed through user-level libraries. These formats range from the generic (e.g., HDF [HDF] and netCDF [NET]) to the domain-specific (e.g., ROOT [ROO] for high energy physics and FITS [FIT] for astronomy). These systems store data together with their associated metadata, provide APIs for efficient storage and retrieval of the data, are available for little or no cost, and are available on all platforms a scientist is likely to use. However, they also have a number of shortcomings:

- *Navigational access:* These formats require scientific programmers to write code using the APIs specific to each format. The programmer must program the details of how to navigate through the data to find the items of interest, as in 1960s commercial DBMSs [Bac73].

- *Format specific and fragile:* Scientists often need to access data from a variety of sources, each with a different storage format; their application code must use a separate API for each format. The resulting applications are format-specific, and must be rewritten when the format changes. For example, when the Center for Simulation of Advanced Rockets at

UIUC shifted from the HDF format to the CGNS format, the entire reader for the new format had to be written scratch.

- *Lack of multi-file utilities:* These libraries are oriented towards operations on a single file at a time, while scientists usually have a huge number of files. Thus scientists must manage the file-level metadata themselves, typically through a complex file-naming scheme. In addition, the libraries do not provide buffering, caching, and indexing facilities that work across multiple files.

1.2 Maitri Scientific Data Management System

To address these shortcomings, scientists need a unified query interface that supports queries over data and metadata stored in a variety of file formats, and provides flexible support for buffering, caching, indexing, metadata management, and concurrency control. In this thesis we propose Maitri, a set of *data-format-independent, application-tailorable* libraries to provide these facilities. Maitri provides scientific application developers with a set of lightweight, mix-and-match components that the developers can combine as needed to provide holistic data management facilities for a particular application. Scientific application developers can use the Maitri components they need and omit the remainder, which simplifies application development, flattens the Maitri learning curve, and avoids unnecessary runtime overheads.

The Maitri framework consists of a set of standard, very narrow interfaces for format agnostic, loosely coupled libraries offering aspects of the functionality listed above (see Figure 1). Format independence is provided by Maitri's block manager module, which encapsulates all code that is specific to a particular scientific data format, and calls the appropriate scientific I/O library to read and write data in that format. The writer of a scientific tool (e.g., a visualization code) supplies the portions of the block manager code that are specific to the scientific I/O library and data schema in use. Implementations of the data management modules (indexing, buffering, concurrency control, query optimization and metadata management) adhering to the interfaces can be written by data

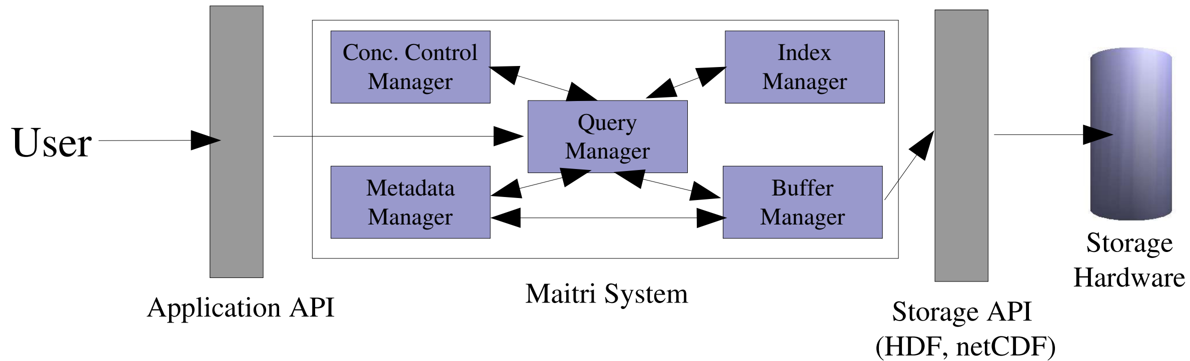


Figure 1.1. Architecture of the Maitri framework for scientific data management.

management experts and used in many different scientific tools.

A complete validation of all the Maitri modules and the overall framework is beyond the scope of this thesis. Our focus is on *Maitri's index manager*.

1.3 Indexing Scientific Data

Database indexing is a well studied field. Every commercial DBMS includes a version of B-trees and/or hash indexes, both B-trees and hash indexes are single dimensional indexes, which means that they are intended to improve lookup performance for a single highly selective query attribute. Scientific data in general **lack high selectivity attributes** and hence cannot utilize single dimensional indexes efficiently. For example consider Figure 1.2, which shows the distribution of values from the Moderate Resolution Imaging Spectroradiometer (MODIS) [1] Estimated Vegetation Index[2] (EVI) attribute used by scientists at the Department of Civil and Environment Engineering at the University of Illinois (henceforth referred to as the earth scientists). Earth scientists query over a subset of the wide 2,000-6,000 range of the EVI attribute. The figure shows that the selectivity of typical restrictions on EVI is low. The typical ranges for queries on other individual attributes are also not very selective; but when taken all together, the combined selectivity of the query restric-

[1] MODIS is an instrument aboard the Earth Observing Satellite launched by NASA. More details can be found at http://modis.gsfc.nasa.gov/.

[2] *Index* is a term used by environmental engineers to represent the density of vegetation at a point of observation.

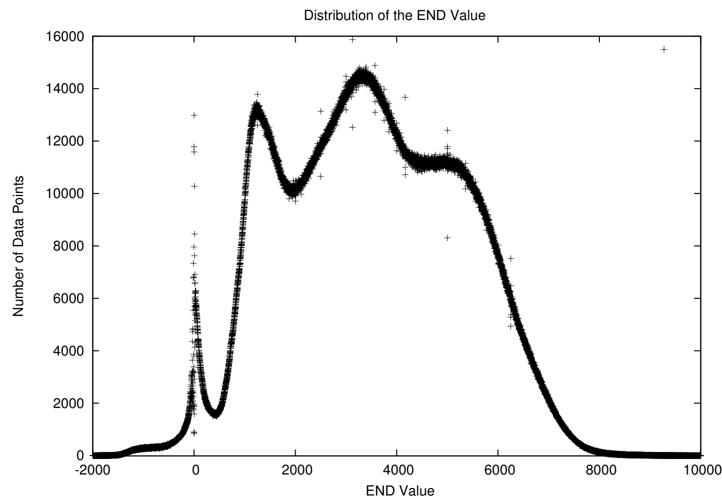


Figure 1.2. Data distribution for the MODIS Estimated Vegetation Index (EVI) attribute. EVI has low selectivity in queries, as most queries request a wide subrange of [2,000, 6,000].

tions is usually high enough for indexes to offer significant potential performance improvement over sequential scans. This is because under the independence assumption, the final selectivity is the product of individual selectivities.

One approach to using single dimensional indexes for scientific queries is to create such an index for each attribute, use them to find the points that satisfy one of the query restrictions and then merge the results. Most scientific queries involve conjunctions on multiple attributes, so the merging process involves intersecting large sets, which is very expensive. Hence we need an indexing structure that is inherently **multidimensional**.

Geospatial databases have long used R-trees for indexing their multidimensional data sets. While R-trees perform very well in multidimensional data sets with low dimensionality, they scale rather poorly as the number of dimensions gets larger [WOS04, Dep04]. Scientific data sets and queries on the other hand have **very high dimensionality**. For example, the MODIS data used by the earth scientists has two dimensions for space, one for time, 19 observational variables, and 22 for quality control. In a single query, the earth scientists want to include restrictions on the values of 4-36 of these attributes.

In some scientific applications, time and space restrictions together form a 3-4 attribute high-selectivity condition for many queries (e.g., “What was the vegetation density in a one kilometer radius around the city of Champaign on February 12, 2005”). R-trees (and other standard multi-dimensional indexes, such as KD trees, quad trees and oct trees) perform satisfactorily for these queries. However in a significant number of scientific applications, spatial and temporal attributes are not part of the query selection criteria. For example, the earth scientists might ask for all points in Illinois where the rainfall in 2000 was 50% greater than the rainfall in 1950. As another example, high energy physics observational data sets have 500 searchable attributes over billions of particles [Dep04], and a single query will restrict the values of up to 40 attributes. Standard multidimensional indexes do not perform well under such conditions.

Another difficulty is that standard multidimensional indexes are efficient only if the query restricts *all attributes* that they index. Scientific queries have **no fixed set of constrained attributes**. Because different queries constrain different attributes, creating an R-tree on any particular subset will not greatly improve performance. For example suppose we have a KD-tree on X and Y, and a query that restricts only X. At each tree level dedicated to Y, we must follow all child pointers out of that level, resulting in many more disk accesses than for a look up in a B-Tree on X.

Even the **set of attributes is not fixed** in scientific data sets. In many experiments, scientists integrate data from multiple sources to study interesting phenomena. For example, the earth scientists study the vegetation patterns in the Appalachian region. The scientists use their data mining tools to examine the data set and determine which variables affect the vegetation pattern. The scientists not only append data corresponding to new observation times, but also occasionally append data about new variables gathered from new sources. This behavior leads to changes in the set of attributes and dimensionality. An R-tree, KD tree or oct tree has to be completely rebuilt to accommodate a new variable.

Their ability to support efficient updates has helped B-trees, hash indexes to be very successful in the business world. Scientific data, however, is **read/append only**. Whether produced by a simulation or observation, scientific data is not changed once it is written. For example, the

scientists at the Center for Simulation of Advanced Rockets at the University of Illinois at Urbana-Champaign (referred to thereafter as the Rocket Center) run simulations of the behavior of rockets at various stages of combustion. These simulations may complete successfully or conclude with a rocket that explodes mid air. To allow scientists to analyze what went wrong in the case where the rocket explodes, the simulation periodically outputs the current values of important variables. These output values are later used to theorize the possible cause of explosion and to determine a way to prevent the explosion in a future run. Because data is not overwritten, indexes for scientific data do not need to support efficient update of individual entries.

1.4 Research Issues in Bitmap Indexes

Many of the requirements listed in Section 1.3 for indexing scientific data are well handled by bitmap indexes [SWS04, Sto01, WOS04]. More precisely:

- Bitmap indexes are suited for read only and append only data, such as scientific data.
- Each bitmap index is built for a single dimension (attribute, variable). An index lookup for a range restriction on that attribute provide a single result bit vector. Multiple bit vectors resulting from restricting several different attributes can be efficiently merged to provide a single result bit vector for a multidimensional query. Hence bitmap indexes are effective even if each query restricts a different subset of the attributes. Moreover, if a new attribute is added to the data set, we simply build a new index on the new attribute and use it in future queries.
- Bitmap indexes scale well to high dimensional queries [WOS04, Dep04].

These three factors suggest that bitmap indexes are well suited for indexing scientific data. However, we found that a few problems needed to be solved before bitmap indexes are ready for common use in scientific data management.

- Previous work has suggested the use of WAH compressed bitmap indexes for querying scientific data [WOS04, WOS06]. However, our experiments with real world data and queries showed that a standard WAH compressed bitmap index performed poorly on **large range queries** over **high cardinality attributes** (e.g., queries with a range restriction containing more than 200 values). We need improvements to provide efficient response to large range queries.

- Any indexing scheme for scientific data needs to scale up well on the **parallel platforms** that scientists use. These platforms are usually clusters of workstations with a single shared file system. We need a method to efficiently merge the indexes created by different processors during the creation phase, and to split the index onto separate processors for computation during the query phase.

- Large scientific data warehouses store 10s or 100s of terabytes of data. With such huge storage requirements, the truism that disk space is cheap no longer holds. Any additional disk space comes at a premium. Most large scientific data warehouses will **restrict the size of the index to a fraction of the size of the data**. We need a modification to bitmap indexes so that they can respond efficiently to queries despite such severe size restrictions.

- Bitmap indexes are efficient at retrieving the data points that satisfy the constraints provided by the scientists. Scientists, on the other hand, are looking for **regions of interest** in the dataset. Each such region is a collection of points that satisfy the query and are directly connected in the underlying mesh. We need an efficient method to combine the points returned by the bitmap indexes into regions.

1.5 Contributions of This Thesis

A complete validation of the Maitri framework and all its modules is the work of multiple PhD theses. In this thesis we focus on the indexing module for Maitri and propose a bitmap based

indexing scheme for retrieving regions of interest from large scientific data sets using small bitmap indexes . The main contributions of this thesis are:

- We propose Maitri, a format agnostic data management system for scientific data. Along with Maitri, we propose a large number of problems that need to be addressed to create a complete data management system for scientific data.
- We show both theoretically and experimentally that most previous optimizations suggested for bitmap indexes are not applicable when indexing scientific data.
- We propose multi-resolution bitmap indexes, which use multiple indexes built at different granularities of data to improve performance 10 fold on scientific queries, when compared to a single bitmap index. The multi-resolution bitmap index should occupy a maximum of twice as much space as an ordinary bitmap index.
- We provide an efficient approach to build bitmap indexes in parallel and then merge them into a single index. We also provide algorithms to efficiently split up the merged index onto separate processors for efficient look up in the future. Both our parallel creation and query algorithms scale up almost linearly with the number of processors.
- We studied the query behavior of scientists accessing large scientific data warehouses, and proposed a model of how scientists query.
- We introduced a variation on the algorithm for creating optimal bin boundaries [RSW05a], to allow us create *locally optimal bitmap indexes*. Our experiments show that on large query logs where the previous algorithms could not compute optimal bin boundaries in days, our algorithm was able to create the locally optimal bin boundaries in a matter of a few minutes.
- We showed how to build an adaptive multi-resolution bitmap index that are an arbitrary fraction of the size of the data. During the query phase, we create and drop additional auxiliary index structures to adapt to the query workload. We use aggressive caching of these

auxiliary indexes to take advantage of the temporal locality of the query values. Adaptive bitmap indexes that are 25% of the size of the indexed attribute proved more than six times speedup compared to a multi-resolution index of the same size. Adaptive bitmap indexes outperform indexes with no space limitation when the adaptive bitmap index is 50% of the size of the indexed attribute in some cases.

- We showed how to use compressed bitmap indexes and properties of scientific geometric meshes to retrieve regions of interest in sublinear time with respect to the number of points that satisfy the query. Our experiments show more than an order of magnitude improvement in performance over an approach that uses bitmap indexes and a traditional connected component labeling algorithm.

- We adapted the bitmap index technology developed in this thes to work with datasets stored in HDF5 scientific I/O library. We are releasing this version as prototype to be used with the HDF5 I/O library.

The problems studied in this thesis were inspired by close collaboration with scientists, and hence motivated by real world applications, and will benefit real world scientists. Our participation in the Center for Simulation of Advanced Rockets at UIUC and our connection to the earth sciences group at UIUC allowed us to study the properties of scientific data and propose bitmap indexes as a viable alternative. Our interactions provided us with the information that high cardinality attributes are common and that efficient handling of large range queries was critical to our effort. Our interactions with the rocket scientists showed us the importance of parallel platforms in scientific computation. From the scientists at the Department of Astronomy at UIUC who collaborate in the Dark Matter Survey, we learned about the size constraints for large scientific data warehouses. Our further interactions with the rocket scientists and the scientists at Princeton Plasma Center showed us that it is of vital importance for the scientists to look at regions rather than individual points. We have also closely collaborated with The HDF Group and are releasing our software so as to work with the HDF5 scientific I/O library.

1.6 Thesis Roadmap

The rest of the thesis is organized as follows. Chapter 2 provides an outline of the Maitri scientific data management framework [SMW05, STMW07]. After introducing various challenges facing scientific data management in general, we concentrate on indexing of scientific data in the rest of the thesis. In Chapter 3 we discuss the related work on indexing scientific data. Chapter 4 presents multi-resolution bitmap indexes [SMW06, SW07]. Chapter 5 presents the algorithm for parallelization of bitmap indexes [SW07]. In Chapter 6 we present adaptive multi-resolution indexes and evaluate their performance under strict space constraints. The last component of the thesis, combining points retrieved by indexes into regions of interest, is described in Chapter 7. Finally, we conclude the thesis in Chapter 8.

Chapter 2

Scientific Data Management: The Big Picture

Neither commercial DBMSs nor scientific I/O libraries have been a resounding success in managing the current deluge of scientific data. Because scientists are reluctant to abandon their chosen data formats and the associated I/O libraries, there is a need for a data management system that supports those formats while also providing improved scalability and, when needed, access to more sophisticated functionality for indexing, buffering, caching, concurrency control, metadata management, and querying.

In this chapter, we propose a framework within which to address these needs. The Maitri framework consists of a set of standard, very narrow interfaces for format agnostic, loosely coupled libraries offering aspects of the functionality listed above. Format independence is provided by Maitri's block manager module, which encapsulates all code that is specific to a particular scientific data format, and calls the appropriate scientific I/O library to read and write data in that format. The writer of a scientific tool (e.g., a visualization code) supplies the portions of the block manager code that are specific to the scientific I/O library and data schema in use. Implementations of modules adhering to the interfaces (e.g., a hash index manager) can be written by data management experts and used in many different scientific tools. We also point out many open problems in scientific data management that can be addressed within the Maitri framework.

2.1 Need for a Scientific Data Management System

Researchers have worked to adapt database technology to the needs of scientists [JGL+92, SCN+93]. The most successful of these efforts are probably SQL Server for the Sloan Digital Sky Survey

(SDSS) and Oracle for biological data. Yet close examination of the SDSS query log shows that scientists only use the relational part of SDSS to find data of potential interest. Once interesting data have been found using the Oracle RDBMS, the scientists download a less processed form of the data objects in a non-RDBMS format, go through the laborious process of cleaning them at home, and analyze them, all without a relational system. Similarly, there are loud rumblings of discontent with Oracle in the biology community.

Most of the reasons for this lack of success have been documented previously, along with the unmet needs and the special characteristics of scientific data and queries [Dep04, GLNS+05, MC99, NTC00, SMW06]. For example, failure has been attributed to the lack of support for arrays, need for considerable performance tuning expertise, lack of portability, expensive or non-existent parallel versions, monolithic architecture, use of proprietary file formats, and scientists' dislike of (learning) SQL. Perhaps surprisingly, performance is also often an issue. For example, a typical commercial data warehouse might hold 2 TB, while the Center for Simulation of Advanced Rockets at UIUC can generate that much data in a week. To quote Mike Folk, CEO of The HDF Group, "It's not unusual for scientists to start with relational databases, which often work well up to a point, but then people find that they don't scale to their problem sizes. This is when people start taking a look at what we're doing with HDF."

In response to the shortcomings of commercial databases, domain and computer scientists developed binary-file-based user-level libraries that store data in formats designed specifically for scientific data. These range from generic formats like HDF and netCDF to domain specific formats like ROOT for high energy physics and FITS for astronomy. These systems allow the scientists to store data with their associated metadata and access them efficiently through the associated libraries. However, the libraries' navigational model for access requires significant programming for each query. Further, the libraries' file-based approach has had the unanticipated side effect that as the data size increases, the number of files usually gets very large (e.g., an 11 GB data set at Rocket Center has 7152 files). Indexing and multi-file data retrieval are absent from these libraries, and scientific tool programmers expend considerable effort to provide multi-file metadata

management, buffering and caching facilities in their tools. To increase the challenge, scientists often need data from several sources, each adopting a different format and requiring use of a different API. The net result is that scientists have to double as data management experts, taking time better spent doing science.

Even if a vastly improved data management facility were made available to scientists, in many cases it is hard for them to adopt it. A number of scientific communities have already standardized on a data format and schema for their domains, and data that do not adhere to that format will not be sharable or be accessible by the tools developed by the community. Further, specific formats and schemas are often chosen for a large long-term project at its inception, such as while a new satellite is being built. Once a large amount of data has been collected or generated, and tools have been written to access that data in a certain format and schema, it is impractical to convert the data and rewrite the tools.

If neither commercial databases nor current scientific I/O libraries will able to handle the scientific data management challenges of the near future, and it will be very hard for scientists to migrate to new facilities, then how can we improve matters? We must allow scientists to use their choice of format and schema, while introducing new functionality and enhancing scalability. Intuitively, we could satisfy almost everyone by taking the high-level functionality of a DBMS, pasting it on top of a variety of scientific I/O libraries, and changing the look and feel of the query language. In theory, this can be done by enhancing the functionality of today's scientific I/O libraries, by extending ORDBMSs to work better with scientific data, or by introducing an entirely new approach that combines aspects of both worlds. We have adopted the third course of action, for the following reasons.

- Assumptions about the data storage format are built into DBMSs, making them incompatible with the vast amounts of legacy scientific data stored in an incompatible format. The removal of these assumptions will impact much of the DBMS (which makes it a great research topic). Further, commercial DBMSs generally use proprietary storage formats, which are unacceptable in most scientific domains.

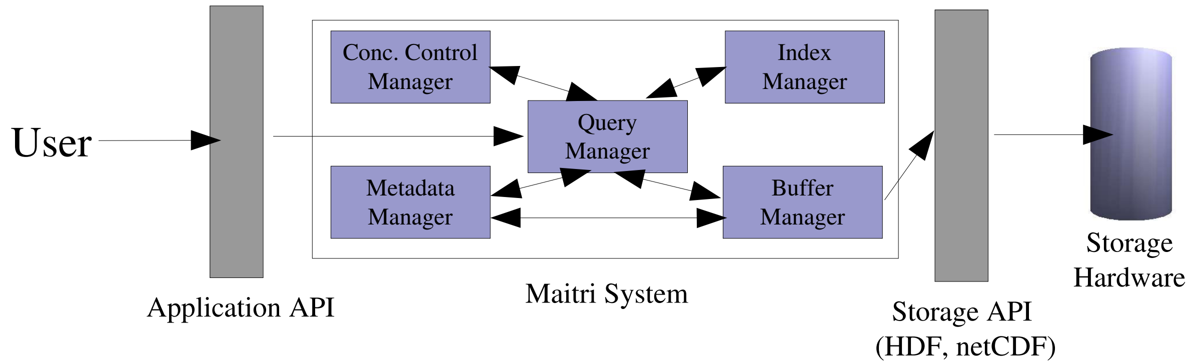


Figure 2.1. Architecture of the Maitri framework for scientific data management.

- Without *native* support for arrays in the storage model, access methods, and query language, DBMSs will find it hard to compete with scientific I/O libraries on performance and usability of large data repositories. While commercial DBMSs have made great strides in extensibility, and the interest in column-oriented storage and the decomposition storage model [CK85] is also a move in the right direction, the array data type remains a second class citizen in commercial DBMSs and it seems unlikely that this will change in the next decade.

- If the groups that maintain today's scientific I/O libraries each separately make significant enhancements in library functionality and performance, scientists will still have to learn a variety of different APIs to gather all the data they need from different sources. We can avoid this problem (and reduce the cost of the enhancements) by encapsulating the new functionality into modules that can be layered on top of multiple libraries.

To enhance a variety of scientific data storage formats with the high-level functionality of a DBMS, we propose the **Maitri** framework (Figure 2.1), whose most notable characteristics are as follows:

- *Loosely coupled, non-monolithic architecture.* Maitri's framework consists of very narrow interfaces for a block, buffer, index, metadata, and query manager. With few exceptions, a scientific tool writer can include the managers s/he needs and omit the remainder.

- *Many different module implementations.* The implementations behind Maitri interfaces can take many different forms, making the framework flexible, adaptable, and extensible. This extensibility allows a tool writer to pick the most appropriate implementations for the tool at hand, and upgrade them in the future without disturbing end users.

- *Format independence.* Format- and schema-dependent code resides only in Maitri's block manager module, which must be instantiated by the programmer of a scientific tool before that tool can use Maitri. A block manager implementation will typically use a pre-existing scientific I/O library to navigate through a schema in the native format of that library, and to read and write data in units that are meaningful and appropriate for the tool. In return for the effort of writing a block manager, the tool writer can take advantage of the high-level functionality that is provided by Maitri's other modules and is not available in the scientific I/O library. Further, the high-level functionality can be implemented once and used with a variety of different block managers, each corresponding to a different storage format.

- *Better scalability.* As their users will testify, the most popular scientific I/O libraries are significantly faster than traditional DBMSs for scanning typical large data repositories. The use of scientific I/O libraries as the lowest layer in the Maitri framework means that Maitri users can enjoy this same performance advantage. Where the libraries' performance falls short is in multi-file and selective data access; Maitri's buffer, index, and query manager interfaces and implementations are designed to overcome those limitations.

- *Better query interfaces.* Today's scientific tool query interfaces typically allow users to restrict the values of a handful of variables (e.g., temperature $>$ 0 AND pressure $<$ 100). Though scientists may not like SQL, in the long term a declarative query language is key to providing more powerful query capabilities in tool interfaces. Maitri is designed to allow the graceful adoption of declarative query languages for tools' selective data access.

2.2 Previous Scientific Data Management Solutions

The projects most closely related to Maitri are the USD [JGL+92], the Tioga system [SCN+93], and the ADR framework [KCC+01] . The goal of these projects is to provide a database-like interface to the scientific data. The USD, Tioga, and ADR projects are specifically designed for visualization applications, whereas the Maitri system is designed to be generic, though it can be tuned to perform well on visualization applications. The Tioga system is built on top of a full fledged database system, whereas Maitri tries to provide a more flexible architecture where the application programmer can pick and choose the components to be used. The smallest unit readable by both Maitri and the ADR framework is an aggregation of objects (blocks of data, rather than individual data points). While the ADR framework uses the unit to improve the performance, the purpose of the block object in the Maitri system is to provide format independence.

Other important projects that are dealing with data management facilities for large scale scientific data management are the SRB project (http://www.sdsc.edu/srb/index.php/Main_Page) and the openDAP infrastructure (http://www.opendap.org/). These architectures are both developed to allow scientists to collaborate and manage remote data. Though the main idea behind these systems is to provide a unified framework to access distributed data, they also provide simplified query interfaces to query data from these data sets, just like Maitri aims to. The major difference between these projects and Maitri is that the query interfaces provided by these systems are still file based, as they do not support queries over multiple files. Also these systems do not provide features like buffering and indexing to the user. Despite the availability of these frameworks for multiple data formats, these are not really format independent. Instead, a lot of manual effort has gone into making them compatible with multiple data formats. With Maitri's simplified Block Manager interface the effort required to write readers/writers for a new data format is much less.

Some researchers have taken a different approach to solving this problem. They have adopted a hybrid approach, where they store the metadata for the application in a traditional database, and the actual data in specialized storage systems. Choudhary et al. and No et al. use such an architecture,

in which the metadata is stored in a relational database, and use the mappings (of metadata to files which contain that data) from this system to allow users to query for various files [NTC00, CKN+99] containing the required data. The advantage of the mix-and-match approach of Maitri system is that the above proposed systems can easily form the metadata management module of the Maitri system.

Tools for storing and reading scientific data efficiently in particular storage formats, such as HDF and netCDF, allow very efficient access to scientific data. However, the interfaces provided by these systems leave a lot of data management burden on the application developer. Maitri is designed to work on top of these storage systems, with only minimal stub code from the application developer. The aim is for the end users and application developers to enjoy the efficiency of storage formats tailored for their kinds of applications, while taking advantage of buffering, indexing and querying facilities that Maitri provides in a layer atop of these storage libraries.

2.3 The Maitri Framework

As shown in Figure 2.1, the Maitri framework provides a scientist with the most important features of a database in a modular lightweight package. End-user scientists usually access data through visualization and analysis tools. Maitri is intended to work both in coordination with such tools and as a stand alone system. In essence, Maitri is a grey box between the scientific tool writer and the scientific I/O libraries being used to read and write the data.

When all of Maitri's modules are in use, the flow of control for processing a query is as follows. First, a scientific tool or user opens a data repository through a call to the **query manager**. Subsequently, the tool or user can forward a query to the query manager. The query manager asks the **metadata manager** for metadata about the repository and the attributes being queried (such as whether the attributes have indexes). The query manager plans how to execute the query; depending on the query manager's level of sophistication, this process can be trivial or very complex. During execution of the query, the query manager passes control to the **index manager** to execute

selected parts of the query. The query manager then uses the metadata manager to find out which data blocks to read to process the rest of the query.

Although scientific data is generally read-only once it is written, there are certain interesting cases where concurrency control is needed, as discussed in Section 2.4.6. Before reading or writing data or metadata that may be subject to conflicting access, the query manager can invoke the **concurrency control manager** to lock those objects and/or the data blocks containing them, using an appropriate lock mode defined in that particular instantiation of the concurrency control manager. The query manager calls the **buffer manager** to read the blocks into memory buffers efficiently. The buffer manager invokes the **block manager** to efficiently read blocks of data in a particular file format and schema.

Most of the Maitri managers are present in standard DBMSs. The most novel aspect of the Maitri architecture is the use of the block manager to encapsulate format-specific code, as discussed in more detail in the next section. The second novel aspect is in our intent that many scientific codes will not use all of the Maitri managers. If a tool writer chooses to omit the query manager (say), the writer is responsible for implementing the equivalent functionality directly in the tool. The tool can still access Maitri through any other instantiated module interface. For example, simulation codes are write-intensive. Their authors may choose to use just a buffer and block manager, as an easy way to get excellent overlap of writes and computation. The functionality that such codes require in terms of metadata management, a query language, and indexing will often be so trivial that it is simplest to code it directly in the tool. As another example, for archived, read-only data and metadata, there is no need to call a concurrency control manager. On the other hand, visualization tool authors will typically prefer to access data through a query manager. In this case, the author needs to know the Maitri query API and the query language supported by the particular implementation of the query manager being used. The writer does not need to learn a new query language for each storage format.

Figure 2.2 shows the interface that each Maitri module currently exports. For conciseness, we have omitted most function arguments and return values in the figure, but discuss the details in

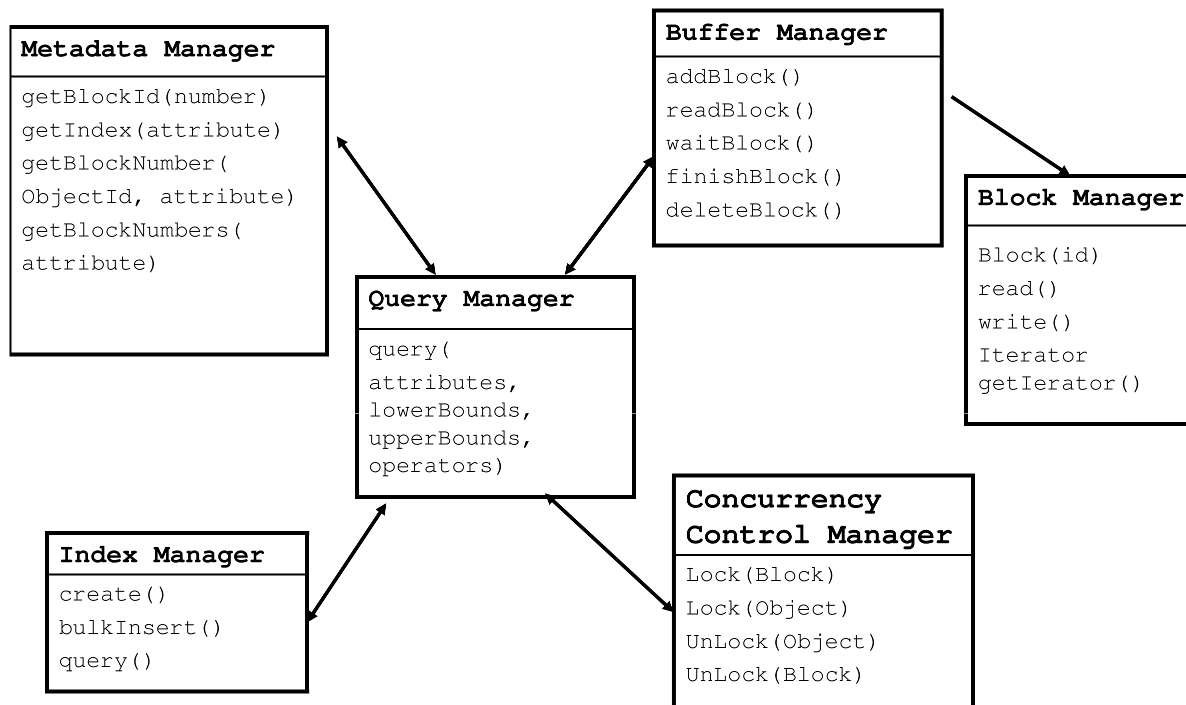


Figure 2.2. Maitri application programming interfaces.

later sections. The interfaces are the outcome of over a decade of working directly with scientists and with the HDF group; on the basis of this experience we claim that the narrow interfaces for the index, query, buffer, and block managers support a large proportion of the data management needs of the scientists we have worked with. Our work on the indexing, buffering, and block manager modules has been more extensive than the other modules, making us especially confident in the effectiveness and completeness of these APIs for today's typical scientific data management needs.

With our current implementations of the Maitri interfaces, the flow of control for a typical query is as follows. Consider the query "select velocity where temperature in [100, 500] & pressure in [5, 21]", over a database with an index on temperature. On receiving the query, the query manager asks the metadata manager to return links to indexes to temperature and pressure. The metadata manager returns a link to the index to temperature and returns a null for the pressure index. A call to the index manager then produces the IDs of the objects that have temperature between 100 and 500. The query manager uses the metadata manager to find out which buffers to read to evaluate the

constraint on pressure. The query manager also uses the metadata manager to determine whether these buffers will also provide values of velocity (in practice, scientists will have stored the velocity and the pressure values separately from one another). The query manager then asks the buffer manager to read the required blocks in a background thread. The query manager tells the buffer manager which blocks it will be reading in the future, so the buffer manager can prefetch them while the query manager is working on the pressure constraint for another block. The buffer manager calls the block manager to read the pressure and temperature data in their native formats.

The interfaces in Figure 2.2 omit 90% of the calls that one might find in the API of a modern scientific I/O library or DBMS; where has all the complexity gone? Some of the complexity lies in the arguments passed to the managers. For example, the query manager accepts strings in an arbitrary query language, and that language can be as simple or complex as needed. Some of the complexity is hidden inside the managers. For example, a query manager might have a sophisticated optimizer; a concurrency manager might have many different lock modes; and a block manager may have to make many calls to the underlying data library to instantiate a data block.

The interfaces allow an inept tool designer to really make a mess of things. This decision is deliberate. Without Maitri, tool designers are responsible for all aspects of data management except those encapsulated by the I/O libraries they use. Maitri does not exempt them from all of these responsibilities. Instead, our intent is that Maitri should allow scientific programmers to collaboratively build up layers of enhanced functionality that can be shared across groups and reused in many different contexts, so that tool writers will often be able to manage data at a higher level of abstraction and with greater power than they can without Maitri. Maitri does not free tool writers from all performance tuning responsibilities, either. The responsibility for performance tuning will lie primarily with the writers of block managers.

2.4 Challenges in Building Maitri Modules

Creating a loosely-coupled format-agnostic data management for large scientific data sets leads to a large number of challenges that need to be solved. In this section we talk about some of the challenges that we have solved and enumerate a large number of challenges that still need to be solved before Maitri can become fully operational.

2.4.1 Block Manager

In traditional databases, a block refers to a logically contiguous portion of a file on disk. In Maitri, a block refers to a set of *logically* collocated objects that are to be read and written as a unit [MWN+04]. Only the block manager can read, write, parse, and understand the internal structure of blocks—and those are its chief responsibilities. The block manager is format dependent, and will usually be schema dependent as well; this sacrifice allows the rest of the managers to be written to be format independent. The block manager interface defines the following methods:

```
INTERFACE BLOCK {
        BLOCK(BlockID id);
        READ();
        WRITE();
        RECORD GETINSTANTIATIONS(Attributes);
        RECORD GETINSTANTIATIONS();
  }
CLASS BLOCKID {
        Hashtable storage_info;
        int id;
        BLOCKID(int id);
        BLOCKID(Hashtable table);
        READ();
        WRITE();
  }
```

A BlockID object (described below) must be initialized before the corresponding block can be read or written using the block manager's READ and WRITE methods. Once a block has been read

into memory, the block manager's GETINSTANTIATIONS method can be called to parse and extract part or all of the data in the block, and place the data in newly created Maitri RECORD objects. The developer of a tool that will use Maitri must implement the GETINSTANTIATIONS method. Each RECORD object contains the values of the attributes for a particular object that resides in the fetched block.

A BLOCKID object contains the metadata that the READ and WRITE methods need to read/write a particular block from/to disk. The BLOCKID contains an integer identifier that uniquely identifies the metadata related to the block. The remainder of the BLOCKID contains the metadata required by the block READ and WRITE methods supplied by the tool developer; this implementation-dependent metadata resides in the BLOCKID's generic "table" data structure, which is currently implemented as an extensible hash table. The BLOCKID's metadata is initially created when a data set is bulk loaded into Maitri for the first time (by a user-supplied routine in the database initializer) or when new data is added to a pre-existing database. A subsequent call to the BLOCKID's WRITE method will cause the metadata manager to store the metadata permanently on disk. Subsequent runs can initialize a BLOCKID object with a unique identifier, and the metadata manager uses the unique ID to find or generate the metadata to read the block.

While the block manager looks relatively simple, a good choice of definition of blocks is critical to good runtime performance. For example, if the data set is stored in many individual 2 MB files, then a block should be defined as an entire file. On the other hand, if each file is 2 GB, it is likely that a different definition of a block is needed for good performance. Without Maitri, scientific programmers are already implicitly saddled with the task of block definition (i.e., deciding how much data to fetch at a time), so in a sense we can hardly make things worse than they already are. However, without good prefetch and write-behind facilities, scientific programmers could never hope to achieve top performance. With the buffer manager's prefetching and write-behind facilities, it is time to raise expectations. At a minimum, we need to help the programmer understand how different implementations of the block manager can change the performance of the tool being developed. Thus an excellent topic for future research is the development of guidelines

and automated facilities that will help tool programmers to tune their block managers, i.e., to choose good block definitions.

2.4.2 Query Manager

Our current query manager implementation supports a very simple query language. All data entry is handled by bulk inserts, and all query conditions take the form of DNF range restrictions on attributes. The query optimizer generates an execution plan that uses all indexes available for restricted attributes, and then performs sequential scans to evaluate the restrictions on the remaining attributes. The query interface consists of a single function query with four parameters: list of attributes, list of lower bounds, list of upper bounds and the number of attributes in the query. While this is sufficient for a surprisingly large proportion of scientists' needs, clearly one can do much better. In general, this area consists of uncharted waters packed with interesting research topics.

As elegantly pointed out by Musick [MC99] and many other researchers, scientists' data models are not relational. Scientists are not usually familiar with data modeling concepts such as schemas, entities, attributes, and so on, and have a documented resistance to learning SQL. On the other hand, scientists are happy to fill out query forms and may be receptive to natural language interfaces. Researchers have recently reported success in weaning scientists onto a natural language front end pasted onto an XML data model [LYJ06]. Previous decades of attempts at database user interface research have resulted in little more than QBE to our community's credit, and efforts to develop natural-language interfaces have always fallen flat in the past—so it is especially exciting to see these recent small successes. Perhaps the database research community can finally make progress on natural-language interfaces, by concentrating on meeting the needs of scientists.

Researchers have also developed a way to help scientists automatically generate form-based front ends for tools [JJ06]. Major scientific data formats such as HDF can already export their data as XML, and the flexibility of XML suggests that it might be an interlingua that one could use as the basis of a generic query language for scientific data. To verify this hypothesis, an open research challenge is how to map between the XML model exported by a scientific I/O library and

the representation of that data in its native format, in such a way that queries can be optimized and then executed efficiently in the native format.

The architecture of Maitri allows the system to pipeline execution, interleave optimization with execution, or pursue many other execution strategies. For example, while we are still using an index to process range restrictions on a particular set of attributes (phase 1), we can ask the buffer manager to simultaneously prefetch data (phase 2) that will be required to evaluate restrictions on the remaining attributes (phase 3), such that only blocks containing objects that satisfy phase 1 are present in memory when we work on phase 3. The decision on what data to prefetch and how to prefetch them can be based on feedback from the early stages of the index lookup, e.g., regarding the selectivity of the restrictions. There are many interesting new possibilities for research in query optimization here, because many assumptions of traditional query optimization are violated in scientific data: we have interesting new indexes (B-trees and hash tables are not very useful for scientific data), scientific data formats generally use a column-oriented store rather than a row-oriented store, and the concept of a join, requiring significant computation, is much more complex in scientific data than in business.

2.4.3 Metadata Manager

Most scientific data sets require managing a large amount of metadata. For example, any file in the NASA EOS data set stores a large amount of associated metadata using the attribute feature of HDF. The metadata ranges from the date and time the file was created, through the algorithm number used to generate the data, to the longitudes and latitude of the pixels. Much of metadata is common across many EOS files and are replicated because the file is treated as an independent individual unit. In many cases no analysis can be done without the metadata. For example, without knowing the latitude and longitude of the pixels the entire data set is useless. Also, a lot of the data is not stored in the format it will be used, instead it is transformed using various formulas for

ease of storage. Such alterations are also recorded in each file, for each dataset [1]. If we have an integrated data management system rather than just the file based system, we can factor out the common metadata, avoiding redundant storage. Even the metadata that is specific to individual files can be stored separately, allowing for faster access as not all the files need to be opened to see what they contain. Since the amount of metadata is usually much smaller than the actual data, factoring out of the metadata does not introduce too much inefficiency. Mappings of blocks to the exact regions on disks adds one more layer of metadata that needs to be managed.

Many researchers have addressed metadata management for scientific data [NTC00, PAW+05, CKN+99]. Most of these researchers use a database engine to store the metadata, with the actual data residing in scientific-format binary files. This guarantees fast file I/O for data and sophisticated data management capabilities for metadata. However, as scientists move between platforms, they either must have their metadata DBMS available on all platforms (costly and annoying) or else use remote calls to access the metadata DBMS (very slow at run time). Maitri's metadata manager is loosely coupled with the other Maitri modules, allowing easy integration new modules into Maitri; the metadata could be stored in a DBMS, special-format binary files, simple ASCII files (most portable), or even as mathematical formulas or lookup tables to map a block identifier to the metadata required to read, write, and parse the associated block.

Our current implementation of the Maitri metadata manager stores the metadata in an ASCII file. The metadata manager API is shown below:

```
INTERFACE BLOCK {
    getIndex(attribute)
    getBlockNumbers(attribute)
    getBlockNumber(ObjectId, attribute)
    getBlockId(number)
}
```

The getIndex method returns a handler to the index for the attribute, if an index exists. In case no index exists we use the getBlockNumbers(attribute) function to get a list of block numbers that

[1] *dataset* is an HDF term used to denote a contiguous array, not to be confused with *data set* which is a general term used to denote a collection of data

store the required data. In case we do have an index we use the index to retrieve a list of object IDs for the objects that satisfy the query constraint using the index manager. The getBlockNumber(objectID, attribute) then translates the object IDs into block numbers. The getBlockId method is then used to retrieve the BlockID object required by the block manager for the corresponding block numbers. The other metadata related to the query can be queried using the query interface just like any other data.

2.4.4 Index Manager

Researchers have pointed out that traditional indexes such as B trees, hash tables, and even R trees, KD trees, and oct trees do not work very well with scientific data; bitmap indexes are the leading proposal for a general-purpose alternative [Dep04, WOS04, SMW06, Wu99, WB98]. While Maitri's modular architecture supports the use of many different kinds of indexes, we have focused on enhanced bitmap indexes. These *indexing schemes* form the focus of this thesis.

2.4.5 Buffer Manager

Maitri's buffer manager is a modified and extended version of the Godiva buffer manager [MWN+04], which uses a relatively simple API to allow the programmer to specify blocks of data that can be read and stored in memory together. The architecture of Godiva is shown in Figure 2.3. Godiva also allows the application developer to specify prefetching and buffering hints, which Godiva will use in deciding when to fetch/write a block and when to evict a block from its cache. Godiva fetches and writes blocks in a background thread. In Godiva, the block manager is an integral part of the buffer manager; in Maitri, the block manager is an independent entity and the buffer manager is format independent. All Maitri modules must call the appropriate block manager when they need to read or write blocks on disk, and when they need to extract information from blocks in memory.

In order to read any block, the query manager first needs to initialize that particular block.

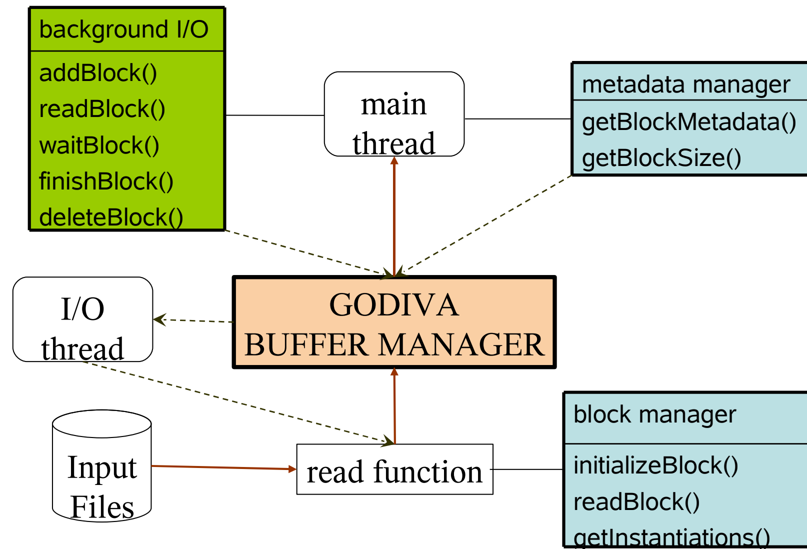


Figure 2.3. The architecture of the buffer manager.

Then using the READ method (provided in the block manager by the scientific tool writer) the query manager reads the data from the disk. Godiva provides an API which allows the scientific tool writer to provide hints to the buffer manager on how to most efficiently read the required data. These methods are:

- ADDBLOCK(int BlockID) - This method adds the block with the given block ID to the list of active blocks. This means that the block gets enqueued to be read.
- READBLOCK(int BlockID) - This method is a blocking call that forces the particular block to be read immediately.
- WAITBLOCK(int BlockID) - This method is a blocking call that causes the thread calling it to wait till the block is read into memory in the order in which it was added to the active blocks list.
- FINISHBLOCK(int BlockID) - This method removes the block from the list of active blocks

but still keeps the block in memory for later use. A block not in the active blocks list will be removed if blocks need to be evicted because of memory size limits.

- DELETEBLOCK(int BlockID) - This method removes the block from the list of active blocks and also deletes the block. This method should be used only when the user is absolutely sure that the block will not be reused.

While the scientific tool processes the data already read in, the background I/O thread reads the blocks registered using the addBlock() method. A detailed description of Godiva and a performance study of Godiva with visualization tools can be found in [MWN$^+$04].

2.4.6 Concurrency Manager

As with traditional databases, assuming serial access to data is impractical. Interleaving of data accesses from different users is critical for good performance. While we have said before that scientific data is mostly read/append only, the potential write-write conflicts. In cases where different processors write to different files there is no overlap of data items being written, and hence no inconsistency can be introduced into the data set. This solution creates huge number of files, leading to a metadata management nightmare. To overcome such problems, scientific codes usually allow different processors to output to the same file. To facilitate this process, different regions of the file are allocated to different processors statically allowing writes without any conflict. In cases where dynamic allocation of file space is required, inconsistencies might occur if orderings of writes are improper. For example, two different processors could be allocated the same location in the file causing one of the data to be lost. Such dynamic allocation of space is a requirement when processors write out compressed arrays. Since the size of the compressed array can not be known a priori, allocation of file space to different processors has to be dynamic.

While write-write conflicts happen mostly in highly parallel environments, read-write conflicts can happen even in sequential environments because of appends done to the data. If the scientist is running an online visualization/analysis tool to analyze the data as it comes, concurrent reads and

writes to the same location can cause inconsistency in the analysis.

Our current Maitri implementation consists of a simple API that uses locks at both object and block level to restrict access to the data.

2.5 Realizing the Maitri Vision

In this chapter, we have proposed Maitri, a data management system to manage large scale scientific data. Maitri aims at providing the scientists with a set of lightweight, mix-and-match, data-format-independent, easy-to-use set of libraries that will provide the scientists with the advantages of commercial databases (indexing support, buffering support, metadata management, and cheap concurrency control, declarative query interface), with none of its inappropriate features (expensive logging and concurrency control, proprietary format, lack of portability and lack of efficient access to array formats), at the cost of writing a small amount of code to make the system format independent.

The task of creating Maitri is far from complete. We still do not know what a query language for such a data management system should look like. While we feel that natural language interface is ideal for such a system, and have seen working prototypes of natural language front ends for scientists, actual scientists need to accept it. Further, our query optimizer is far from complete. Moreover, we feel work can be done at develop a better consistency model for scientific data sets. Also, at this point Maitri leaves the job of deciding the block sizes and layouts to the scientists. While it is appropriate for them to make the final decisions, it would be helpful for them to have many more guidelines on how to choose such blocks.

While Maitri and its index manager both have a large number of open research problems, in the rest of the thesis we concentrate on retrieving regions of interest from large scientific datasets using small bitmap indexes.

Chapter 3

Related Work

3.1 Traditional Indexing Schemes

B-trees and hashing have stood the test of time as the premier disk-based indexing approaches for relational databases. These approaches index a single dimension (attribute) of data in each index. In business-oriented transaction-processing queries and updates, often one attribute has a very high selectivity. Thus very good runtime performance is obtained by indexing these high selectivity attributes and using one of them as the first selection criterion when performing the query. In contrast, scientific data does not usually have such high-selectivity individual attributes; instead, a combination of conditions on multiple attributes makes the selectivity high enough for indexes to be useful. While some scientific queries do use time and space as high selectivity attributes, many do not. For example, high-energy physics produces data sets with 500 searchable attributes over billions of particles [Dep04], where no individual attribute has much discriminatory power and a typical query might impose conditions on 40 attributes. In these scenarios, a multidimensional index is needed for good lookup performance.

The need to index multiple attributes in a single index eliminates B-trees and hashing from consideration. Traditional multidimensional indexing schemes such as R-trees [Gut84], KD-trees [GG98], quad trees [GG98], oct trees [GG98], and so on are effective for indexing a handful of dimensions simultaneously (e.g., three geographic coordinates plus a time dimension). Unfortunately, none of these schemes scales well once more than 15 dimensions are to be indexed [Dep04], and none of them is very good at handling queries that impose conditions on only (say) half of the attributes in the index. For example, if an R-tree is built on three dimensions, and the query re-

stricts just two of those dimensions, then the number of paths in the tree that must be traversed is much larger than when all three attributes are restricted. These indexes are also relatively bulky, sharing the characteristic of B-trees that they are likely to be three to four times larger than the attribute data being indexed [WOS04].

Scientific queries typically involve identification of regions of interest, in a two step process [Dep04]. The first step involves searching for objects that satisfy user-defined criteria; the second step involves growing the objects into regions by identifying nearby objects that also satisfy the criteria. For example, one might start by querying for all the points in a rocket that experience very high temperature during a span of time, then consolidate those points to form the hot rocket regions. With traditional indexes, to achieve best performance, one orders the data on disk according to the values of the most important indexed attribute. This is not appropriate when there is no single primary selection attribute, as in scientific data. Due to the order in which data are gathered or generated, scientists typically store their data so that spatially close objects are located close together in the data set; this makes the region growing step faster.

3.2 Bitmap Indexes

The characteristics discussed in Section 3.1 have led researchers to suggest the use of bitmap indexes for scientific data [WOS04, SWS04, Sto01].

Bitmap indexes have been shown to give good performance for high dimensionality queries in large data warehouses [O'N87, Wu99, WB98]. Bitmap indexes are typically smaller in size than B-trees and other traditional indexes over the same amount of data [WOS04]. Bitmap indexes can be very efficiently updated when new data are appended to the database (e.g., when a new day of satellite observations arrives) by appending to the bit vectors. If a new attribute is added, it is easy to create a new bitmap index for that attribute. Bitmap indexes are known to be poor performers for update operations other than appending new data, but that is not a problem for scientific data. Bitmap indexes are effective for queries that restrict an arbitrary subset of the indexed attributes,

as well as for queries that restrict all indexed attributes. Since there is no single primary selection attribute for most scientific data, it is important that bitmap indexes function very well as secondary indexes. This is in contrast to traditional indexes, which perform best when used as primary access methods (i.e., when their data are ordered on disk according to the values of the most important indexed attribute).

Bitmap indexes store the index information as a sequence of bit vectors and use logical boolean operators to respond to queries. Patrick O'Neil is credited with the popularization of these indexes with his work on the Model 204 Data Management System [O'N87, OQ97]. In a conventional B-tree index, each distinct attribute value v is associated with a list of record identifiers (called the RID-list) of all the records that are associated with the attribute value v. In a bitmap index, the RID-list is replaced by a bit vector representing the RID-list. Figure 3.1(a) shows an example of a bitmap index over attribute A of the database. The size of each bit vector is equal to the number of records in the database, and the kth bit in each bit vector corresponds to the kth record in the database. In a simple unencoded and uncompressed bit vector for attribute value v, the kth value is set to 1 if the kth record in the database has value v for that particular attribute. For example, with such an index, answering multidimensional queries becomes a series of bitwise AND, OR, and NOT operations on bit vectors, which can be done very fast using low level instructions.

3.3 Previous Work on Bitmap Indexes

The data set in Figure 3.1(a) contains just 12 objects and 9 different values. A real scientific data set can have billions of objects and even more distinct possible values, due to the presence of floating point data. For example, the earth scientists' climate data contains 1.2096×10^8 objects for the Appalachian mountain region, each with 52 attributes. Each uncompressed bit vector in a bitmap index for this data is approximately 14 MB in size. As one example, the EVI attribute is stored as short integers ranging from -2,000 to 10,000, giving us 12,000 possible values and a total uncompressed bitmap index size of 167.01 GB for this attribute, while the data size for EVI is only

OID	A	B^0	B^1	B^2	B^3	B^4	B^5	B^7	B^7	B^8
1	7	0	0	0	0	0	0	0	1	0
2	3	0	0	0	1	0	0	0	0	0
3	1	0	1	0	0	0	0	0	0	0
4	2	0	0	1	0	0	0	0	0	0
5	6	0	0	0	0	0	0	1	0	0
6	2	0	0	1	0	0	0	0	0	0
7	0	1	0	0	0	0	0	0	0	0
8	5	0	0	0	0	0	1	0	0	0
9	7	0	0	0	0	0	0	0	1	0
10	8	0	0	0	0	0	0	0	0	1
11	8	0	0	0	0	0	0	0	0	1
12	4	0	0	0	0	1	0	0	0	0

(a) A bitmap index for attribute A

OID	A	I^0	I^1	I^2	I^3	I^4
1	7	0	0	0	1	1
2	3	1	1	1	1	0
3	1	1	1	0	0	0
4	2	1	1	1	0	0
5	6	0	0	1	1	1
6	2	1	1	1	0	0
7	0	1	0	0	0	0
8	5	0	1	1	1	1
9	7	0	0	0	1	1
10	8	0	0	0	0	1
11	8	0	0	0	0	1
12	4	1	1	1	1	1

(b) Interval encoding for a bitmap index

OID	A	B^0	B^1	B^2	B^3	B^4	B^5	B^7	B^7
1	7	0	0	0	0	0	0	0	1
2	3	0	0	0	1	1	1	1	1
3	1	0	1	1	1	1	1	1	1
4	2	0	0	1	1	1	1	1	1
5	6	0	0	0	0	0	0	1	1
6	2	0	0	1	1	1	1	1	1
7	0	1	1	1	1	1	1	1	1
8	5	0	0	0	0	0	1	1	1
9	7	0	0	0	0	0	0	0	1
10	8	0	0	0	0	0	0	0	0
11	8	0	0	0	0	0	0	0	0
12	4	0	0	0	0	1	1	1	1

(c) Range encoding for a bitmap index

Figure 3.1. Bitmap index basics.

228.92 MB (the size of an array that contains all the EVI values and nothing else). To address this problem, researchers have proposed ways to reduce the size of each bit vector, the number of bit vectors that must be read to answer a query, and the total number of bit vectors—via compression, clever encodings, and binning, respectively. We discuss each of these below.

Interval Encoding. The most interesting encoding for our purposes is *interval encoding* [CI98, CI99], which allows us to answer any range or equality query on one attribute by reading a maximum of two bit vectors. Figure 3.1(b) shows the interval encoded bitmap for the example in Figure 3.1(a). In this type of encoding, every bit vector represents a range of $\lceil \frac{C}{2} \rceil$ values, rather than a single value, where C is the cardinality of the attribute. For example, in Figure 3.1(b), I^0 represents the range $[0, 4]$, I^1 represents the range $[1, 5]$, and so on. Any query with range greater than four can be answered by an OR of two bitmaps, and any query with a range smaller than four can be answered by an AND of two bitmaps.

Interval encoding has been shown to be optimal for certain kinds of queries [CI99, CI98], but many other encodings have been proposed in the past. Perhaps the best known of these is range encoding [OQ97, WLO$^+$85, WLO$^+$86], shown in Figure 3.1(c). Under range encoding, the vth bit vector represents all attribute values in the range [0, v]. Range encoding allows us to find all the objects with an attribute value within a given range by accessing at most two bit vectors, assuming that we store as many bit vectors as there are attribute values.

Binning. We can reduce the size of bitmap indexes by having each bit vector represent a bin of values. This considerably reduces the number of bitmaps that need to be stored. In the absence of range or interval encoding, binning also substantially reduces the number of bitmaps that must be brought into memory to answer a typical range query. Yet regardless of whether we use range or interval encoding, the result of the Boolean operations performed on binned bit vectors may contain *false positives*. Figure 3.2 illustrates this situation for three range queries over a single attribute. The range restrictions specified in query $q1$ do not exactly coincide with the bin boundaries. The *candidate bins*—the first and last bins that overlap the range—can contain objects that fall outside the specified range for that attribute, introducing false positives into the

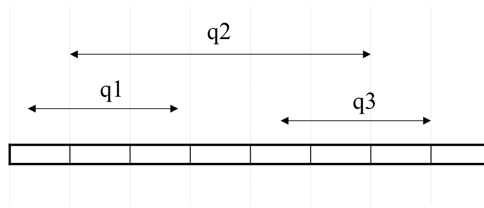


Figure 3.2. Using a bitmap index with 8 bins to respond to queries.

query response. The query processor must fetch the objects in the candidate bins and remove those that fall outside the target range. Query $q1$ has two bins containing false positives that need to be removed, while $q3$ has just one such bin. Query $q2$, on the other hand, has no false positives, and hence can be efficiently answered with the bitmap index, without fetching the indexed objects into memory.

Range encoding can be extended to ensure that the index still works efficiently with binned data, though we will need to access the objects in the candidate bins to make sure that they satisfy the query conditions [Kou00, Sto01, SDHS00, WY98].

The work in [Sto01] was extended in [SWS04] to offer three different strategies for checking for false positives when a query condition involves more than one dimension and the bit vector corresponds to a bin of data values. In the first strategy, the entire range query is executed for all the dimensions, and then false positives are removed from all the dimensions by reading candidate vectors in each dimension, one by one. In the second strategy, the candidates for each dimension are checked for false positives before merging the results from the different dimensions. Strategy three interleaves the first two strategies. First the entire results from different dimensions are merged, and then for each dimension, the candidate vector is modified by removing objects that are known to be false positives in other dimensions. The authors evaluate all three strategies and show that the third one is the best overall. The authors of this technique also offer a framework for deciding when it is better to use a generic range encoded bitmap index and when it is better to perform a sequential scan of all the data.

Other researchers have improved bitmap indexing by basing binning strategies on the data and query distributions [RSW05a, RSW05b]. This work uses the data distribution and the distribution of queries to reduce the total I/O cost associated with checking candidate bit vectors. In [RSW05a], the authors use a dynamic programming method to incorporate the information learned from answering previous queries into the decision of where to place bin boundaries for a particular attribute. They provide a function that estimates the I/O cost to answer queries using different choices of bin boundaries, and optimize this function to get the optimal binning. A more detailed discussion of the technique can be found in Section 6.2. In [RSW05b], the authors extend their technique to solve the problem of generating optimal bin boundaries when the queries are multidimensional.

Rotem et al. evaluated the cost of removing false positives, using 378 MB of data consisting of 10^8 synthetic objects with a single attribute, whose values were chosen according to a Zipf distribution [RSW05a]. They built bitmap indexes using 1000 bins, with bin boundaries selected using an equidepth, equiwidth, or their own optimal strategy. Rotem et al. evaluated the indexes against 100 single-attribute queries with randomly selected endpoints.

When the Zipf distribution parameters were set to produce a uniform distribution $(z = 0)$, over 80% of query workload processing time was spent on checks for false positives, no matter what binning strategy was used. When the Zipf distribution was more skewed $(z = .5)$, approximately 70% of total workload processing time was spent in candidate checking under optimal binning, significantly better than with equidepth or equiwidth binning. With a classic Zipf distribution $(z = 1)$, the candidate check time still dominated with equiwidth and equidepth bins, while with optimal binning the candidate check time dropped to 55% of total processing time. Altogether, these experiments showed that false positive removal is very expensive, no matter how the data values are distributed and bin boundaries are placed. This is true even when the data are in a single binary file that is read into memory into its entirety, as was the case in the experiments of [RSW05a]. Those experiments used synthetic data stored in raw form, i.e., without any of the standard self-describing formats for scientific data. The removal of false positives will be even

more costly in the decomposition storage model and multi-file data stores prevalent in real-world scientific data, because the false positives may be spread over many different files. Because each attribute is typically stored in a separate file, each check for a false positive must open as many files as the number of attributes in the range restrictions. Further, opening one of these files is an expensive proposition, requiring metadata management activities similar to those needed to open a relational database. Thus opening and reading each of hundreds of relevant files can severely degrade query performance.

Compression. Compression is an important factor in minimizing the size of bitmap indexes. Standard compression techniques such as gzip and bzip2 reduce the amount of disk space required, but do not change the total amount of memory needed to process the data, and increase the time for processing by adding the cost of decompression [AYJ00, Joh99, JL01, WOSN02]. Bit based, byte based, and word based compression methods have been developed to avoid the latter pitfalls. Run-length encoding is a common type of bit based compression. PackBits and Byte-aligned Bitmap Code (BBC) [Ant95] are examples of byte based compression. Hybrid run-length encoding and word-aligned hybrid (WAH) [WOS02] run-length encoding are examples of word based compression techniques. A detailed study of these techniques can be found elsewhere [WOSN02]. A number of other bitmap compression techniques have been suggested for specific problems [Cha03, PN06, MMNM03, WKA05].

Overall, WAH compression appears to be the best choice for compressing scientific bitmap indexes [WOS02], though research is continuing in this area and better alternatives may be found [ACFT06]. The worst case size of a WAH-compressed bitmap index is $8N + 8B$ bytes (assuming uniform distribution of attribute values, which is the worst case), where N is the number of bits set to 1 in the vectors (which, in straightforward applications of bitmap indexes, is equal to the number of objects being indexed), and B is the number of bit vectors (bins). In contrast, in commercial database systems, a B-tree index usually occupies $10N$ to $15N$ bytes ($3N - 4N$ words), which is larger than the theoretical maximum for a WAH-compressed bitmap index [WOS04, WOS06]. Although the size of a tree-based index can be reduced if the index is constructed in a batch

operation, this prevents quick incremental indexing of newly arriving data.

Our work in this thesis builds on WAH compression, which is a combination of run-length encoding and uncompressed bitmaps. A WAH-compressed bitmap contains two kinds of words, *literal* and *fill* words. The word size should be chosen to give good in-memory performance on today's platforms, e.g., 32 bits. In this case, the uncompressed bitmap is divided into 31-bit segments. The most significant bit of each WAH-compressed word is used to differentiate between literal and fill words. A literal word contains one segment of the uncompressed bit vector verbatim, while a fill word represents a sequence of contiguous segments in the uncompressed bitmap that either have all 0s or all 1s. The body of a fill word tells us the number of contiguous segments that are all 0s or all 1s, as well as their value (0 or 1). WAH compression is *word aligned*, i.e., the position of the bits in a literal word is the same as in the corresponding segment from the uncompressed bit vector.

Previous research has found that bitmap indexes are very effective even for high cardinality attributes, as long as the indexes are properly encoded, binned, and compressed, and an appropriate evaluation strategy is used while executing the query [Kou00, Sto01, SDHS00, WOS04, WY98]. In our experiments with rocket science and earth science data, however, we found that bitmap indexes performed rather poorly for large range queries over high cardinality attributes (such as the EVI query discussed earlier). As explained in detail in Section 4.2, previously proposed versions of bitmap indexes either required too much storage or were too slow in answering our scientists' queries.

3.4 Retrieving Regions of Interest

A typical approach to retrieving regions of interest in a scientific data set has two phases: constraint application and region growing. The constraint application phase involves selecting of some of the objects in the data set based on the range constraints in the query. The region growing phase merges neighboring objects that satisfy the range constraint into regions of interest.

The problem of retrieving regions of interest from large spatial scientific data sets (images, meshes) has been examined by the image analysis, visualization and database communities. A typical image analysis approach works with individual pixels of an image that has regions of interest [DST92, FG96]. Once retrieved, individual pixels are then merged together to form regions of interest. A typical visualization approach utilizes an oct-tree or an R-tree on the spatial coordinates, and hence handles the object consolidation phase efficiently [HJ05, LHJ99, SW98]. But this approach is not as efficient in the constraint application phase. Typical database techniques of indexing individual points to speed up the constraint application phase suffer in the object consolidation phase [SJ02, PAL^{+}06].

All of the above mentioned techniques solve the problem of retrieving regions of interest in $O(N)$, where N is the number of points in the retrieved regions. Our work is based on previous methods [SSBW05, SSWB05] that used the concept of bounding boxes to retrieve objects in structured meshes. Their approaches use rectangular bounding boxes, while we use only line segments. This allows us to simplify the algorithms and make them more suitable for analysis. Additionally, we use a union-find data structure to track connected line segments, while their approaches directly produce lists of bounding boxes. During the region-growing step, these lists dynamically grow and merge. Managing these lists requires more time than managing a union-find data structure.

Chapter 4

Multi-resolution Bitmap Indexes

The unique characteristics of scientific data and queries cause traditional indexing techniques to perform poorly on scientific workloads, occupy excessive space, or both. Refinements of bitmap indexes have been proposed previously as a solution to this problem. In this chapter, we describe the difficulties we encountered in deploying bitmap indexes with scientific data and queries from two real-world domains. In particular, previously proposed methods of binning, encoding, and compressing bitmap vectors either were quite slow for processing the large-range query conditions our scientists used, or required excessive storage space. In this chapter, we show how to solve these problems through the use of multi-resolution bitmap indexes, which support a fine-grained tradeoff between storage requirements and query performance. Our experiments with large data sets from two scientific domains show that multi-resolution bitmap indexes occupy an acceptable amount of storage while improving range query performance by roughly a factor of 10, compared to a single-resolution bitmap index of reasonable size.

4.1 Drawbacks of Previous Solutions

4.1.1 Interval Encoding

Let us consider a data set with n distinct objects and an attribute A over a domain with m distinct values. We assume that the value of A for each object is chosen uniformly from the underlying domain. Under this assumption, the expected number k_v of objects that have value v for A is

$$k_v = \frac{n}{m}. \tag{4.1}$$

If L is the expected length of the ideally compressed string, then information theory tells us that

$$L \geq H, \tag{4.2}$$

where H is the entropy of the input string [CT]. Formula 4.2 says that the expected length of the ideally compressed bit vector is at best equal to the entropy of the bit vector. To derive the entropy of the bit vector, we need to find the probability distribution of the bit vector.

There are $\binom{n}{k}$ different arrangements of k 1s in a bit vector of length n. Thus if there are k objects with value i for the attribute, then the probability of any particular bit vector representing the bitmap for attribute value i, under the uniform distribution assumption, is:

$$p = \frac{1}{\binom{n}{k}}. \tag{4.3}$$

By definition, the entropy of a system is

$$H = -\sum_{\phi} p \log_2 p,$$

where ϕ is the set of all probability states. Hence the entropy of a bit vector with k ones is

$$H = -\sum_{\phi} \frac{1}{\binom{n}{k}} \log \frac{1}{\binom{n}{k}}.$$

This summation reduces to the following:

$$H = \log \binom{n}{k}. \tag{4.4}$$

We can also derive formula 4.4 using the standard definition of the entropy of a uniform distribution, where the entropy is the log of the number of states. Hence the worst case expected size, L_{bv}, of an ideally compressed bit vector is

$$L_{bv} = \log \binom{n}{k}. \tag{4.5}$$

Recall that m is the cardinality of the attribute being indexed. A sequence $\langle p_1, \ldots, p_m \rangle$ of nonnegative integers is a *partition of* n if $n = \sum_{1 \le i \le m} p_i$. Let P be the set of all partitions $\langle p_1, \ldots, p_m \rangle$. Then from formula 4.5, we have

$$L_{bm} = \frac{1}{m^n} \sum_{\langle p_1, \ldots, p_m \rangle \in P} (\binom{n}{p_1}\binom{n-p_1}{p_2} \cdots \binom{n-p_1-\cdots-p_{m-1}}{p_m} \sum_{1 \le j \le m} \log \binom{n}{p_j}). \tag{4.6}$$

In formula 4.6, we see that the maximum size of a bitmap is achieved when

$$p_1 = \cdots = p_m.$$

Let us now analyze the size difference for unencoded and interval encoded bitmap indexes. For the unencoded bitmap index, there are m bit vectors with a total of n 1s. Hence in the worst case, each bit vector has $\frac{n}{m}$ 1s. Using formula 4.5, the worst case size of an ideally compressed unencoded bitmap is

$$m \log \binom{n}{\frac{n}{m}}. \tag{4.7}$$

For an interval encoded bitmap, there are $\frac{m}{2}$ bit vectors with a total of $\frac{mn}{2}$ 1s. Hence in the worst case, each bit vector has $\frac{n}{2}$ 1s. Using formula 4.5, the worst case size of the ideally compressed interval encoded bitmap is

$$\frac{m}{2} \log \binom{n}{\frac{n}{2}}. \tag{4.8}$$

Thus the ratio of the worst-case size of an ideally compressed interval encoded bitmap index and an ideally compressed unencoded bitmap index is

$$\frac{\log \binom{n}{\frac{n}{2}}}{2 \log \binom{n}{\frac{n}{m}}}. \tag{4.9}$$

Using Stirling's approximation [CLRSa], we can approximate equation 4.9 as the following[1]:

$$\frac{m}{\log m} \tag{4.10}$$

Thus with ideal compression, the worst case size of an interval encoded bitmap index is much larger than the worst case size of an unencoded bitmap, with the size difference increasing very fast as m grows. For example, the earth scientists' Estimated Vegetation Index (EVI) attribute is defined over 1.2096×10^8 objects, with domain cardinality $m = 12000$. Thus the size ratio of the ideally compressed bitmaps for this case is $\frac{12000}{\log 12000}$, which is approximately 1278. This ratio means that an index that takes a few megabytes if unencoded will require a few gigabytes of storage under interval encoding in the worst case, even with ideal compression.

For the specific case of WAH compression, the total size of the bitmap index is bounded above by $8n + 8b$ if the data values are uniformly distributed, where n is the number of 1s in the bitmap and b is the number of bit vectors [WOS04]. In an unencoded bitmap the number of bit vectors is m, hence the total size of the bit vector is $8n + 8m$. Each bit vector in the interval encoded bitmap corresponds to half the cardinality of the attribute. Assuming a uniform distribution of the data values, each bit vector will consist of $\frac{n}{2}$ 1s on average. The total number of bit vectors in an interval encoded bitmap is $\frac{m}{2}$. Hence the total worst case size of an interval encoded bitmap index is $2mn + 4m$. Thus the ratio of the worst case size of an interval encoded bitmap to the worst case size of an unencoded bitmap index is

$$\frac{m(n+2)}{4(m+n)}. \tag{4.11}$$

[1]The proof is given in the Appendix A.

The ratio also increases very quickly with an increase in m. For the EVI data set, this ratio is $\frac{12000\times1.2\times10^8}{2(12000+1.2\times10^8)}$, which is approximately 6000. For most storage scenarios, this ratio is unacceptably large.

In conclusion, while the storage requirements of interval encoding are acceptable for small attribute domain sizes m, they are likely to be unacceptable for the large values of m common in scientific data. In addition, the time required to create such an index will be very high for large m, because there will be too many bit vectors (files) for the tails of all the files to fit into real memory. Thus there will be many cache misses when appending the index entries for new objects.

Our experiments reinforced this theoretical analysis by showing that the storage space and creation time requirements for an interval encoded WAH-compressed bitmap index are very high. We placed 231 MB of NASA's Earth Observing System's Moderate Resolution Imaging Spectroradiometer data for the EVI attribute (cardinality 12,000) on the local disk of a 2.6 GHz machine, with 1 GB RAM and a 200 GB disk with a maximum transfer rate of 100 MB/s. The resulting index was 32 GB in size—more than one hundred times the size of the original data. We also created a range encoded index, and it occupied 33 GB. These sizes are smaller than the ones predicted from the 1278 and 6000 ratios derived above, because real-world data values are not uniformly distributed and therefore compress better than in the theoretical analysis. Yet the ratio is large enough to conclude that even with a moderate-cardinality attribute, range and interval encoding produce indexes that are prohibitively large.

4.1.2 Binning

Binning reduces the index size, but introduces false positives. As mentioned earlier, researchers have shown that fetching the underlying data from disk to remove false positives is a dominant factor in workload performance [RSW05a]. The number of false positives can be minimized by matching the bin boundaries as closely as possible to the expected query range restrictions [RSW05a, RSW05b], or by reducing the bin sizes. The problem with the first approach is that when the query workload changes, the bin boundaries will no longer be optimal. Readjusting the

Figure 4.1. The effect of rocket query range width. The Y axis shows the number of iterations of the inner loop of the OR code (upper line), and the number of words read from the original and intermediate-result bit vectors (lower line). The X axis shows the width of the query range.

bin boundaries can be very expensive, as it may be necessary to scan the data to create the new bins.

The problem with decreasing the size of the bins is that eventually we reach the point where each bit vector corresponds to an individual value, at which point we have lost the advantages of binning. With either of these approaches, it is non-trivial to determine an appropriate bin size that will be resistant to changes in query patterns. While careful tuning of bin sizes and boundaries can address the tradeoffs between the number of false positives, space requirements, and query performance, ideally one will like to provide scientists with an indexing approach that is more robust to changes in the query workload.

4.1.3 Unencoded Bitmap Indexes

We ran experiments to show the effect of changes in query range restrictions on the performance of WAH compressed bitmap indexes. The experiments used the rocket science data set explained in detail in Section 4.4.2. The queries fetched the IDs of all objects whose temperature fell in the range $[500, 501]$, $[500, 505]$, $[500, 510]$, $[500, 525]$, $[500, 550]$, $[500, 600]$, $[500, 700]$, $[500, 1000]$, or $[500, 1500]$. For each of these range restrictions, Figure 4.1 shows the number of bit vector 4-byte words read to answer the query and the number of iterations of the inner loop of the Boolean OR code used to answer the query. Each iteration of the inner loop in the OR code takes two words (one

from each bit vector) and ORs them together appropriately based on their type (fill words, literal words, or one of each). The amount of work done in each iteration is almost constant, so the total run time of the entire OR operation is proportional to the number of iterations. Figure 4.1 shows that the performance of an unencoded unbinned WAH-compressed bitmap index is very good for smaller ranges, but as the range sizes grow the performance of the index tapers off quickly. Since scientists very frequently query over large ranges, we must address this performance issue while limiting the index size and avoiding costly checks for false positives.

4.2 Multi-resolution Bitmap Indexes

We introduce **multi-resolution bitmap indexes** to combine the benefits of binning (good performance for large range restrictions) and indexing of individual values (no false positives, and good performance for small ranges). We build multiple bitmap indexes on the same data set, but build them at different resolutions. In other words, at one level of resolution we store one bit vector for each individual value. At lower resolution levels, each bit vector corresponds to a bin of values. This leverages the fact that low-resolution indexes require fewer bit vectors to be brought into memory and that higher resolution indexes provide exact answers to queries. By combining both we gain efficiency, at the cost of increased storage for the new levels of indexing. Figure 4.2 shows how a 2-resolution index can be used to answer a query. When the index manager gets a query over a particular range, it looks up the bins in the lower resolution index that lie completely inside the query range. Then instead of searching the data indexed in the candidate bins to find false positives, the index manager switches to the higher resolution index to find the query answers in the remainder of the range.

The narrow bins of the highest resolution index can represent individual values, for integer data. This is not practical for floating point data, so instead we define a small bin size within which each value is to be considered equal during query processing—a standard approach for handling equality tests with floating point numbers. In the remainder of the chapter, we will assume that *the*

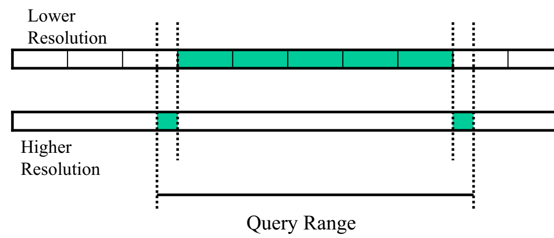


Figure 4.2. Answering a query using a 2-resolution bitmap index. The highlighted boxes show which parts of the high and low-resolution indexes are used to answer the query.

resolution of the queries themselves is no finer than the resolution of the highest resolution bins. In other words, if one value in a highest resolution bin is in the range of a query, then any other values in that bin also satisfy the query. We have not seen any applications where it is desirable to violate this assumption; if such exist, then one should augment the index lookup code with a check for false positives for the data in the two candidate bins of the highest resolution index.

Many different binning strategies could be used to form the bins. We experimented with the two most common binning strategies: **equidepth** (where the sum of the frequencies of occurrence of the values in a particular bin is the same for all bins) and **equiwidth** (where the width of the range of values in a bin is the same for all bins). We ran initial experiments with both types of bins and saw no significant advantage of using one over the other in terms of performance or storage. Since the domain is usually known in advance and the distribution of data values is not, we chose to implement equiwidth bins.

4.2.1 Reducing the Number of Bit Vectors

We ran experiments to determine how the number of bit vectors affects the overall performance of a range query. These experiments compare the performance of unencoded WAH-compressed bit vectors with a two-level multi-resolution bitmap index and with a single-level bitmap index, where both the single-level and two level indexes are WAH compressed. The queries restrict the temperature attribute of the rocket science data set discussed in detail in Section 4.4, with the same nine range restrictions as before: $[500, 501]$, $[500, 505]$, $[500, 510]$, $[500, 525]$, $[500, 550]$,

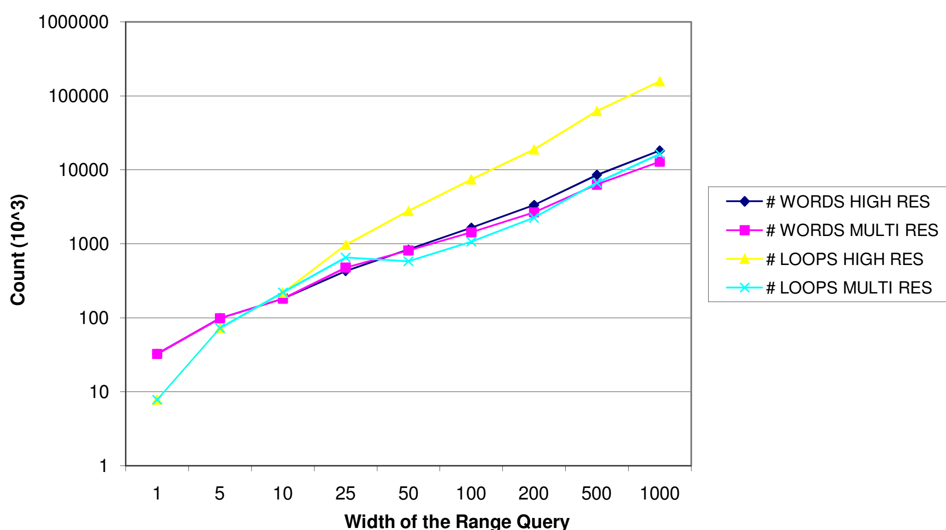


Figure 4.3. The effect of rocket query range width with single- and multi-resolution bitmap indexes. The log-scale Y axis shows the number of bit vector words read and the number of inner loops of the OR code. The X axis shows the query range width.

$[500, 600]$, $[500, 700]$, $[500, 1000]$, and $[500, 1500]$. Figure 4.3 shows the number of bit vector words read to answer each query, and the number of iterations of the inner loop of the Boolean OR code. The label HIGH RES in the figure is for unencoded single-level WAH-compressed bitmap vectors corresponding to individual values. The label MULTI RES in the figure is for a two-level multi-resolution bitmap index, with 2425 bins at the lower resolution level.

Table 4.1 shows the number of bit vectors that are operated upon (i.e., brought into memory and subjected to Boolean operations) for each of the queries in Figure 4.3. Table 4.1 and Figure 4.3 show that the number of loop iterations required is very closely related to the number of bit vectors that are operated upon. Figure 4.3 shows that when a small set of bit vectors is operated upon, the number of iterations is very similar to the number of words read. As the number of vectors increases, the number of iterations increases much faster than the number of words read from disk. This is because the intermediate bitmaps generated tend to be bigger than the ones read (higher density and hence lower compression). Thus while it is important to minimize the size of the bitmaps, that is insufficient to guarantee good performance. It is also important to minimize the number of bitmaps that must be read into memory.

Query	Number of Bit Vectors	
	HIGH RES	MULTI RES
Temp in $[500, 501]$	2	2
Temp in $[500, 505]$	6	6
Temp in $[500, 510]$	11	11
Temp in $[500, 525]$	26	8
Temp in $[500, 550]$	51	15
Temp in $[500, 600]$	101	20
Temp in $[500, 700]$	201	30
Temp in $[500, 1000]$	501	60
Temp in $[500, 1500]$	1001	110

Table 4.1. Number of bit vectors operated upon.

4.2.2 Querying a Multi-resolution Bitmap Index

Figure 4.4 presents the algorithm to respond to a range query using a multi-resolution bitmap index, for the case of a single range restriction. (The case of multiple range restrictions is a straightforward generalization, and we do not present it here.) On receiving the range restrictions, the index manager passes the attribute name, the upper and lower bounds, and the resolution level of the lowest resolution index to the QUERY method. The QUERY method determines which bins of the passed-in index fit entirely inside the query range, and ORs together the corresponding bit vectors. The QUERY method then recursively calls the next higher resolution index and ORs the returned bit vector with the bit vector it has already computed.

For the MODIS EVI attribute with 12,000 possible values, we defined the highest resolution index to have 12,000 bit vectors. For the lower resolution index, we divided the values into 500 equiwidth bins, giving 500 low-resolution bit vectors. Under WAH compression, the high and low-resolution portions of the index occupy 563 MB and 370 MB, respectively. To answer a typical query that looks for EVI values between 6,000 and 7,000, we do the following:

- Identify the low-resolution index bins that lie completely inside the range $[6000, 7000]$, and read the corresponding bit vectors. These are the bins starting with 6008 through the bin ending with 6992 (40 bit vectors).

```
GLOBAL VARIABLES:
  n_res_levels; // The number of resolution levels in the index.
  n bins[n_res_levels]; // Number of bins at each level.
  bins[n_res_levels][]; // Bins at each level. Stores the lowest value present in each bin.

QUERY(attr_name, lb, ub) {
  return QueryLevel(attr_name, lb, ub, 0)
}

QUERYLEVEL(attr_name, lb, ub, res_level) {
  BitVector answer1, answer2, result;
  if (level == 0)
    return MultipleOR(attr_name, lb, ub, res_level);
  else {
    lb_level = BinarySearch(level, lb, LOWER);
    ub_level = BinarySearch(level, ub, UPPER);
    if (bins[res_level][lb_level] > bins[res_level][ub_level])
      return query(lb, ub, level - 1);
    else {
      result = MultipleOR(attr_name, lb_level, ub_level);
      if (bins[res_level][lb_level] != lb)
        result = result | query(lb, bins[res_level][lb_level], level++);
      if (bins[res_level][ub_level]!= ub)
        result = result | query(bins[res_level][ub_level], ub, level++);
      return result;
    }
  }
}

MULTIPLEOR(attr_name, res_level, start, end) {
  bitvector result = Readvector(attr name, bins[res_level][start], res_level);
  for (i = start=1; i < end; i++)
    result = result | Readvector(attr name, bins[res_level][i], res_level);++ return result;
}

READVECTOR(attr_name, bin_value, res_level) {
  // Determines which file corresponds to the bit vector, reads it into memory, and returns it.
  return bitvector(attr_name, bin_value, res_level)
}

BINARYSEARCH(level, search value, search type) {
  Search for the search_value in the global variable bins.
  if (search_value is found)
    return search_value
  else if (search_type == LOWER)
    return value just greater than search_value
  else if (search_type == UPPER)
    return value just smaller than search_value
  }
```

Figure 4.4. Range query processing algorithm for a multi-resolution bitmap index with equiwidth bins.

- Use the high-resolution index to read the bit vectors for values between 6000 and 6008, and between 6992 and 7000 (16 bit vectors).

- OR together all the vectors read. This involves 56 bit vectors, as opposed to 1002 bit vectors if only high-resolution bit vectors were read.

4.2.3 Index Creation

To design a compact and high-performance index, we need to analyze the factors affecting performance. As shown in Section 4.2.1, we must minimize the number of bit vectors that are read to respond to a query. In Figure 4.3, the performance starts degrading if more than 100 bit vectors are read while answering the query; the threshold might be lower for much larger data sets, and higher for smaller data sets. The number of bit vectors read to respond to a range query depends on the number of resolution levels and the width of the bins at each resolution level. In choosing the number of levels and the bin width at each level, it is very helpful to know the typical query ranges. A domain expert can give rules of thumb for typical query ranges. Given these rules of thumb and the bin widths, Theorem 4.2.1 tells us how many bit vectors must be read to answer a query.

Theorem 4.2.1. *Let L be the number of resolution levels of a multi-resolution bitmap index, and let w_l be the bin width at level l. Let Q be the width of the range restriction in a single range restriction over the indexed attribute. Then the maximum number of bit vectors B that must be read to find the objects satisfying that restriction is*

$$B = \lfloor \frac{Q}{w_0} \rfloor + 2 \sum_{1 \le l \le L} \lfloor \frac{w_{l-1}}{w_l} \rfloor .$$

Proof: Resolution level l is used to process the part of the range restriction that could not be handled by the lower resolution levels. The width of the ranges passed to level l is at most twice the (candidate) bin width at level $l-1$. Thus for each resolution level l except level 0, the number of bit vectors required is

$$2\lfloor\frac{w_{l-1}}{w_l}\rfloor$$

At level 0, the number of bit vectors required to respond to the query is

$$\lfloor\frac{Q}{w_0}\rfloor.$$

Hence the total number of bit vectors required to answer a particular query is at most

$$B = \lfloor\frac{Q}{w_0}\rfloor + 2\sum_{l=1}^{L}\lfloor\frac{w_{l-1}}{w_l}\rfloor.$$

□

We can minimize the number of bit vectors required to answer an arbitrary query if the ratio w_{l-1}/w_l is equal to $|a|^{1/L}$ for all l, where $|a|$ is the cardinality of the attribute being indexed. In this case, Theorem 4.2.1 reduces to

$$B = \frac{Q}{a^{\frac{1}{L}}} + 2|a|^{\frac{1}{L}}. \quad (4.12)$$

As the number of levels L increases, the number of bitmaps B to process for a query drops. Thus increasing the number of levels can improve performance, but at the cost of increased storage—a tradeoff between storage requirements and performance. The performance of a multi-resolution index can be maximized by creating as many levels as permitted by the storage constraints.

We can further reduce the number of bit vectors read if we have additional information about the upcoming workload. Rotem et al. have shown how to optimize the placement of bin boundaries for a single-level bitmap index and a given workload [RSW05a]. They have also show that the problem is NP-hard when the optimization is for multiple attributes [RSW05b]. The proof can easily be extended to multiple levels instead of multiple attributes. The question of appropriate heuristics for placing bin boundaries in a multi-resolution index for a particular workload is a topic for Chapter 6. For now, given a typical query range width and a proposed index, we can use Theorem 4.2.1 to compute the number of bit vectors that must be read to answer the query, and adjust the bin boundaries and/or number of levels by hand.

For example, if the attribute cardinality is 125,000, then a three-level index with a bin width of 50 values at level 1 and 2500 values at level 0 will index the attribute very efficiently, even for large query ranges. Even with a query range as wide as 100,000, Theorem 4.2.1 says that the number of bit vectors read is

$$\frac{100000}{2500} + 2 \times \frac{2500}{50} + 2 \times \frac{50}{1} = 240,$$

as opposed to 100,000 bit vectors if only a high resolution index is used. But if the average range of the queries is 2000, such an arrangement will be overkill because the lowest resolution level will rarely be used. For such a scenario, a two level index with bin sizes of 50 would be good enough, as then the number of bit vectors read to answer a typical query is

$$\frac{2000}{40} + 2 \times \frac{50}{1} = 140.$$

4.3 Sizes of Multi-resolution Bitmap Indexes

Our examples and analysis show why we expect the performance of multi-resolution bitmap indexes to be better than the performance of unencoded bitmaps, and the experiments in Section 4.4 will confirm this expectation. Interval encoded bitmap indexes outperform multi-resolution indexes for range queries, but require excessive storage; Section 4.1.1 showed that an interval encoded bitmap index for the MODIS EVI attribute required almost 100 times more storage than the attribute data itself. For WAH-compressed bitmap indexes, the size of the index is at most proportional to the number of 1s in the uncompressed version of the index, which is the same at each level. Hence a multi-resolution bitmap index is at most L times larger than a single-level high-resolution index, where L is the number of levels. In practice, we have found that the actual size of every level is 15-65% of the size of the next highest level, and is on average half the size of the next highest level. Thus the expected size of the index is approximately $\sum_{0 \leq i \leq L} .5^i$ times the size of a single-level high-resolution WAH-compressed bitmap index. As already stated, three different resolutions will be enough to index most data sets, so in the worst case a multi-resolution

index will be thrice the size of a single-level high-resolution index and in practice we expect a three-level multi-resolution index to be 1.75 times the size of a single-level high-resolution index.

4.4 Experiments

To evaluate the performance of multi-resolution bitmap indexes, we conducted a series of runs using data and queries from the rocket scientists and earth scientists. The sequential experiments for the earth scientists' vegetation data employed a 2.66 GHz Intel Pentium processor with 512 KB L2 cache and 1 GB memory, with data on a local disk with "up to 100 MB/sec transfer rate" (according to the disk manufacturer, Maxtor). The sequential experiments for the rocket simulation data employed a 1.4 GHz Intel Pentium processor with 256 KB L2 cache and 512 MB of memory, with data on a local disk with "up to 100 MB/sec transfer rate" (according to the Maxtor web site).

Our bitmap index implementation follows that described in [WOSN02], with the exceptions noted below; we refer the reader to [WOSN02] for details of our implementation. The implementation in [WOSN02] assumes that the entire index and the data set fit in memory at the same time. Our implementation allows parts of a bitmap index to be written to disk while the index is being built. As in [WOSN02], we assume that the portion of the index being operated upon fits in main memory during query processing. We also store individual bit vectors in separate files, while [WOSN02] stores the entire index in a single file. The advantage of using separate files is that it is easy to add new objects to the index; the disadvantage is the extra cost of opening and closing the relevant files during lookups.

All our performance measurements are the average of five consecutive runs. The error bars in the figures show the 95% confidence interval for the mean. Where error bars are not visible, that is because the error is very small.

4.4.1 Earth Science Domain

Our experiments employed MODIS data used by earth scientists from the NASA EOS Data Gateway (http://edcimswww.cr.usgs.gov/pub/imswelcome), by selecting MODIS TERRA 1KM SIN GRID V004 16-DAY ALBEDO INDEX and VEGETATION INDICES, and 8-DAY LEAF AREA INDEX and LAND SURFACE TEMPERATURE. The data were collected for a 16 day period starting on June 26 of the years 2002-5. The resulting 9.7 GB of data has 52 attributes stored in 376 HDF files. The attributes consist of (a) the three implicit attributes time, latitude, and longitude (encoded in the array indexes and file names); (b) 25 scientific variables such as the EVI attribute used in previous examples, 20 Albedo bands, Land Surface Temperature (LST) and Leaf Area Indexes (LAI); and (c) 24 quality control attributes, all merged into four 1200×1200 arrays, with each array stored in a single HDF dataset[2]. The 20 Albedo attributes are stored as a single 3 dimensional array of size $1200 \times 1200 \times 20$. The remaining five scientific variables are stored in five separate datasets. Sequential scans over the datasets for individual attributes took 17 seconds for the EVI datasets and 213 seconds for each Albedo band attribute (due to their unique storage format, which stores multiple Albedo band attributes in each byte), with other attributes requiring times intermediate between those extremes.

The experiments are designed to compare the performance of multi-resolution WAH-compressed bitmap indexes and single-resolution WAH-compressed indexes during query processing. For the multiprocessor experiment we created a 2-resolution index. The number of bins for each level for the attributes is shown in Table 4.2. The time taken to create the indexes varied according to the time required to scan the data. For example, it took 72 seconds to create the 2-resolution index on the EVI attribute. The index sizes are shown in Figure 4.5.

The first set of experiments contains nine queries suggested by our earth scientists, which vary the size of the range restriction in the query to show its effect on the two indexing schemes. The queries all ask for the latitude, longitude, and time coordinates that have certain values for the EVI

[2] *Dataset* is a technical term in HDF, meaning a multidimensional array together with its supporting metadata, stored in a single file. We use the spelling *data set* to refer to a logically related collection of data that may be heterogeneous and distributed across multiple files and sites.

Attribute Name	Low Resolution Level		High Resolution Level	
	Bin Count	Bin Width	Bin Count	Bin Width
Albedo Attributes	40	25	1000	1
EVI	500	24	12001	1

Table 4.2. Number of bins and bin widths for vegetation attributes on which indexes were created.

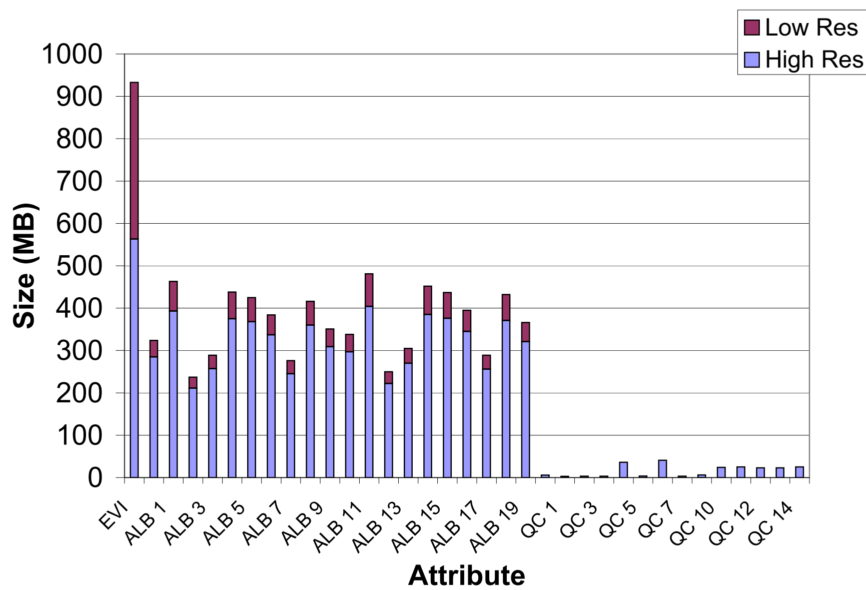


Figure 4.5. Index sizes for vegetation data.

Query Number	Query
Q1	EVI IN [4000, 4125]
Q2	EVI IN [4000, 4250]
Q3	EVI IN [4000, 4500]
Q4	EVI IN [4000, 5000]
Q5	EVI IN [4000, 6000]
Q6	EVI IN [4000, 5000]
	AND ALBEDO 2, 4, 13, 15 IN [400, 500]
Q7	EVI IN [4000, 5000]
	AND ALBEDO 2, 3, 4, 7, 13, 15, 16, 17 and 18 IN [400, 500]
Q8	EVI IN [4000, 5000]
	AND ALBEDO All Bands IN [400, 500]
Q9	EVI IN [4000, 5000]
	AND ALBEDO 1, 2, 3, 5, 7, 9, 11, and 18 IN [400, 500]
	AND QCA All = 1

Table 4.3. Queries used in earth science domain experiments.

and Vegetation Index Quality (VIQ) attributes. The queries use an equality condition for the VIQ quality control attribute, because it is categorical. Table 4.3 summarizes the set of queries we ran.

We ran Q1-Q9 under three paradigms for the high cardinality attributes: the two-resolution bitmap index, the high-resolution index only, and the low-resolution index only. We used only high resolution indexes for the categorical attributes, as the number of high-resolution bit vectors is very modest. Two resolution levels were sufficient to give good performance, so we did not experiment with three-resolution indexes.

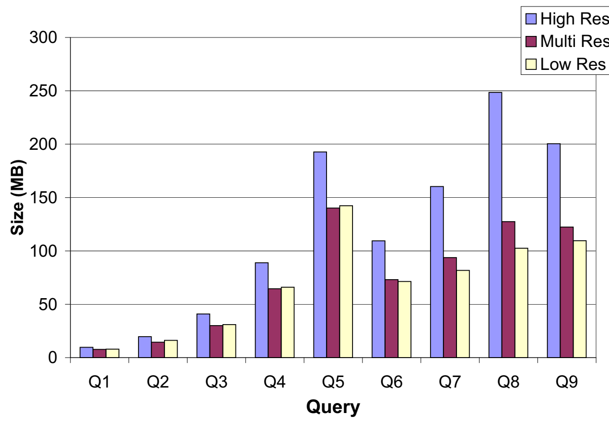
Figure 4.6(a) shows the total number of bytes read from disk to answer Q1-Q9, Figure 4.6(b) shows the number of bit vectors read, and Figure 4.6(c) shows the elapsed time for the bit operations. The figures show that the number of bit vectors read for a query is up to ten times larger when a single-resolution index is used, but the total bytes read differs by no more than a factor of 1.5. This is because some of the vectors in the multi-resolution case come from the low-resolution index. As mentioned earlier, lower-resolution bit vectors are much less compressible than higher-resolution vectors. Thus despite a drastic reduction in the number of bit vectors read, the reduction in the number of bytes read from disk is not that high. If we look at the time taken to respond to individual queries, we see that despite the disk reads being reduced by only a small factor, the

overall performance of the indexes improves by a factor of ten when we use the multi-resolution index. This indicates that the bitmap indexing is primarily CPU bound. The correlation between the elapsed times and the number of bit vectors suggests that the multi-resolution approach's reduction in the number of bit vectors is critical for good performance, even if the reduction comes with an increase in the number of bytes read.

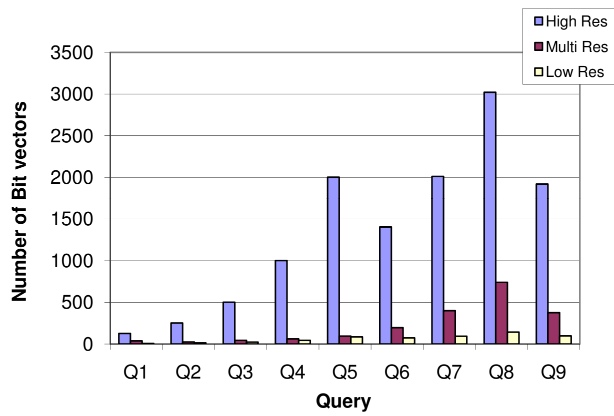
The response times for Q7 and Q9 are almost identical, despite the addition of 10 additional dimensions to Q9; this holds for both single-resolution and multi-resolution indexes. There is a marked increase in elapsed time for Q8 over Q9 for the single-resolution bitmap index; the two queries restrict the same number of dimensions, but Q8 has much wider ranges than Q9. When we use multi-resolution indexes for Q8 and Q9, the increase in time is almost negligible.

In Figure 4.6(c), runs with the multi-resolution index take almost the same amount of time as runs with the lower-resolution only index. This apparent parity vanishes once we consider the cost of checking all data in the candidate bins for false positives in the low-resolution-only index. As mentioned earlier, Rotem et al. have shown that even with optimal placement of bin boundaries for a query workload, the cost of removing false positives exceeds the cost of bitmap operations [RSW05a]. The multi-resolution approach avoids these checks, making it much faster overall than the low-resolution-only approach. The performance difference will be especially pronounced in multidimensional queries, where we have to check the candidate bins in each attribute for false positives. Because scientists like to store each attribute separately on disk, locality will be very poor for the checks.

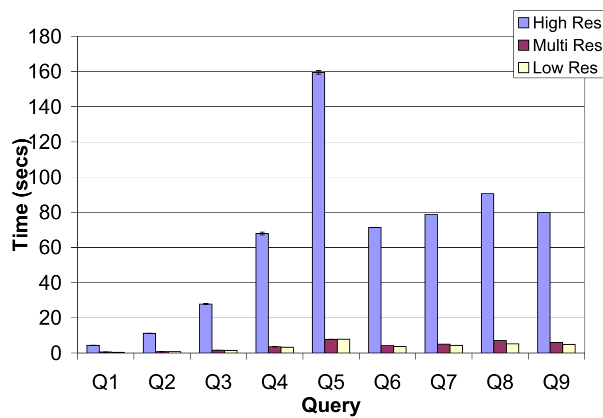
Even better performance with wide query ranges can be obtained by range or interval encoding of bit vectors. For example, the range query that restricts EVI to be in [4000-6000] is answered in 0.08 seconds on our platform using an interval-encoded WAH-compressed bitmap index, versus 7.9 seconds for the same range under our approach. Further, an interval-encoded index can answer any single-dimensional range restriction query in about that amount of time, while the time required with multi-resolution bitmap indexes can increase as the range size increases. However, we have already seen that range or interval encoded bit vectors are prohibitively large if used with



(a) Total size of bit vectors read



(b) Number of bit vectors read



(c) Elapsed time for query response

Figure 4.6. Uniprocessor performance for earth science queries Q1-Q9, excluding removal of false positives.

narrow bins; the index is 32 GB for a 220 MB EVI attribute. Though multi-resolution bitmap indexes are larger than single-resolution indexes, the increase is not prohibitive (as shown in Figure 4.5), and is offset by a factor of 10 improvement in query performance.

4.4.2 Rocket Science Domain

These experiments employed an 11.1 GB data set generated by a medium complexity rocket simulation run provided by our rocket scientists. The data set had four implicit attributes (X, Y, Z, and Time) and 11 scientific variables, with 10 of type *double* and one of type *integer*. Each variable was stored in 7,599 separate HDF data sets (each a 3D array of $12 \times 12 \times [28 - 144]$ points), with one dataset for each attribute in each of the 7,599 files. The amount of data for each attribute is approximately 220 MB, and a sequential scan over any one of the attributes takes approximately 1,020 seconds, owing to the need to open and close 7,599 different files for each attribute. The attributes of interest to us in this domain are Temperature, Pressure, X, Y, Z, Time, Vel-X, Vel-Y, and Vel-Z.

As with the vegetation data, the experiments are designed to determine whether multi-resolution WAH-compressed bitmap indexes outperform single-resolution WAH-compressed indexes during query processing. The index sizes are shown in Figure 4.7. Our early experiments suggested that query performance starts degrading if we OR more than a few hundred bit vectors. Because the rocket scientists rarely specify a temperature range greater than 2000, we chose bin boundaries so that queries would not fetch more than approximately 200 bit vectors. For the Temperature attribute, we built a two-resolution index with 2425 bins at the low resolution level and 24,250 bins at the high-resolution level. For the Pressure attribute, we built a two-resolution index with 282 bins at the low-resolution and 2820 bins at the high resolution. For the Vel-X, Vel-Y, and Vel-Z attributes, we used 253 low-resolution and 2530 high-resolution bins. Finally, for the X, Y, and Z attributes, we used 276 low-resolution and 2760 high resolution bins. The time taken to create an index depended on the time needed to scan the attribute; for example, the index on Temperature took 1200 seconds to build.

Query Number	Query
Q1′	Temp IN [5, 130]
Q2′	Temp IN [5, 255]
Q3′	Temp IN [5, 505]
Q4′	Temp IN [5, 1005]
Q5′	Temp IN [5, 2005]
Q6′	Temp IN [5, 505]
	Press IN [5, 505]
Q7′	Temp IN [5, 505]
	AND Press IN [5, 505]
	AND Vel-X IN [5, 505]
	AND Vel-Y IN [5, 505]
Q10	Temp IN [5, 505]
	AND Press IN [5, 505]
	AND Vel-X IN [5, 505]
	AND Vel-Y IN [5, 505]
	AND Vel-Z IN [5, 505]
	AND X IN [5, 505]
	AND Y IN [5, 505]
	AND Z IN [5, 505]

Table 4.4. Queries used in rocket science domain experiments.

The first set of experiments contains five different queries of a type suggested by our rocket scientists. The queries vary the size of the range restriction to show its effect on the indexing schemes, in a manner as close as possible to the queries used for the vegetation data. The rocket queries all ask for the X, Y, Z, and Time coordinates that have certain values for their Temperature attribute. Table 4.4 shows the set of queries we ran for these experiments. We ran each query under three paradigms for the high-cardinality attributes: the two-resolution bitmap index, the high-resolution index only, and the low-resolution index only.

Figure 4.8(a) shows the total number of bytes read from disk to answer each query, Figure 4.8(b) shows the number of bit vectors read, and Figure 4.8(c) shows the elapsed time for the bit operations. These figures reinforce our results from the earth science domain. As before, the number of bit vectors read for a query is up to ten times larger when a single-resolution index is used, but the total bytes read differs by no more than a factor of 1.5. If we look at the time taken to respond to individual queries, we again see that despite the disk reads being reduced by only a

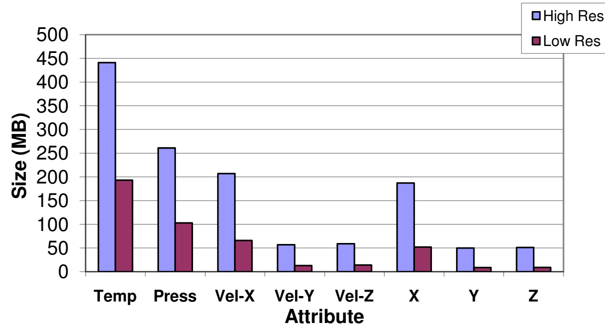
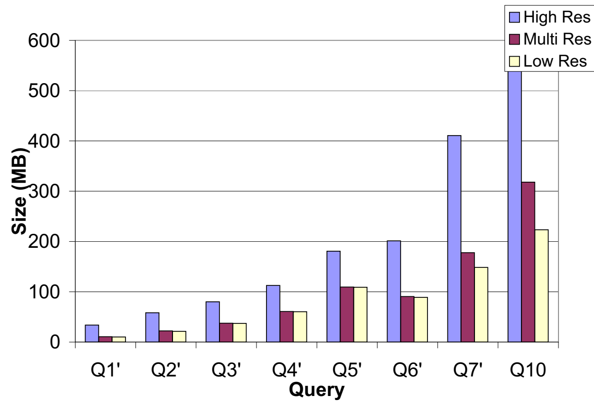


Figure 4.7. Index sizes for rocket science data.

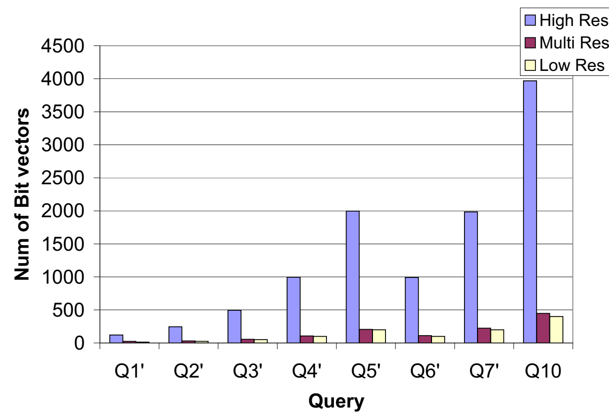
small factor, the overall performance improves by a factor of ten when we use the multi-resolution index. As before, the bitmap index processing is primarily CPU bound, and reducing the number of bit vectors that are read during query processing is crucial for good performance. As before, if we use only the lower resolution index, the performance is very close to that with a multi-resolution index. Once we consider the high cost of removing false positives, the multi-resolution approach is superior, as it produces no false positives.

Looking at the results for queries Q6′, Q7′ and Q10, we see that as we increase the number of dimensions, the query response time also increases. This is due to the width of the ranges specified for each new dimension, and the effect is much more prominent in the query over the high-resolution index than the multi-resolution index. The increase is smallest with the low-resolution index, because that index has the smallest increase in the number of bit vectors to be processed. Thus the low-resolution index outperforms the multi-resolution index in the bit operations—but the gap is less than 10 seconds, far less than the time required to check for false positives in the candidate bins of the low-resolution indexes in each dimension.

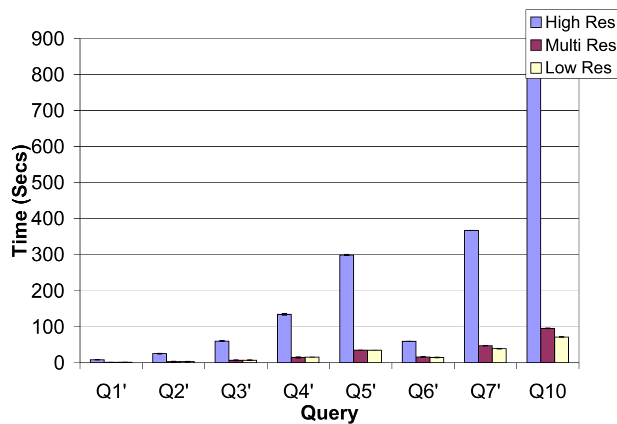
The fraction of files that must be opened and read to eliminate false positives will depend on the particular query and data layout. The rocket science data set contains 5172 files, all roughly the same size. For queries Q1 through Q5, the number of files that must be open and read to check for false positives is 403, 1718, 3596, 3622, and 3675, respectively. Queries Q6, Q7, and Q10 all include the restriction “Pressure in [5, 505]”, whose two candidate bins are completely empty;



(a) Total size of bit vectors read



(b) Number of bit vectors read



(c) Elapsed time for query response

Figure 4.8. Uniprocessor performance for rocket science queries Q1′-Q7′ and Q10, excluding removal of false positives.

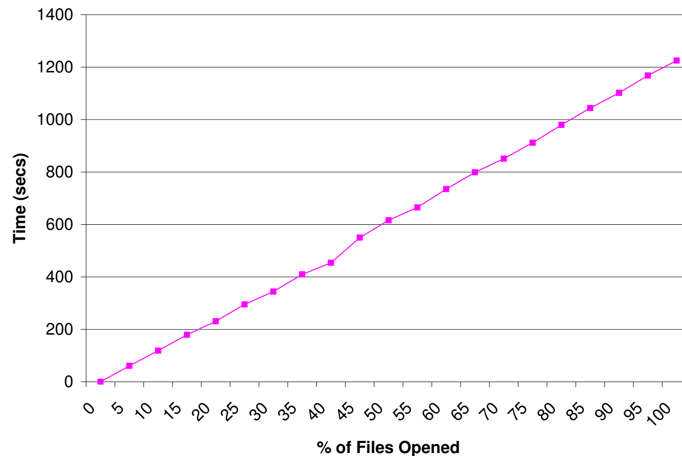


Figure 4.9. Elapsed time to read underlying data files to remove false positives.

therefore we do not have to check for false positives for these three queries.

To generalize these results for arbitrary queries, we measured the false positive removal costs (opening and reading HDF files) for a cold run with an initially empty file cache. The results are presented in Figure 4.9, which shows the time required to read different percentages of the files in the rocket science data set. Figure 4.9 shows that the removal time increases linearly as the number of opened files increases. If we have to read 5% of the files, the time required is 61 seconds, which more than doubles the time required to respond to every query in our experiments except Q10. If 20-25% of the files must be read, over 200 seconds are needed to remove the false positives. As the scale of simulations increases in the future and their output data sets grow larger, the number of files will also rise, and hence removal of false positives will become even more expensive.

The results of previous experiments have suggested that bitmap indexes are compute bound. In the next experiment, we verified this claim by timing the computation phase and I/O phase for Q6′. In this case, the time for I/O was .69 seconds while the time for computation was 10.49 seconds. We also used Intel VTune Performance Analyzer [VTu] to get further information on how the processor time was being spent for query Q6′. Table 4.5 shows the level 2 cache misses sampled, clock ticks sampled, instructions retired (IR) sampled, mispredicted branches retired (MBR) sampled and clock ticks per instruction (CPI) sampled. The hardware counters were sampled once

Module	L2 Cache Misses	Clock Ticks	IR	MBR	CPI
Kernel	4452	1365	153	26	8.922
Query	3017	10206	10765	5796	0.948

Table 4.5. Processor utilization for query $Q6'$.

every millisecond.

The cache miss counts show that a large proportion of the L2 cache misses, almost 60%, were handled by the kernel. This means that the kernel was able to anticipate those cache misses and hence allow the program to run with fewer L2 cache misses. This is also confirmed by the low instruction retired count for the kernel, as that indicates that the kernel spent a large portion of the time prefetching the cache lines from memory. The L2 cache misses in the query program are due to separate bit vectors being stored in separate arrays. If we could merge these bit vectors into one array, we could considerably improve cache performance. Merging could be accomplished either by rearranging the data once it is read in memory or by storing the bit vectors contiguously on disk. The problem with storing the bit vectors contiguously on disk, is that we lose the ability to extend the index by appending bits representing new objects to each bit vector (file).

The second most vital statistic is the large number of branch mispredictions. This is because the WAH compression/decompression algorithm requires many comparisons. In essence, with a bitmap index we need to check the first bit of each word. This introduces a large number of branches and hence causes a large number of branch mispredictions. This results in a fairly high CPI of .948. It would be interesting to explore the potential performance improvements of a more L2-cache-aware and branch-prediction-aware version of bitmap indexes.

Chapter 5

Parallelizing Bitmap Indexes

Parallel platforms enable scientists to solve larger size problems, at the cost of a painful parallelization process, lack of portability, painful tuning of application code and a relatively unfriendly runtime environment. Sequential read or write access to a bitmap index would be a bottleneck in a parallel environment, so we need to support parallel creation and search of indexes. Simulations running on parallel platforms use large numbers of processors for computation, and they need index creation and search methods that do not require a large amount of message passing, as that would slow down the entire simulation.

The type of parallel platforms we consider is driven by the needs of scientists. Most scientists use large clusters of PCs or workstations as the parallel platforms for their applications, because such clusters are inexpensive to purchase. Most of these platforms have a central file system that exports files to the cluster. These file systems can be inherently parallel (e.g., Lustre [LUS] or GPFS [GPF]), or simple networked file systems like NFS. The processors on the cluster use ordinary I/O calls to these file systems, or else call parallel I/O libraries like ROMIO [ROM] to do the required I/O. While each node of the cluster may have a local disk, these disks are only used as scratch space by scientists. Otherwise, a subsequent run on different processors might cause undesirable file data transfer activity on the interconnect, slowing down other users' jobs and robbing them of good local disk performance. When implementing a parallelization scheme for indexes in such a system, we assume that the indexes are resident on the platform's file server (not the local disks).

Up until recently post processing of the data sets produced by scientific simulations was done on a uniprocess desktop computer. With the proliferation of large clusters in the scientific comput-

ing domain, the amount of data being produced by parallel simulations has increased many fold. Post processing of data produced by these simulations itself requires multi-processor machines. For example, Tech-X Corporation, a private company providing software for plasma simulations, produces 2-3 terabytes of data in a single simulation lasting a week. Even when indexes are used, the visualization and analysis of such large amounts of data itself requires high performance clusters. Sequential indexing would become a big bottleneck in such a case. So we need an efficient method to create and read bitmap indexes in parallel on such clusters. In this chapter, we present a parallel version of multi-resolution bitmap indexes, and evaluate its performance.

5.1 Problems with Previous Solutions

While this is the first work on parallelizing bitmap indexes, others [YKM99, KF92] have proposed as parallel versions of tree based indexes. The assumption in previous work is that the data and hence the index are split across multiple disks, and the aim is to improve performance in the presence of tight concurrency control constraints. Scientific data, however, is read/append only in most cases, making the concurrency control redundant. In many clusters the only persistent store is a unified single file server, which forces us to store the entire index on a single server. The problem we are facing is the creation of indexes on multiple nodes, their merger into a single index on the external storage, and the use of that single index on multiple compute nodes of the cluster efficiently. For indexes created with $n \geq 1$ processors, we also need to support the parallel execution of an arbitrary range query on m processors, where $m \neq n$.

5.2 Bitmap Indexes for Clusters

To build a bitmap index for simulation data, a natural solution approach is to divide the work evenly across the processors by having each processor create the index entries for the objects it is going to write out as part of the simulation, and concatenate the indexes thus created into a single

index in object ID order. To look up data in parallel, in theory we can split the index evenly among the processors used in the current run with each processor receiving part of every bit vector. Then every processor can work on all of the queries that need to be processed, maximizing parallelism.

With this approach, we must ensure that all the concatenated indexes for the same data set conform to the same object ID order. This is easily done in practice. The single most common data type used to represent a scientific variable is a sequence of 2D or 3D arrays, corresponding to three or four implicit attributes (two or three spatial coordinates, one temporal) and one scientific variable. The entries in the arrays have no explicit object IDs. (For data that do have explicit object IDs, we can store an additional mapping m that maps position i in the uncompressed bitmaps to object ID $m(i)$, if query answers need to return object IDs.)

For a given variable and point in time, scientific simulation codes either use a single large array that is divided across all the processors in a run, or else use numerous smaller arrays, one or more of which are placed on each processor. Our rocket scientists use the latter approach. When a rocket simulation is ready to output a snapshot of the state of its variables, each processor can create its index entries by traversing its arrays for a variable in in-memory storage order, and outputting the appropriate bit vectors for all implicit and explicit attributes. Each variable is divided across processors in the exact same manner, so the processor can repeat the procedure for each variable, and all the bit vectors it creates will correspond to the same object ID order.

If each variable is represented at a point in time by a single large array divided across many processors, the standard practice is for the array to be incrementally reorganized into canonical order (using an implementation of the MPI-IO standard for parallel I/O, such as ROMIO) as it is output to disk. There are two ways to create a bitmap index in such a situation. The first possibility is for each processor to create the part of the index corresponding to the data in its possession, and then reorganize the resulting bitmaps as they are written out, in the same manner as the data themselves are reorganized. (MPI-IO does not currently require its implementations to support reorganization of bit vectors, but this is a straightforward extension.) The second option is to create a bitmap index incrementally as the arrays are reorganized into canonical order, by having

the parallel I/O library call the index creation utility. Because reorganization is done in parallel, the reorganized data is spread across many processors and the index creation will take place in parallel.

The challenges in making these approaches practical revolve around the handling of bitmap compression. For efficiency, we must compress the bitmaps in parallel as we create them. However, the compression process and subsequent merging will leave 'seams' where the bytes generated by one processor meet the bytes from the next. The compression paradigm will usually be violated at these seams, causing the lookup process to return incorrect object IDs. Each compression scheme will require a different approach to prevent problems at the seams; we present an approach for WAH compression, which is the best compression paradigm for bitmaps over scientific data.

WAH compression's alignment property suggests a clean but impractical solution to the seam problem. Consider the case where the word size is 32 bits. If each processor except the last gets a multiple of 31 objects to index, then concatenation of the bit vectors produced by each processor will produce a legal WAH-compressed bit vector—even though this vector may be different from that produced by sequential index creation.

Unfortunately, it is impractical in general to divide the objects into exact multiples of 31. If the data to be indexed have already been written out to files, then it would be very inefficient to read each file (to determine how many objects it has) before deciding which processor(s) would index the data in that file. If the data are not already on disk and we wish to create an index, then it is best to create the index without moving the data from the processors where they are resident, as data transfers are a hefty and undesirable expense.[1] Since we would like to have a uniform paradigm for creating indexes in parallel, we need a solution to the seam problem that does not depend on the details of how the data are divided across the processors. We do require that all variables be

[1]Data reorganization *is* worthwhile if there is significant load imbalance during index creation, due to some processors having much more data than others. This kind of load imbalance is rare, but we have encountered it in adaptive mesh refinement applications. Fortunately, if the imbalance is extreme enough to merit data reorganization for index creation, then it will also be advisable to reorganize the data to speed up I/O of the data. Thus we can create the index after the data have been reorganized, and load imbalance will not be a problem.

divided in the same manner, but scientific applications already do this for computational efficiency.

To solve the seam problem, we store the bit vectors created by different processors in separate locations (e.g., in separate directories). The index creation routine also creates a metadata file containing the ID of the first object indexed by each processor. Subsequent query processing can first apply the QUERY algorithm of Figure 4.4 unchanged, at each separate location. The result is a sequence of bit vectors (one per location) that can be concatenated carefully to form a single bit vector indicating the query answer, or carefully converted to a list of object IDs that satisfy the query. We can generate an object ID list for each location and then add the ID of the first object stored in that location to each of the returned object IDs for that location.

Another challenge for index creation on today's popular parallel platforms is the well-known problem of high contention causing severe performance degradation, due to a large number of processors trying to write a large number of files to a single storage server. To solve this problem, our index creation facility uses a "log-based I/O" library that reduces the apparent I/O cost by logging write requests, grouping write operations into batches, and interleaving I/O with computation [MSWJ05].

An index created by m processors may be queried in a subsequent n-processor run. Scientists strongly prefer that m and n be powers of two, so if $n \leq m$, we will obtain good performance by assigning each processor m/n directories from the index and converting the results to object IDs as described earlier. If $n > m$, we need to split each bit vector in a location into n/m fragments and send each fragment to a different processor. This is hard, because *all* the bits corresponding to object i in any bit vector must be sent to the same processor, so that Boolean operations on that processor will have all the bits that they need to work correctly. If we want each processor (except the last) to have roughly the same number j of objects to search, then we need to know which word in each vector corresponds to bit position j, $2j$, $3j$, and so on. With WAH compression, this requires us to read the whole bit vector before splitting it. To do this efficiently, the PQUERY algorithm in Figure 5.1 assigns n/m processors to each location, and assigns each processor a separate subset of the bit vectors in its location. (Bit vectors that are not relevant to the current

```
PQUERY(attr_name, lb, ub, n, m) {
  if (n ≤ m)
    assign m/n index locations to each processor.
    for each processor
      perform needed Boolean operations for query [lb, ub] at each location.
      compute resulting object IDs, and return them.
  else
    for each index location d
      assign a group g of n/m processors to work on d.
      assign each bit vector file in d to a processor in g.
      for each processor p in g
      for each bit vector bv assigned to p
      find bv's split positions.
      create any needed fill/literal words for the splits.
      send the pieces of bv to their destinations.
     receive pieces of bit vectors from other processors.
   perform needed Boolean operations on received bit vectors.
   compute and return the object IDs.
}
```

Figure 5.1. Processing a query $[lb, ub]$ in parallel. Here n is the number of processors performing the query and m is the number of index locations (the number of processors involved in index creation). Both are powers of 2.

query can be ignored.) Each processor determines exactly where each of its assigned bit vectors should be split, employing the heuristic that each processor gets a multiple of 31 objects (so that no bit shifting operations will be needed when sending the bit vectors to their destinations). If a file needs to split on a fill word, the processor also computes the words that will replace that fill word. The processors then use MPI scatter/gather calls to move each split-off piece of each bit vector to the processor where it belongs, without doing bit shifting or any data copying at the application level.

The above discussion concentrates on creating indexes in massively parallel simulations. Observational data is most often not collected in parallel, but such data is rarely used as is. The streaming data from observational sources goes directly to storage and is later processed to a level where scientists can actually use it, e.g., by calibrating for idiosyncrasies of the instruments that recorded the data. This processing is primarily done in parallel, and indexes can be created quickly at this point.

5.3 Experiments

As discussed earlier, due to the low cost of acquisition, the most popular parallel environment for scientists today is clusters of PCs with an attached high-performance file server. The parallel experiments employed the Turing cluster owned by the Computational Science and Engineering Program at the University of Illinois. The Turing cluster consists of 768 Apple Xserves, each with two 2 GHz G5 processors and 4 GB of RAM, for a total of 1536 processors. The primary network connecting the cluster machines is a high-bandwidth, low-latency Myrinet network from Myricom. In addition, all machines in the cluster are also connected by a 100 Mbs switched, full-duplex Ethernet using switches from Cisco Systems, and there is a 1 Gbs link between the front-end array and the primary Cisco switch. The operating system for the Turing cluster is Mac OS X Server, currently version 10.3.

5.3.1 Earth Science Domain

This set of experiments is intended to investigate the scalability of creation and querying of multi-resolution bitmap indexes in a cluster environment. We used an MPI-based parallel program on 16 processors to create an index on Albedo band 0, and wrote it to disk. The data and the index are the same as in Section 4.4.1. Then we used 2-32 processors to query for all the spatial and temporal coordinates for which the Albedo band 0 has a value in the range $[200, 500]$.

In Figure 5.2, index creation time decreases linearly as the number of processors increases, until the improvement begins to level off a bit at 16 processors. The log-based I/O methodology delays the onset and severity of the contention that we start to see at 16 processors, but does not eliminate it. If we increased the number of processors past 32, eventually the contention caused by increasing the number of writers would outweigh the reduction in the amount of work that each processor must do.

In Figure 5.2, query performance scales almost linearly as we increase the number of processors. Typically, the number q of processors querying will be smaller than the number c of

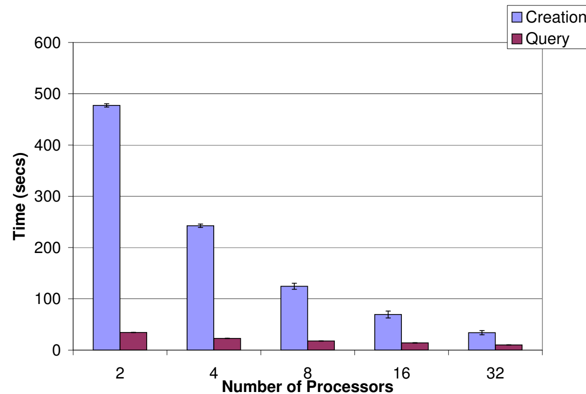


Figure 5.2. Multiprocessor performance for index creation and lookup, with earth science data.

processors that created the index, where both q and c are powers of 2. In this case, the only query-time overhead imposed by the use of an index that was created with more than q processors is that the data read by each querying processor is divided into more files than if the index had been created by q processors. The individual bit vector files should still be very large, as otherwise there is no motive to create them in parallel using so many processors. Thus the time for each processor to open and close the additional files is a very small component of the entire query run time, with a negligible effect on total query run time.

When the number of processors querying is greater than the number of processors that created the index, Figure 5.2 shows a slight hit in performance. In this case, we have to split one bit vector in a single directory into parts so that two or more processors can work on parts of that vector. This extra computation on the individual nodes causes the reduction in performance seen in the case where we are querying using 32 processors. As the number of querying processors increases, eventually the cost of sequentially scanning and splitting the vector will eliminate the advantage of adding more querying processors—e.g., if each querying processor gets one word of the vector.

5.3.2 Rocket Science Domain

As with the earth science domain, we use these experiments to investigate the scalability of the creation and querying of multi-resolution bitmap indexes in a cluster environment. We used an

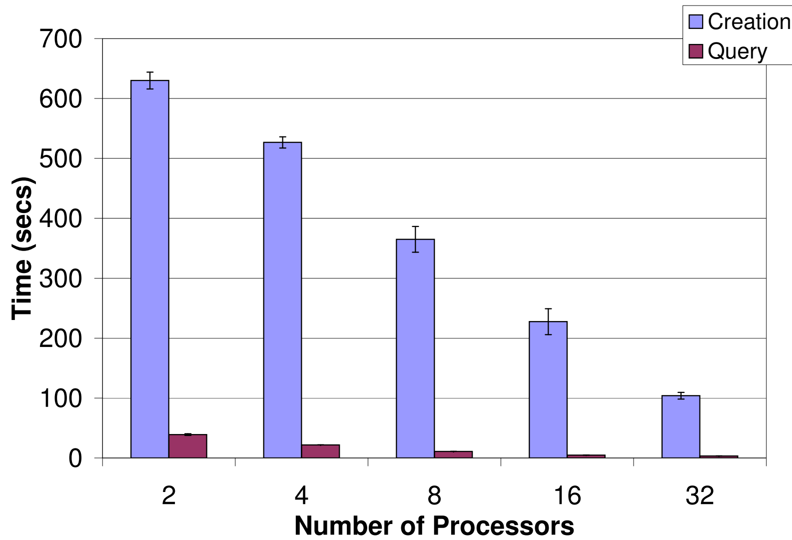


Figure 5.3. Multiprocessor performance for index creation and lookup, with rocket science data.

MPI-based parallel program on 16 processors to create an index on the Pressure attribute, and wrote it to disk. The data and the indexes are the same as in Section 4.4.2. Then we used 2-32 processors to query for all the spatial and temporal coordinates for which Pressure has a value in the range $[200, 700]$. Figure 5.3 presents the results. Compared to the earth science domain, we see an even better scale up during the index creation phase. This is because the log based I/O method is able to efficiently interleave the writes of the indexes with the large number of data read operations; there is much more data to read than in the earth science example. The queries again follow the same pattern as in the earth science domain, showing impressive speedups as we increase the number of processors.

Chapter 6

Adaptive Multi-resolution Bitmap Indexes

With data sizes measured in hundreds of terabytes, scientific data management systems often operate under tight memory **and disk** space constraints. In this chapter, we introduce adaptive bitmap indexes, which conform to both kinds of space limits while dynamically adapting to the query load and offering excellent performance. So that adaptive bitmap indexes can use optimal bin boundaries, we show how to improve the scalability of optimal binning algorithms so that they can be used with real-world workloads. As the removal of false positives is the largest component of lookup time for a small-footprint bitmap index, we propose a novel way to materialize and drop additional auxiliary indexes to eliminate the need to visit the data store to check for false positives. Our experiments with real-world data and queries show that adaptive bitmap indexes offer approximately 100-300% performance improvement (compared to standard binned bitmap indexes) at a cost of 5 MB of dedicated memory, under disk storage constraints that would cripple other indexes. We also show that an adaptive bitmap index performs **better** than an ordinary multi-resolution bitmap index twice its size.

6.1 Disk Space Constraints in Scientific Data Warehouses

As the state of the art seems sufficiently far advanced for technology transfer, we tried to convince scientists to use bitmap indexes in their projects. We quickly learned that **"big science" is not willing to accept the space overheads traditionally associated with indexing**. Bitmap indexes are small compared to most indexes, imposing roughly a 100-150% space overhead in practice for the indexed data. While this overhead would be completely acceptable in the business world, it is

not for big science:

> "The number of ESD or AOD [data files] is estimated in the range of 10 to 100 million files (a few tens of TB to a few hundred TB). Adding 30% would not be a disaster." [Rene Brune, ROOT project founder and provider of data management facilities for high-energy physics experiments at CERN]
>
> "If we can allow efficient access to the raw data at around 20-30% additional [storage] cost, then we can help science a lot." [Joseph Mohr, data management system designer for the Dark Energy Survey]
>
> "If we index the metadata only, then the size of the index does not matter. But if we index the data itself, then storage restrictions will limit index sizes to no more than a quarter of the size of the actual data. Once scientists see the value of indexes, i.e., that indexes allow them to answer new kinds of scientific questions, I expect the storage constraints to loosen." [Michael Folk, CTO of The HDF Group, data management software supplier for NASA EOSDIS and Boeing]

Scientists' reluctance to invest in more storage may seem incomprehensible until one realizes that the data rates of large-scale scientific instrumentation are much higher than in a business data warehouse. A typical business warehouse may hold a terabyte or two of data, but the high-profile scientific instruments being designed and deployed now can generate a terabyte *a day*. Storage on this scale is very costly to purchase and maintain, and there is currently no hope of getting enough additional funding to double the amount of storage so that the data can be indexed.

Existing approaches to bitmap indexes can provide excellent performance while occupying far less space than other kinds of high-performance indexes, but they cannot perform well under the stringent space limits imposed by big science. In this chapter, we introduce *adaptive bitmap indexes*, which combine the strengths of today's leading approaches to bitmap indexes while greatly reducing their size and retaining excellent performance. More specifically:

- As current algorithms for computing optimal bin boundaries [RSW05a] do not scale up to

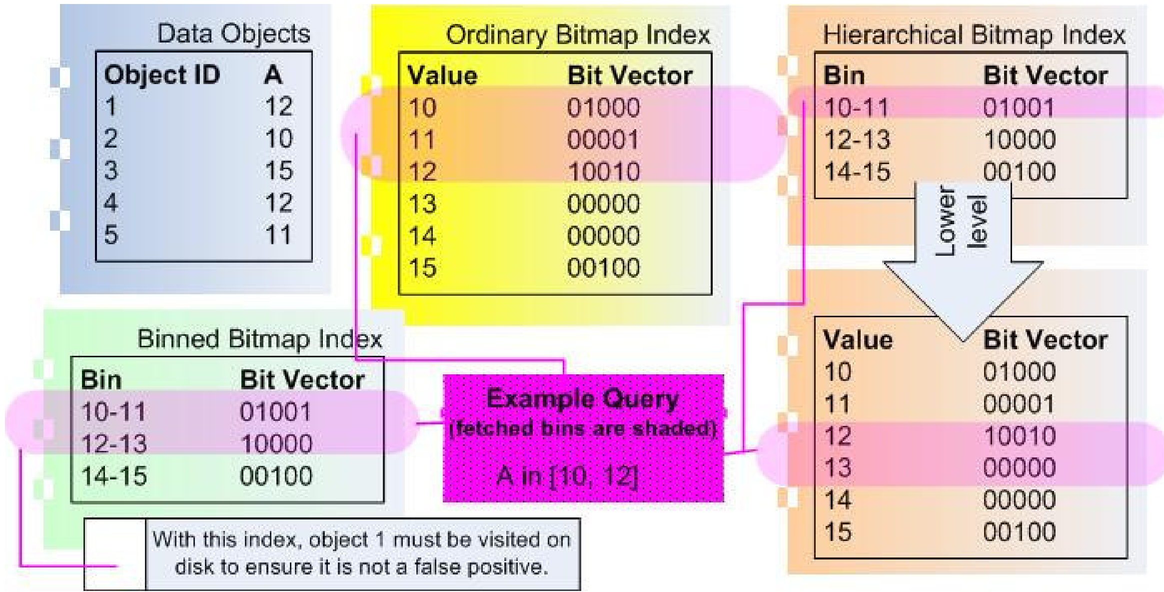


Figure 6.1. Answering queries with an unbinned (center), binned (left), or multi-resolution (right) bitmap index.

real workload sizes, we show how to compute *locally-optimal* bin boundaries for a bitmap index in a cost-effective manner.

- Even with optimal bin boundaries, false positive removal makes index lookups very expensive. To avoid expensive trips to the data store, we materialize and drop auxiliary projection indexes, without ever exceeding predefined memory and disk limits. Our model of scientists' query patterns explains why this approach is effective.

- We evaluate our approaches with a large real-world workload from the Sloan Digital Sky Survey, and with synthetic queries. We show that when constrained to occupy less than 25% of the space of the data being indexed, adaptive bitmap indexes are roughly 2 to 3 times faster than other kinds of bitmap indexes. We know of no other indexing technique that works well under such space constraints. With more generous space constraints, our methods continue to outperform other indexes. For example, our experiments show that an adaptive bitmap index is 32% faster than an ordinary multi-resolution bitmap index twice its size, for processing a real-world query log.

6.2 Locally Optimal Bin Boundaries

Rotem et al. use dynamic programming to place bin boundaries in a manner that minimizes total query processing time for a particular workload [RSW05a]. They show that the best choice of bins depends on both the data and query distributions. With respect to the latter, they observe that only the endpoints of the range restrictions used in queries (called *query endpoints*) need be considered to find optimal boundaries; this significantly reduces their search space.

The inputs to Rotem et al.'s algorithm are n, the number of bins to be created; and the training workload Q, a set of range queries with a sorted list of distinct endpoints $E(Q) = \langle e_0, \ldots, e_k \rangle$. Their algorithm chooses a *binning*, i.e., selects $n - 1$ of the members of $E(Q)$ to serve as bin boundaries. Let b_{ij} be the potential bin with endpoints $[e_i, e_j)$. Let $FP(b_{ij})$ be the set of all queries in Q that must be checked for false positives in bin b_{ij}; such queries have one endpoint that falls inside b_{ij} but is not equal to e_i. For any query in $FP(b_{ij})$, let Pages(b_{ij}) be the number of data pages that must be visited to check for false positives in that bin. If b_{ij} is chosen for a binning B, its contribution to the I/O cost for the index lookups in Q is

$$Cost(b_{ij}) = Pages(b_{ij}) \sum_{q \in FP(b_{ij})} p_q, \tag{6.1}$$

where p_q is the number of times q occurs in the query log, divided by the total number of queries in Q.

Using dynamic programming, an optimal binning for a subsequence of $E(Q)$ can be extended to an optimal binning for a longer subsequence, as shown in Theorem 6.2.1.

Theorem 6.2.1 ([RSW05a]). *Let $B(x, y)$ be an optimal binning for an arbitrary subsequence $[e_x, \ldots, e_y]$ of Q. Then*

$$Cost(B_{opt}(e_i, l)) = \min_{i < j \leq k-(l-2)} (Cost(b_{ij}) + Cost(B_{opt}(e_i, l - 1))).$$

The complexity of Rotem et al.'s algorithm is $O(nk^2)$. Their dynamic programming approach stores a $k \times k$ matrix in memory, which limits the workload size. For example, on a machine with 2 GB of memory, at most $k = 46{,}612$ query endpoints can be considered. While the storage issue can be addressed by out-of-core techniques, the computation cost grows quadratically with k, which makes this approach prohibitively expensive for realistic workloads.

We solve the scale-up problem in this chapter by first using a fast heuristic approach to choose the number of bins and assign their boundaries. The resulting structure is the lowest level of a new multi-resolution bitmap index. Then for each bin at the lowest level, we compute a set of bin boundaries *that are optimal within that bin*, to serve as bins at the next highest level. We can do this efficiently because only a small fraction of the workload will have query endpoints in that particular bin, so k is relatively small. We repeat the process with the other bins at the first level of the index, then generate additional levels in the same manner. In the remainder of this section, we describe each step of the resulting **LOMBI Algorithm**.

Step 1. Choose number of bins at level 1. If we use too few bins at level 1, then a check for false positives in a level 1 bin will require scanning almost every data page. If we use too many bins, then there will not be enough space for a second level or for auxiliary PIs.

To analyze the situation, let us assume 4-byte (float) data values uniformly distributed across 4 KB data pages. Scientists like to use column stores, so each page is filled with data values for a single attribute. Further, object IDs are typically not stored explicitly on data pages; the ith value stored for an attribute is assumed to be the value for object i. Under these assumptions, a single attribute for N objects will occupy $\frac{4N}{4096}$ pages. If we use, say, 1000 bins, then we have $\frac{N}{1000}$ objects per bin and we expect each page to contain $\frac{N}{1000} \times \frac{4096}{4N}$ objects from each bin, i.e., approximately one object from each bin. In other words, we will have to visit most of the data pages every time that we check for false positives in a bin. In this chapter, we use a default value of 1000 bins at the lowest level; a sensitivity analysis in Section 6.5.2 shows that performance is insensitive to the number of bins at level 1. If insufficient space remains for a second index level, then we increase the number of bins at level 1 until the index size approaches the maximum allowable.

Step 2. Assign bin boundaries at first level. We use equidepth bins at the lowest level, so that the optimal binning algorithm will have the same number k of query endpoints in each bin. This ensures that the $O(nk^2)$ bin boundary computation will not take too long for any individual bin. Further, the resulting bin boundaries correspond to actual query endpoints, which reduces the number of index lookups that visit the next highest level. By the time we create the second level, the values of k will be smaller and we do not need to worry about the optimal binning algorithm's performance.

Step 3. Compute the number of bins for the next level of the index. Once level l of the index has been created, the next step is to compute the number b_{l+1} of bins for the next level. The *clustering factor* for an attribute is the average number of consecutive object IDs whose values fall in the same bin. The size of a bitmap index is proportional to the clustering factor [WOS06], and the ratio of the clustering factors is always smaller than the ratio of the number of bins. In this chapter, we build 1- and 2-level bitmap indexes; in this case, the second level of the index should fill up the remaining available space. Thus we initially set the number of bins at level $l + 1$ to be $b_{l+1} = \lceil \frac{size(l)}{space} \times b_l \rceil$, where $size(l)$ is the size of the index at level l, *space* is the remaining storage space available for levels $l + 1$ and higher, and b_l is the number of bins at level l. We then build the index at the level $l + 1$ with b_{l+1} bins and measure its actual size. Typically it will not nearly fill up the remaining space, as bitmaps tend to be more compressible at higher levels. We use the actual size of the index at level $l + 1$, and the formula given above, to conservatively estimate the number b'_{l+1} of bins that will overflow the available space. We perform a binary search between the bin counts b_{l+1} and b'_{l+1}, rebuilding level $l + 1$ of the index at each visited bin count, to determine the largest number of bins that can be used at level $l + 1$ without overflowing the available space. In our experiments with SDSS data, this process required at most 5 iterations and 15 minutes of computation.

With a 25%-of-storage-space limit, and the need to have enough bins to avoid visiting every data page to check for false positives, in practice one can only build a two-level index with the SDSS data that we used. In a situation where more space was available, or the data values were

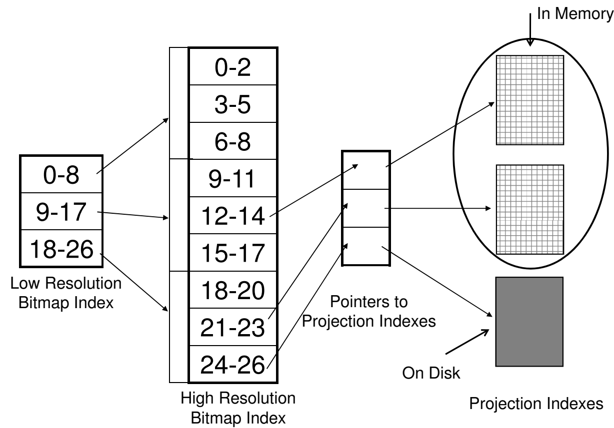


Figure 6.2. Architecture of a two-level adaptive bitmap index (ABI).

very nicely clustered, we would allot a predetermined fraction (e.g., half or two-thirds) of the available space to each level except the last, which can occupy almost all of the remaining space.

Step 4. Choose the "branching factor" for each bin. Next, for each bin at level l, we decide how many bins to split it into at level $l + 1$. Let q be the total number of query endpoints in the workload. If p of those endpoints fall into a particular bin at level l, then it splits into $b_{l+1}p/q$ bins at level $l + 1$.

Step 5. Assign locally-optimal bin boundaries. We apply the optimal binning algorithm of [RSW05a] to each separate bin at level l.

Step 6. Repeat at higher levels. We then repeat steps 3-5 until there is not enough space to create another level of the index. We refer to the result as a *locally-optimal multi-resolution bitmap index (LOMBI)*, as bin boundaries at higher levels are optimal with respect to their parent and siblings.

Figure 6.2 shows a two-level LOMBI, for the case where each attribute value is used equally often as a query endpoint. For clarity, level 1 of the LOMBI has three equidepth bins rather than 1000. Level 2 has 9 bins, because Step 3 found that 10 or more bins would overflow the space constraint. Step 4 assigned locally-optimal boundaries to each of the 9 bins, which are equiwidth in this example. Figure 6.3 shows the overall algorithm for creation of a LOMBI.

The workload Q used by the LOMBI algorithm should be large enough to capture long-term

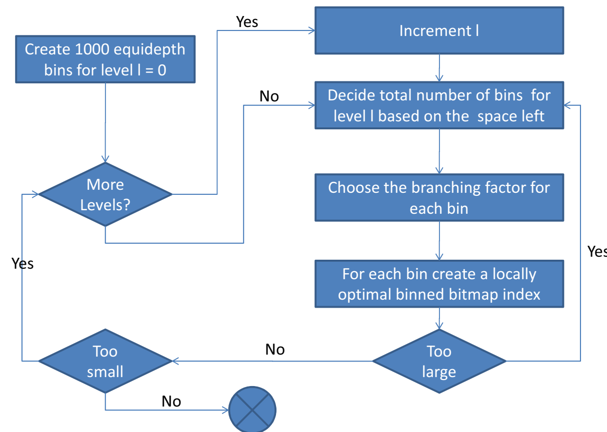


Figure 6.3. Algorithm for creation of a LOMBI

behavior. Figure 6.4 shows that over a short period of time, SDSS users usually query in only one part of the sky; but over a longer period, they examine many different areas. If Q is both small and representative, we can use optimal binning instead of a LOMBI. If Q is too small to be representative, we should use heuristics to place index bin boundaries until a longer workload is available.

6.3 Exploiting Query Locality

Even with a LOMBI, the cost of removing false positives is still extremely high, as shown by experiments presented later. Fortunately, scientific workloads exhibit spatial and/or temporal locality (with respect to the time a data value was observed or generated). We can exploit this locality to reduce workload cost.

The Sloan Digital Sky Survey (SDSS) is an astronomy survey that is collecting detailed optical images of more than a quarter of the sky, and generating an accompanying map of roughly one million galaxies and quasars. In SDSS, spatial locality takes the form of similar values for the *right ascension* (RA) and *declination* (DEC) coordinates used to describe the location of objects in the sky. Figure 6.4 shows a graph of the RA values used as query endpoints (Y axis) during 18 months of SDSS queries (X axis). Left endpoints are shown in blue, and are mostly obscured by

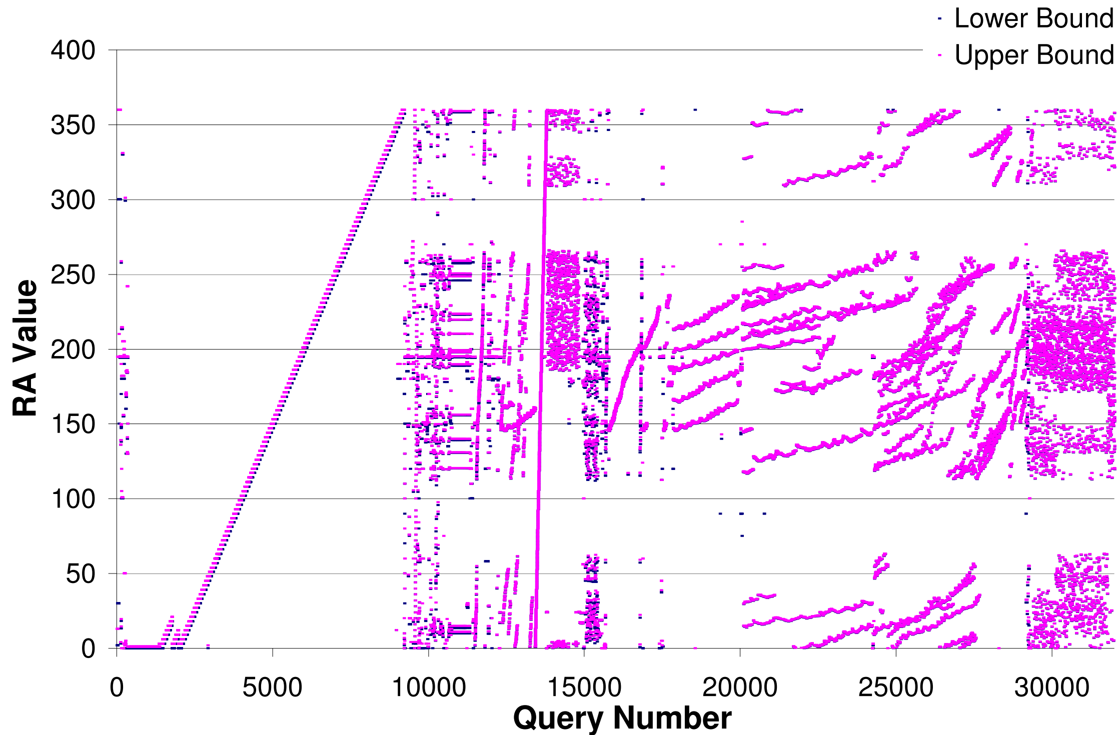


Figure 6.4. Spatial locality of SDSS queries with RA endpoints.

the magenta colored right endpoints. Except for brief periods, the graph shows that consecutive queries tend to request very similar values of RA.

We take advantage of this locality to reduce the cost of false positive removal. Whenever we go to the data to check for false positives, we create an auxiliary projection index (PI) for that particular bin, consisting of an ⟨objectID, attributeValue⟩ pair for each object in that bin. We keep each PI sorted on its attribute value column. A subsequent query whose right endpoint e falls in the same bin can do binary search for e in the PI for that bin; all objects in the PI whose attribute value are greater than e are false positives and can be eliminated without visiting the data. Scanning a PI is much quicker than parsing a data file and searching for the objects belonging to a particular bin.

As shown in Figure 6.2, an *adaptive bitmap index (ABI)* consists of a LOMBI plus a small set of PIs. As an ABI cannot ever exceed the disk space restrictions imposed by scientists, we create the LOMBI so that it is slightly smaller than the space limit. The leftover disk space is our disk cache (DC), used to hold PIs, which are relatively small. Given our storage constraints, we can

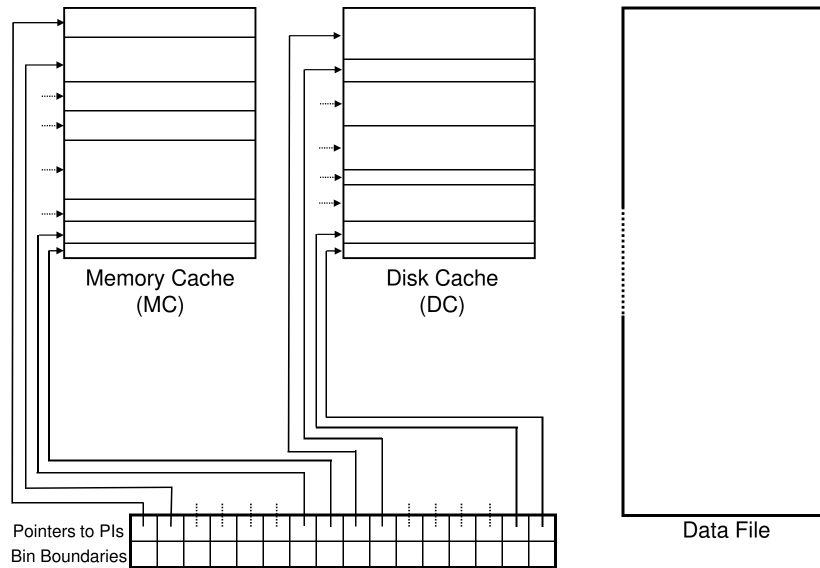


Figure 6.5. The projection index cache architecture.

materialize only a small fraction of the PIs in our memory cache (MC), and cache only a few more in DC.

Figure 6.5 shows the details of adaptive bitmap index caching. At the lowest level are the data files themselves. Whenever we visit the data file to check for false positives in a bin, we materialize a PI for that bin and put it in the MC. While querying we need to look up a particular PI in the MC to do false positive filtering. To do so we keep a data structure that maps bins to the pointers to the respective PIs, as shown at the bottom of Figure 6.5.

To make room in MC for a new PI, we must often remove one or more PIs from MC. Because scientific queries exhibit locality, we adopt a least-recently-used policy for this purpose. Since the removed PIs also have been recently used, instead of throwing them away, we store them in DC. To make room in DC for the PIs coming from MC, we drop the least-recently-used PIs in DC.

6.4 Experimental Data and Queries

6.4.1 Real-World Data and Query Log

Our experiments use real-world data and queries from SDSS. Our test data are from SDSS Data Release 1 [SDSb] and our workload is a subset of the SDSS query log from 1/1/2004 to 6/30/2006. During these 18 months, approximately 30% of the queries selected the stored image corresponding to a particular object ID; these are not interesting for indexing, and we dropped them from the log. Many of the remaining queries reference user-defined functions and stored procedures in their selection clauses, often for a nearest-neighbor search. While indexing support for these kinds of functions is an interesting topic for future work, it would be premature to consider them at this point, so we eliminated these queries from the log as well.

The remaining log consists of range queries on a variety of attributes. Previous work has examined a smaller portion of the same log, and determined that RA and DEC are the most popular attributes for range restrictions, occurring in 30% of the range queries [SWS04] and also widely used as inputs to the previously-mentioned user-defined functions. Thus we chose to use the 105,000 range queries over RA and DEC as our workload Q. To address the needs of the remaining 70% of range queries, we chose to apply the space restrictions on a per-attribute basis, so that *any* attribute occurring in a selection clause can be indexed. In other words, in our experiments, except where otherwise noted, the index for a particular attribute cannot occupy more than a given fraction of the space required to store the data values for that particular attribute. However, we treat DC and MC as a single unified LRU cache shared across all indexes. This approach benefits the attributes that are queried most often.

6.4.2 Modeling Query Patterns and Data

It is dangerous to draw general conclusions from evaluating an indexing technique only with real-world data, even for so large a data set and workload as SDSS. To tweak workload parameters and

experiment with synthetic queries, we developed a model for the query behavior of a scientist that imitates the systematic local explorations described in previous sections.

We model a scientist's query behavior as a Markov process with two states, as shown in Figure 6.6. The scientist begins by querying a random location in the sky (state 1). Then the scientist is likely to move to state 2 and look at nearby objects. Chances are high that the scientist will stay in state 2 for additional queries, but eventually she will return to state 1. For simplicity, we merge the probabilities of staying in state 2 and of moving to state 2 into a single parameter, called the *query clustering factor* (qcf), which determines how long a scientist is likely to issue queries for the same neighborhood. A second important model parameter is the size δ of that neighborhood, which we characterize as the minimum distance from the initial query location that will automatically trigger an exit from state 2 and a return to state 1. Figure 6.7 shows the locality of queries on RA for a subset of the SDSS query log. When multiple scientists submit queries concurrently, the overall degree of locality drops. The appropriate user model, then, is a set of Markov processes, one for each scientist; and the number of processes should change over time as scientists join and leave the system.

We can further tune the model by choosing the distribution of local queries inside a neighborhood, based on the size of δ. For a small δ, a uniform distribution models the SDSS query log accurately, as can be seen from Figure 6.8: it is hard to predict exactly where the next query will be. For larger δ, the query pattern can be approximated by a straight line with a certain slope, as seen in Figure 6.7.

Figure 6.7 gives a close-up view of the SDSS query log in Figure 6.4, for queries 18,000–20,000. The close-up view shows a distinct search locality pattern that is common throughout the workload, except in the portions of Figure 6.4 that show a sharp vertical line. Figure 6.8 zooms in on one such vertical region, between queries 30,000 and 32,000; here too we see some locality though it is very diminished.

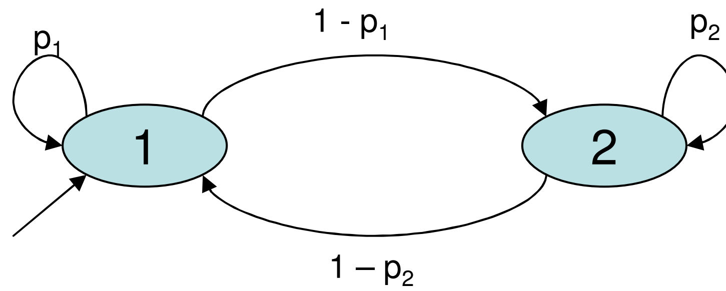


Figure 6.6. The state diagram for scientific queries.

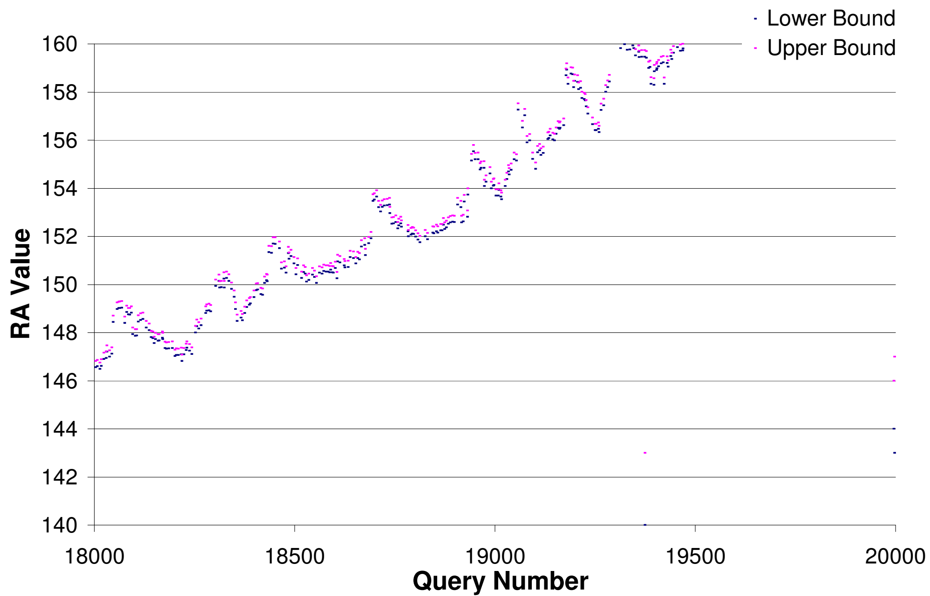


Figure 6.7. SDSS queries on RA, between query numbers 18K and 20K.

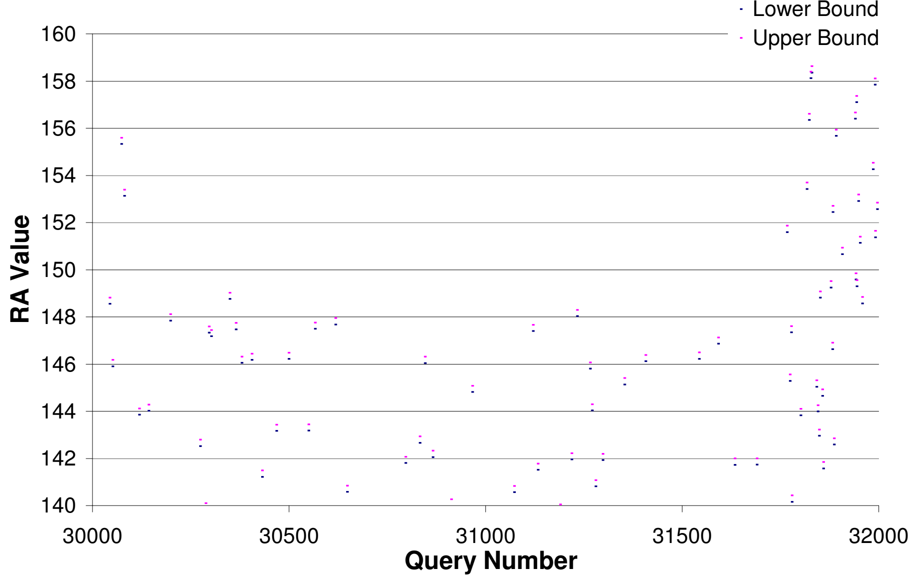


Figure 6.8. SDSS queries on RA, between query numbers 30K and 32K.

6.4.3 Generating Synthetic Data

To demonstrate the potential advantages of adaptive bitmap indexes for other data sets, we performed experiments using synthetic data generated by the Markov process described in [WOS06]. Each attribute of this synthetic data can be quantified by two parameters, the attribute cardinality C and the clustering factor f. Recall that the expected size of bitmap in a single-level WAH-compressed index is as follows [WOS06, Eq. (3)]:

$$m = \frac{N}{w-1}\left(1-(1-d)\left(1-\frac{d}{(1-d)f}\right)^{2w-3} - d\left(\frac{f-1}{f}\right)^{2w-3}\right), \tag{6.2}$$

where m is the number of computer words occupied by the index, N is the number of objects in the data set, w is the number of bits in a word (32 on our platforms), and d is the bit density, i.e., the fraction of uncompressed-bit-vector bits that are set to 1. Without binning, the bit density is $1/C$, leading to an index size of approximately Cm. For a large range of attribute cardinalities, the total size of a WAH-compressed index is nearly independent of C [WOS06, Prop. 4]:

$$Cm \sim \frac{N}{w-1}\left(1+2w-3/f\right).$$

When the index uses B equiwidth bins, the average bit density of each vector is approximately B/C and each bin represents C/B values. By the formula above, for a large range of B, the total size of the index is primarily determined by its clustering factor.

With a two-state Markov process generating the data, the clustering factor of a bitmap can be thought of as the inverse of the transition probability from a state represented by bit value 1 to a state represented by bit value 0 [WOS06]. In a C-state Markov process with clustering factor f, the transition probability from one state to any other state is given by $(C-1)^{-1}f^{-1}$. The probability of going from one of the C/B values in a bin to a value in another bin is $(C-C/B)(C-1)^{-1}f^{-1}$. If $B > 10$, then, $(C-C/B)/(C-1) \approx 1$. In these cases, the clustering factor of bitmaps with binning is nearly the same as those without binning. Therefore, the index sizes of our synthetic data are insensitive to the number of bins used in our tests. In particular, if we build a two level index, the coarse level with 1000 bins and the fine level with 100,000 bins, the total sizes of the bitmaps on the two levels are nearly the same, something not commonly seen in real world data sets. The clustering factor of the Markov process has a very strong influence on the index sizes, causing the index sizes to be nearly inversely proportional to the clustering factor.

Moreover, the index lookup time is proportional to the size of the bitmaps being operated upon. Thus we see that we have a complete analytical characterization of WAH compressed bitmap indexes when they are generated by the Markov process given in [WOS06]. This detailed characterization of the resulting bit vectors was why we chose this process to generate the synthetic data.

6.5 Experiments

The RA and DEC values are doubles, which occupy more space and provide more precision than wanted in an index. Astronomers told us that a large proportion of the scientists will be satisfied if an exact-match lookup on RA or DEC returns all objects that agree on their first 4 digits after the decimal point (the real precision in RA and DEC went to the 10th digit after the decimal point).

This high precision requires that the data be stored as doubles. Since a large proportion of the queries can be answered using a much coarser precision, we stored the RA and DEC values as 7-digit positive integers in the PIs for RA and 6-digit positive and negative integers in the PIs for DEC. The resulting data set has 644 MB of RA values and 644 MB of DEC values, each stored in a single file, with approximately 1.68×10^8 objects in each file. The data values for RA range from 0 to 3,600,000, and the data values for DEC range from -112,669 to 687,585. Since we are reducing the precision available to the scientists by using integers instead of doubles, we felt it was only fair to keep the constraints on the index sizes relative to the sizes of the integer data rather than the double data. To ensure a representative workload, we used the first 90,000 queries as input to the binning algorithm, and the remaining 15,000 queries for testing. We ran our experiments on a dual core 2.4 GHz machine with 1 GB memory and two 500 GB disks, each with a manufacturer's claimed bandwidth of 100 MB/s maximum.

6.5.1 The Effect of Memory and Disk Cache Size

In the first experiments, we limit the ABI for RA and DEC to 25% of the data size, i.e., 161 MB each. As described in Section 6.2, we created the lowest level of each LOMBI with 1000 bins and placed the bin boundaries according to the query distribution. If that level had exceeded our space limit, we would have switched to a smaller number of bins, using the binary-search methodology described in Section 6.2. If that level did not approach the space limit, as was the case for RA, we added a higher level to the LOMBI. At 146 MB, the single-level 1000-bin DEC LOMBI was close to the space limit, so we did not add another level to it. The RA LOMBI had 2 levels with 1000 bins at level 1 (27 MB) and 15,000 bins at level 2 (120 MB). The characteristics of the resulting indexes are summarized in Table 6.5.1.

Experiment 1 evaluates the effect of MC size on the run time for the 15,000 test queries, while limiting the space for indexes to 25% of the data size. Figure 6.9 summarizes the results for the RA attribute in a stacked column graph showing:

Attribute	Level 1		Level 2	
	Level Size (MB)	# of Bins	Level Size (MB)	Fanout
RA	27	1000	120	15
DEC	146	1000	N/A	N/A

Table 6.1. SDSS adaptive bitmap index sizes.

- **readFP**: time to read the data file to create the PI. Happens when a candidate bin does not have a PI. *ReadFP* is the largest component of I/O time.
- **writePI**: time to copy PI from MC to DC. Happens when MC reaches its size limit and PI needs to be moved to DC.
- **readPI**: time to read PI from MC to DC. Happens when a candidate bin has a PI in DC.
- **checkPI**: time to parse a PI that is in MC, to filter out false positives. Happens when a candidate bin has a PI in MC.
- **readBV**: time to read bit vectors from disk (entirely I/O).
- **computeBV**: time to perform OR and AND operations on bit vectors (no I/O involved).

Figure 6.9 shows the results for 6 MC sizes. The X axis has 12 entries, two for each MC size. For each MC size, the second half of the figure is for a configuration with both MC and DC, and the first half has just MC. The Y axis on the left side of the graph shows the time taken to answer the queries, in seconds. The Y axis on the right side shows the percent of queries answered without visiting the data file, i.e. answered entirely using the adaptive bitmap index. In all configurations, DC size is 15 MB, the amount of leftover disk space after creating the LOMBI for RA.

In Figure 6.9, the query times are completely dominated by the time spent visiting the data to find false positives (readFP); any increase in readFP translates to an increase in total run time. Such visits are unavoidable when the index does not include single-value bins. The spatiotemporal locality of the queries leads to much faster query responses once PIs are cached.

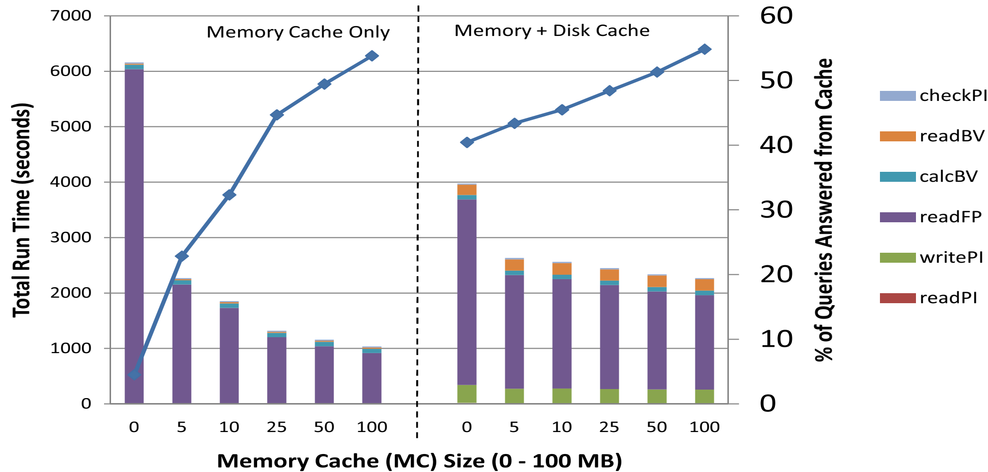


Figure 6.9. Time to run all RA queries in the 15,000 test query log, using a two level index. The curve represents the percentage of queries answered from cache.

In Figure 6.9, the workload run time improves significantly as MC increases from 0 to 5 MB, regardless of DC size. As MC grows beyond 5 MB, performance continues to improve if there is no DC, but plateaus if there is DC. This is because the second largest component in MC-without-DC run times is computeBV, while in the MC+DC version the two most dominant components are writePI and readBV, whose contention for the disk arm and space in the file system cache is slowing down both components. For the 5 MB MC with DC, the performance improvement compared to no MC is substantial, but not nearly so large as for 5 MB MC with no DC. The extra costs with DC are due to four activities contending for the disk arm (readFP, readBV, readPI, and writePI), as opposed to just readFP and readBV without DC. This contention for disk I/O also brings about a considerable increase in readBV.

To fully explain the results of Experiment 1, we need the results from Experiment 2, which measures the effect of the file system cache for a set of 100 queries on RA, randomly selected from the 15,000 test queries. We unmounted and remounted the file system before running each new query, to ensure a cold file system cache. If we read the data file to check for false positives in every candidate bin, the 100 queries took 710.96 seconds. When we stored *all* needed PIs in the MC before running any queries, and did not use the DC, the queries took 10.26 seconds. When we stored all needed PIs in the DC before running any queries, and did not use the MC,

the queries took 15.2 seconds. The stored PIs occupied 222.5 MB. On average, each query took approximately 7.1 seconds with no MC or DC, .1 second with MC and no DC, and .15 seconds with DC and no MC. Hence without caching, on average we spent 7 seconds reading the data file to find false positives! In many scientific data sets this number would be much larger, as each attribute is typically stored in many different files, and we used a single file for each attribute.

Returning now to Experiment 1: even with no DC and no MC (the leftmost column), the average time to answer each query in Figure 6.9 is approximately .4 seconds. Since Experiment 2 required 7.1 seconds per no-cache query on average, the .4 second response time reflects very positively on the file system caching algorithms, which are very effective at storing the more frequently accessed part of the data file in the file system cache in memory. The file system cache would be much less effective in a production run, however, because our experiments only involve index lookups in 644 MB of RA and DEC on a machine with 1 GB memory. A production SDSS run devotes file cache space to many other less popular attributes fetched by almost-concurrent queries. Typically, the memory will be a few orders of magnitude smaller than the recently-accessed data.

For a clearer picture of the expected file system cache behavior in a production system, consider the following analysis. The first check of a bin for false positives must read the corresponding objects from disk. If the file system cache does not already contain those objects (as would be likely in production runs), the average query execution time will be close to 7 seconds (the cold read time in Experiment 2). For queries whose PIs are already in MC, the query time will be .1 second. Thus even with 5 MB MC and no DC, we can answer 45% of the queries in .1 second and 55% of the queries in 7 seconds. This means that ABIs provide a considerable advantage for the 45% of queries answered from the MC. For further speedup, one could add active buffering [MWN+04] of bit vectors to reduce the apparent I/O times for both reads.

We repeated Experiments 1 and 2 using the DEC attribute. Experiment 2 with 100 randomly-selected queries, each with a cold file system cache, required 874 seconds with no cache, 33 seconds with MC and no DC, and 42 seconds with both MC and DC. The queries in the run with DC read 1360 MB of objects and PIs to check for false positives. This is double the size of the data

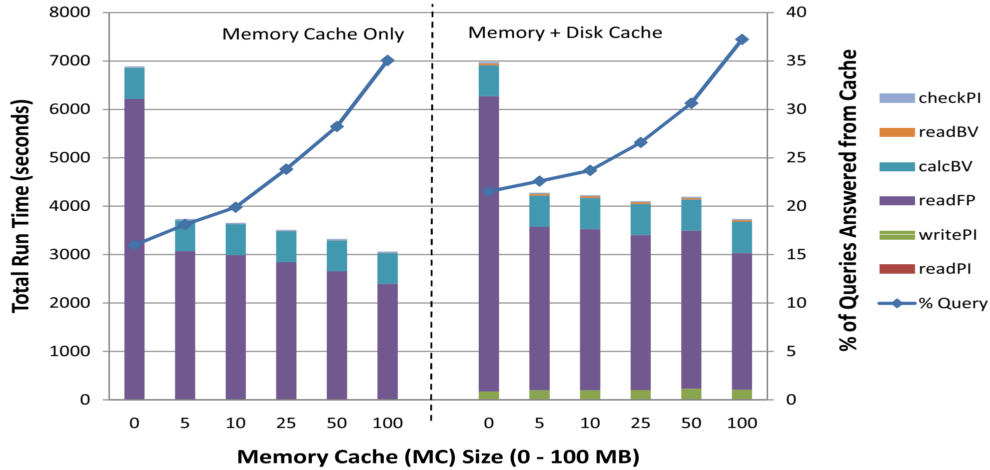


Figure 6.10. Time to run all DEC queries in the 15,000 test query log, using a one level index, with 25 percent space restriction. The curve represents the percentage of queries answered from cache.

themselves, because some PIs were read multiple times. Some PIs were much larger than others (up to 300 MB total).

The lowest (and only) resolution level of the DEC index occupies 146 MB, almost 25% of the size of the data. Thus each PI is considerably larger than for RA, so fewer PIs can be cached at one time, and fewer queries can be answered from the MC than for RA. Nonetheless, Figure 6.10 shows that up to 35% of queries are answered from MC. The overall patterns of behavior for the RA attribute also reappear in the graphs for DEC, though the total run times are much larger for DEC. More precisely, readBV is similar because the RA and DEC indexes are roughly the same size. DEC's bins are larger than RA's, leading to longer false positive lookup times (writePI, readPI, checkPI).

6.5.2 The Effect of the Number of Bins

Experiment 3 evaluates the effect on performance of varying the number of bins at level 1, while holding the space constraint constant at 25%. Overall, these experiments show that ABI performance is quite robust to changes in the bin counts as long as we *create the largest index allowed by the storage constraint*.

We experimented with three different level 1 bin counts for RA: 750, 1000, and 1250. The

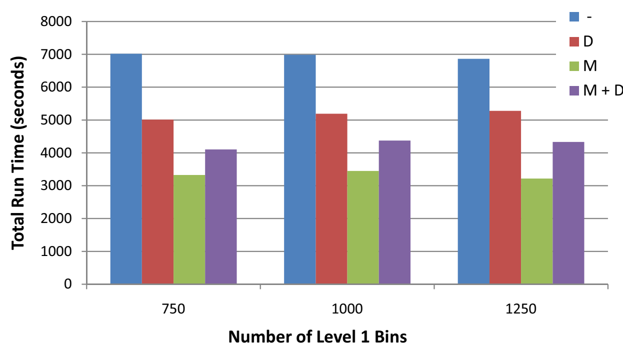


Figure 6.11. Performance effect of varying the number of level 1 bins, using the 15000 query log from SDSS for the RA attribute with a 25% space constraint.

resulting lowest level of the index occupied 21, 27, and 32 MB, respectively. We selected the number of bins at the second level so that the total size of the index in each case was as large as possible without exceeding the space constraint (25% in this experiment). The number of bins at the second level was 11,541, 11,214, and 9,972, respectively. The size of the level 2 index was 123, 120, and 113 MB, respectively. The remaining disk space that could be used for DC was 17, 14, and 17 MB respectively. The time to generate the second level locally optimal bin boundaries was 151, 192, and 201 seconds, respectively. For each bin count we ran experiments with no DC or MC, just DC, just 5 MB of MC, and both DC and MC.

Figure 6.11 shows the time taken to run the entire SDSS test query log for RA. The performance of the three binnings differs by less than 5% with no DC or MC. The slight variance in bin boundaries between the three configurations leads to small differences in false positive removal costs. When we introduce MC, even this performance difference is masked, as a 5 MB MC is effective in reducing the number of disk accesses for filtering false positives. Performance is better with DC than with no caching, showing that caching helps. When both DC and MC are used, the DC is small. In this case, the potential additional advantage afforded by the presence of DC (over and above MC) is negated by the contention for the file system cache caused by frequent reads and writes to the DC, causing a drop in performance. In particular, all three binnings perform almost identically, whether we use just the DC, or DC with 5 MB MC.

Experiment 4 measures the effect of varying the number of bins at level 2, for an index on RA

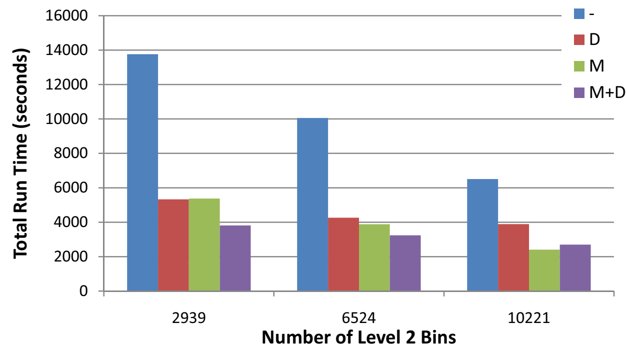


Figure 6.12. Performance effect of varying the number of level 2 bins, using the 15000 query log from SDSS for the RA attribute with a 25% space constraint.

with 1000 bins at level 1. We used indexes with 2939, 6524, or 10221 bins at level 2. The resulting LOMBIs were 70, 102, and 142 MB, allowing DCs of 96, 64, and 24 MB, respectively. For each binning, we ran experiments with no DC or MC, just DC, just 5 MB MC, and both DC and 5 MB MC, using the SDSS test query log for RA.

Figure 6.12 shows that increasing the fanout reduces the total run time, for all caching schemes. In other words, it is best to have as many bins as possible, even though less room is left for DC. For a given number of level 2 bins, performance is much better with MC or DC than without, which is expected. At the lower numbers of level 2 bins, which have many false positives, it is best to have both MC and DC; with 10221 bins at level 2, it is better to have just MC. This shows that as the size of the DC decreases, its utility also decreases, because the advantage gained by the additional PIs is lost because of added disk contention.

Comparing runs with no cache and those with just MC, we see that as the number of level 2 bins (and hence size) of the index is reduced, the relative performance gain afforded by the caching scheme increases. This means that at any index size smaller than 25%, we would always benefit, from having a caching scheme. In absolute terms, though, the fastest index is the largest index with caching enabled. Another interesting inference from Figure 6.12 is that when we create the index with the largest number of bins possible, the disk cache actually hurts performance. Thus in the remainder of our experiments we do not report results from configurations that use disk cache.

6.5.3 The Effect of the Index Size Limit

Experiment 5 evaluates the effect of varying the amount of space allowed for the ABI. We built ABIs on the RA attribute that occupied 25%, 50%, and 100% of the size of the RA data. All three ABIs had 1000 bins at level 1, occupying 27 MB. The 25% and 50% ABIs had 10,221 and 34,290 bins at level 2 (115 MB and 292 MB, respectively). The 100% ABI was identical to the 25% index at levels 1 and 2, and also included a level 3 index with 79,797 bins (483 MB). All runs used no DC and either no MC or 5 MB of MC.

Figure 6.13 shows the total run times of the six configurations. All run times are better for the 50% ABI than for the 25% ABI. With no caching the 50% ABI is 4 times faster than the 25% ABI, with only a small additional improvement if we switch to the 100% ABI. Caching improves performance by very roughly a factor of 2 for the 25% and 50% ABIs, but has almost no effect on the 100% ABI, as that index incurs almost no false positives. In fact, the best performance overall is from the 50% ABI with caching—not from the 100% ABI! This is because the increase in computation time for the 100% ABI outweighs the time spent dealing with PIs in the 50% ABI. For a fixed set of queries, as we increase the number of bins in the index, the bit vector computation time also increases and so does the total query time. The surprising conclusion is that for applications that are not disk-space-limited, we are better off building a 50% ABI than an ordinary multi-resolution bitmap index.

We also evaluated the effect of different size constraints on ABI performance for the DEC attribute. We built ABIs on the DEC attribute that occupied 25%, 50%, 100%, and 150% of the size of the DEC data. The 25% and 50% ABIs had just one level with 1000 and 3351 bins respectively. The 100% and 150% indexes had two levels, with level 1 containing 1000 bins, and level 2 containing 15771 and 78003 bins respectively. The sizes of the indexes were 153 MB, 336 MB, 665MB, and 1005 MB respectively. All runs used no DC and either no MC or 5 MB of MC.

Figure 6.14 shows the total run times of the eight configurations. When we go from a 25% sized index to 50% sized index the total time taken for responding to all the queries decreases if we don't

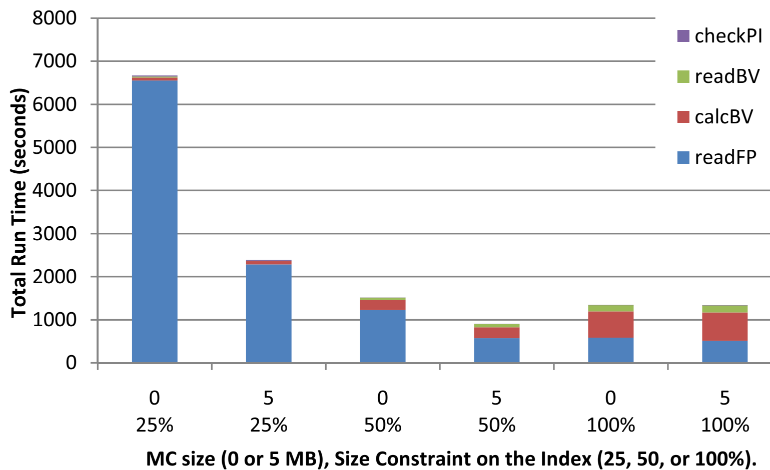


Figure 6.13. Total run time for the RA queries in the SDSS test log, under different size limits.

use the MC, but increases when we use MC. This is because bitmap computation time increases significantly (three times as many bit vectors must be ORed together) and the false positive removal cost drops only slightly. This is because the distribution of DEC attribute causes objects with the same value to be present across large number of disk pages. In the 100% sized index, we finally have a 2 level ABI, and hence the total runtime is better than both the 25% ABI and the 50% ABI. In the 150% index the calcBV (bitmap computation time) is much more than in the 100% index and is already dominating the cost of the entire query, with false positive removal only a small part of the total run time. The ideal size of the index would be where the cost of bitmap computation starts dominating the false positive removal cost. Any additional increase in space would create much larger calcBV (except if we increase the number of levels), and reduce performance. In our case such a situation would arise at around 125% index size, and hence increasing the size of the index beyond the 150% would not improve overall run time.

Figures 6.13 and 6.14 show a considerable difference between performance of ABIs on RA and DEC attributes. The main difference are

- The number of resolutions of the ABI in the same size constraint is much larger for RA than for DEC.

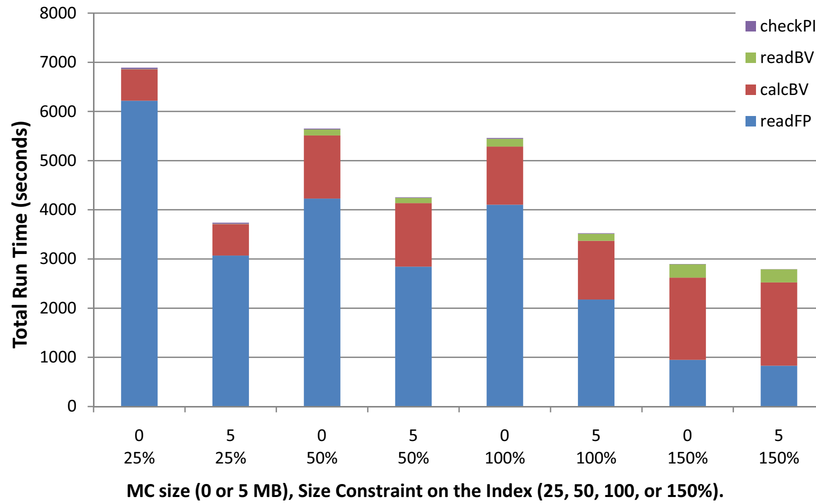


Figure 6.14. Total run time for the DEC queries in the SDSS test log, under different size limits.

- The index performance of similar configuration is better for RA than for DEC.
- False positive removal cost does not decrease a lot when we increase the size of the index DEC, while it does for RA.

The main reason for this is the difference in distribution of the RA and DEC variables. While the RA variables are very clustered with similar values being consecutive, it is not true for DEC. This causes more page lookup in case of DEC when compared to RA and also poorer index performance.

6.5.4 Performance with Synthetic Query Logs and SDSS Data Set

Experiment 6 evaluates the effect of parameters qcf and δ (Section 6.4.2) on performance, for synthetic logs of 5000 queries that model the behavior of a single scientist. As explained before, the query clustering factor (qcf) determines the number of consecutive queries in the same region, while δ determines the size of the region. We chose qcf values of 1, 5 and 10 as larger qcf did not add interesting results. We considered δ values of 0, 10 100, with query endpoints uniformly distributed within distance δ of the first query in each neighborhood. δ values of 500 and 1000 showed behavior which was very similar to $\delta = 100$, so we do not discuss them here. We ran

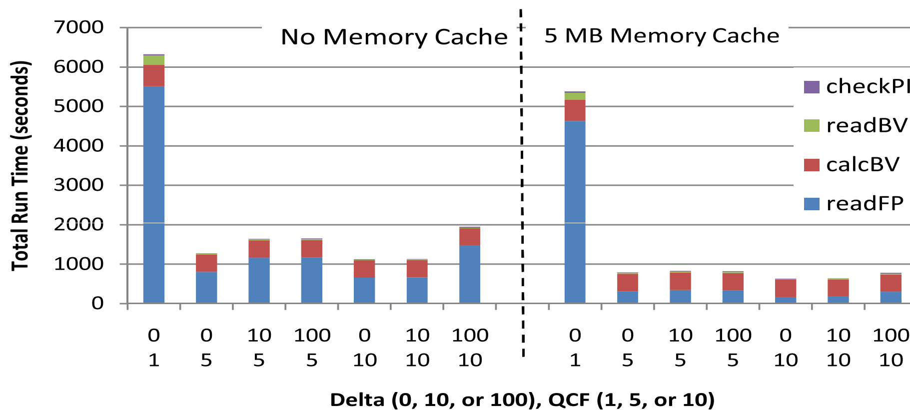


Figure 6.15. Total run time for 5,000 synthetic RA queries over real SDSS data.

experiments with no caching and with a memory cache of 5 MB.

Figure 6.15 shows the total run times for Experiment 6. When qcf is 5 or 10, MC offers a significant improvement. For $qcf = 5$, this improvement holds steady at around a factor of 2 as δ increases from 0 to 100. For $qcf = 5$ with MC, the addition of DC hurts performance; as with $qcf = 1$, DC is still too small to hold a working set of PIs, and the constant writes to DC fill the file system cache with blocks that would be better devoted to data pages. When qcf is 10, adding MC almost completely eliminates the readFP cost for $\delta = 0$, because most queries are answered from MC. When $\delta \neq 0$, MC still offers almost a factor of 2 speedup.

6.5.5 Synthetic Data and Real-World Query Log

In Experiment 7, we ran the RA portion of the test query log on a 400 MB data set generated using the Markov process mentioned in Section 6.4.3. We generated 7 different versions of the data, corresponding to clustering factors 5, 6, 7, 8, 9, 10 and 15. ("Clustering factor" refers to the clustering factor of the data without binning.) To generate the data, we first randomly select an attribute value for object 1. Given the attribute value for object i, the probability that the value for object $i + 1$ will be a different uniformly-randomly-selected value is $p = \frac{1}{f}$. With a clustering factor below 5, the size of a single level ABI with as few as 500 bins was greater than 25% of

the size of the data, so we did not consider $f < 5$. With clustering factors 5 and 6, the size of the lowest level of the index was similar to that of the DEC attribute from SDSS, implying that the clustering factor for DEC is in this general range. With clustering factor 15, the size of the lowest level of the index is similar to that of the lowest level for RA in our earlier experiments. Given the change in size of bitmap indexes for different levels, we can conclude that the RA and DEC attributes are not generated by a Markov process. Yet this being the most complete analytical model of WAH compressed indexes we have, we approximated the RA and DEC attributes using this model. Thus $f = 5, 6$ approximate the DEC attribute in our model while $f = 15$ approximates the RA attribute.

The first level of the resulting bitmap indexes occupied 94 MB ($f = 5$), 82 MB ($f = 6$), 74 MB ($f = 7$), 67 MB ($f = 8$), 62 MB ($f = 9$), and 63 MB ($f = 10$), all with 1000 bins. With these clustering factors and a 25% size constraint, there was not enough space available for a second level of the index. At $f = 15$, we created an index with 1000 bins at the lower level and 79,825 bins at the higher level, using the methodology described earlier. The levels occupied 23 MB and 30 MB, respectively.

Figure 6.16(a) shows the results of Experiment 7 for $5 \leq f \leq 10$. The X axis shows whether DC was used and whether 0 or 5 MB of MC was allocated for each run. (We did not test beyond 5 MB, as our previous runs have shown the effect of larger sizes.) Except when $f = 5$, performance using only DC (MC = 0 MB) is better when only MC is used (MC = 5 MB). For $f = 5$, very little space is available for DC, reducing its utility. When $f = 6$, DC is only 18 MB, so the performance is only a bit better than for $f = 5$; DC is much more helpful at 37 MB when $f = 10$. Overall, we should use DC for data sets and query workloads that are sufficiently well clustered for a small LRU cache to be helpful. As mentioned before we also need to take into consideration the overall space available for the DC before deciding on whether or not to use it. The presence of DC in addition to MC is always helpful in this scenario, because of the specific characteristics of the synthetic data set and can not be generalized to any arbitrary dataset. Our experiments with real world data sets show that it is always better to build the largest possible LOMBI and use just the

MC.

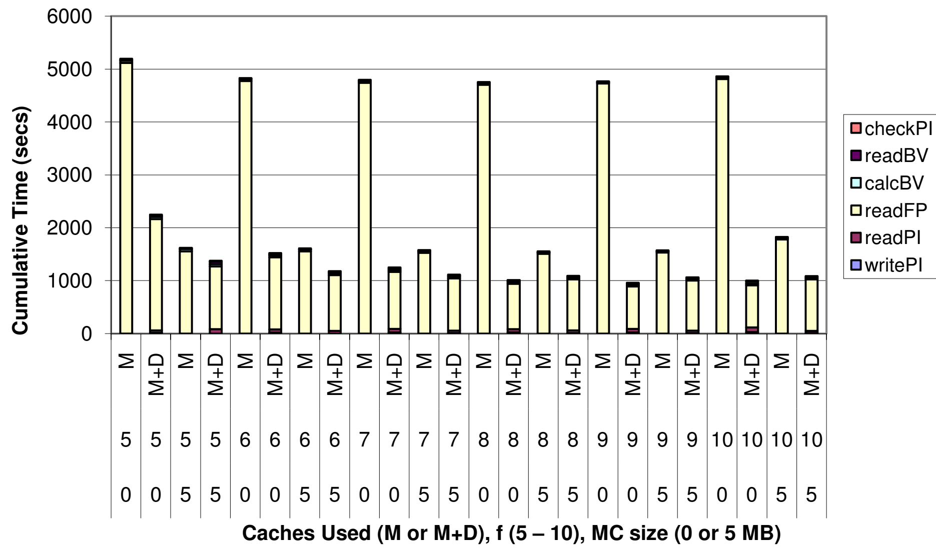
Figure 6.16(b) presents the performance for $f = 15$. With so many bins in the upper level, each bin contains relatively few values. Because the number of potential false positives is greatly reduced, the computation time begins to dominate the run time. While MC and DC still reduce readFP, the overall improvement is lessened because readFP no longer dominates the run time.

6.5.6 Multiple Files

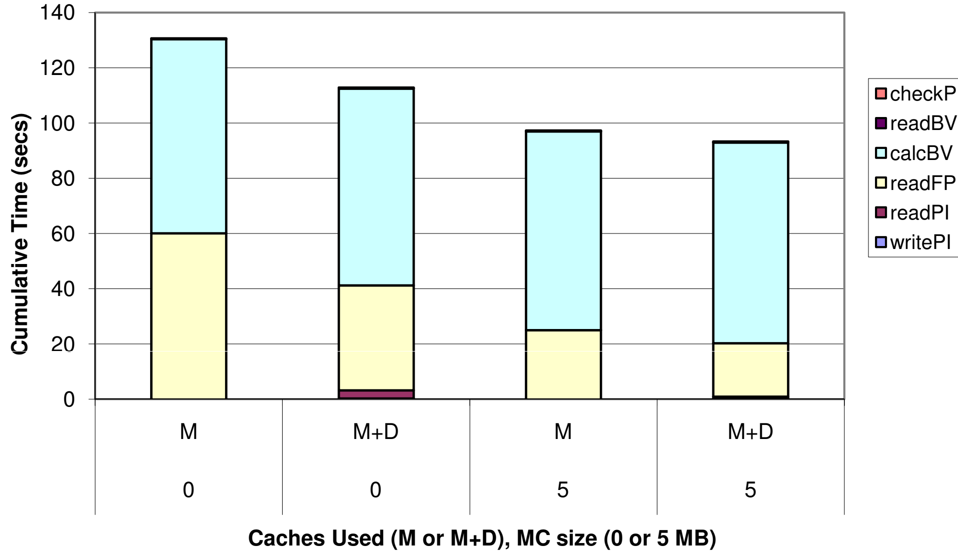
The previous experiments stored each attribute in a single separate file. As mentioned in Section 6.1, often scientists split each attribute across many separate files, which will increase the cost of visiting data pages to remove false positives. To measure this effect, in Experiment 8 we split the RA data into multiple files with 1024×1024 objects each. Further, we store each file in FITS format [FIT], not as a raw binary file. Real scientific runs on SDSS always use FITS format and this increases the cost of parsing the file to remove false positives.

We ran a microbenchmark to see how removal of false positives from multiple files compares with removal of false positives from a single file. For a file format like FITS, in general the most efficient method for removing false positives is to read the whole array into memory and check for the false positives in memory, as long as the array sizes are no more than a few megabytes. In this experiment we have generated 6 different numbers of potential false positives, 10, 50, 100, 500, 1000, 5000. These potential false positives are object IDs generated randomly for the entire data set. For each set of false positives, we measured performance for the case of a single binary file and the case where we split the entire data set into FITS files with 1024×1024 objects each. For false positive removal from a single file, we use the *lseek* library call to seek to each suspect object in the file, and allow the file system to optimize disk access. For false positive removal from multiple files we read the entire file into memory and filter false positives, as long as we must check at least one potential false positive in the file.

Table 6.17 summarizes the results of this experiment. We see that when the number of false positives is relatively low, the single file version performs 4-8 times faster than the multiple file



(a) f = 5-10



(b) f = 15

Figure 6.16. Total run time for 15000 queries from the SDSS query log, on synthetic RA data.

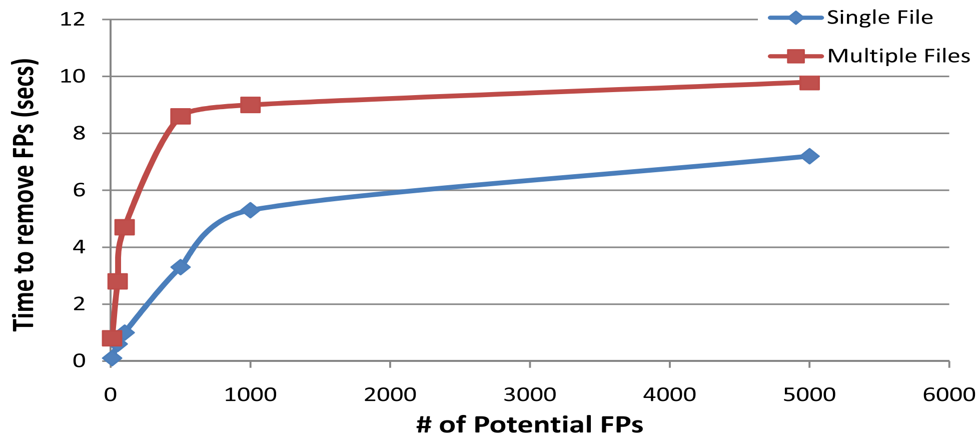


Figure 6.17. False positive removal costs with multiple files, in seconds.

version. As the number of false positives rises, the performance curve for each scheme starts to flatten out, and the relative difference between the two schemes leaches away. If there are sufficiently many false positives, almost all the disk pages in the file are read, leading to the worst case performance for single file. For the multiple file version, the worst case performance is reached as soon as we have to open almost all files, a case which is reached much earlier.

Chapter 7

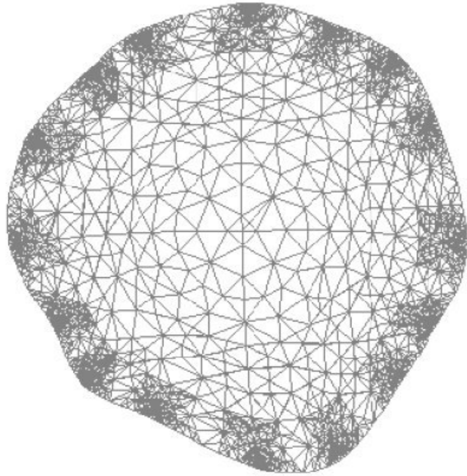
Consolidating Query Results into Regions

Most scientific data are spatio-temporal, such as sensor observational readings taken at particular locations and times, or simulations of the evolution of physical phenomena over time. Whether scientific data are produced by observations or simulations, they are discretized into points (or individual objects). In a simulation, each data point is a point in space for which variables are calculated. The smallest observation that an instrument can take is determined by the least count of that instrument. This leads to a discretization in observational data too.

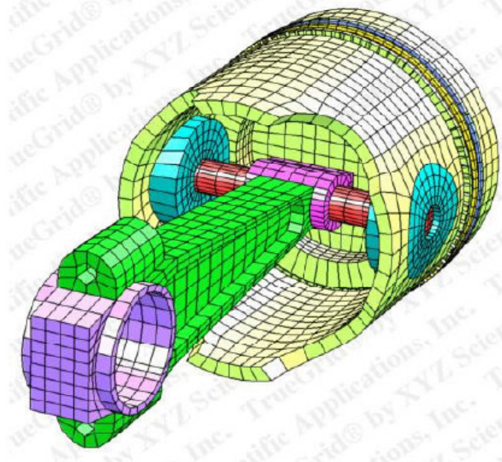
The points at which readings are taken or simulated do not exist in isolation; rather, they are connected to one another in a *mesh*. Figure 7.1 shows four example meshes. Figure 7.1(a) shows a 2 dimensional unstructured mesh representing the cross section of a diseased tree from the Scientific Computing & Applied Inverse Problems group [SCI]. Figure 7.1(b) shows a 3 dimensional structured mesh used to model a piston at XYZ Scientific Applications [TG]. Figure 7.1(c) shows a 3 dimensional semi-structured mesh used for modeling plasma flow in a magnetic field at the Princeton Plasma Physics Laboratory [PPP]. Figure 7.1(d) shows a 3 dimensional unstructured tetrahedral mesh used to model a airplane at the International Institute for Software Technology [IIS].

The connectivity of a mesh can be provided implicitly, as in images and video (where every pixel is connected to its immediate neighbors in 2- or 3-space and time) or in structured meshes (where an observation is present for every combination of X, Y, Z, and/or temporal coordinates (as in Figure 7.2)). Alternatively, connectivity can be given explicitly, as in the semi-structured and unstructured meshes in Figure 7.1.

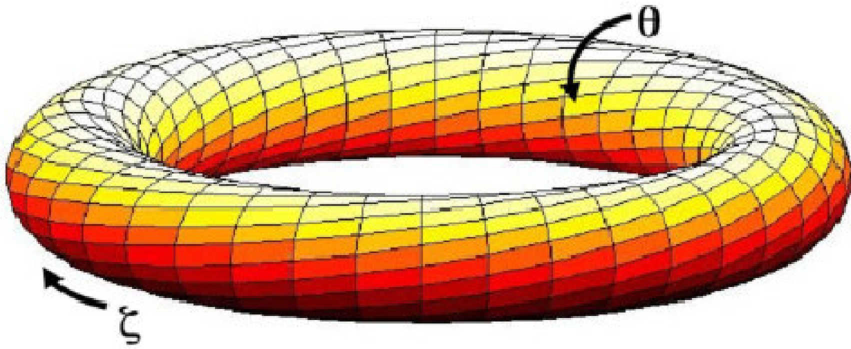
Given a user query, i.e., a set of restrictions on observed or simulated values, a *region of interest*



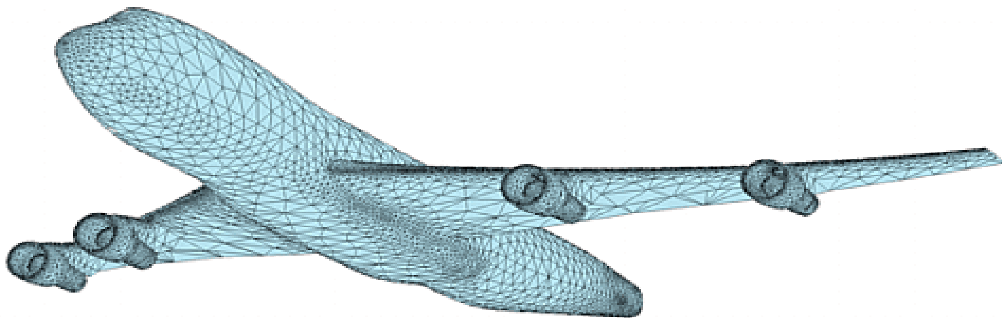
(a) 2-D unstructured triangular mesh



(b) 3-D structured mesh



(c) 3-D semi-structured toroidal mesh



(d) 3-D unstructured tetrahedral mesh

Figure 7.1. Examples of different types of meshes used in scientific/engineering computations.

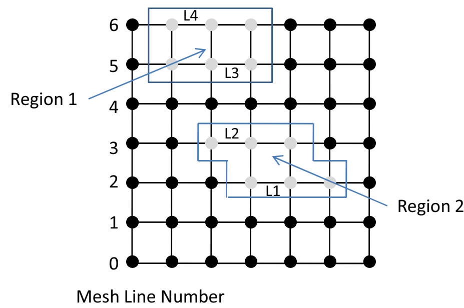


Figure 7.2. A 49-point mesh with two regions of interest.

is a *set* of connected points, each of which satisfies the user-specified constraints on its attributes and on the simulated or real time of the observation. Regions of interest are important in scientific query processing because they provide query answers to a higher level of abstractions than is possible with points alone. For example, a telescope records images of the sky in terms of pixels. An astronomer using a telescope to study various astronomical objects might start by searching for points in an image that have more than a particular brightness. These points by themselves are not of interest to scientists, but when grouped together to form stars, galaxies and quasars they become of paramount importance.

Current scientific practice is to find regions of interest in two phases. In the *constraint application phase*, we find all points that satisfy the given constraints (e.g. $temperature \in [1000, 5000]$ $and pressure > 200$). The time required for this phase can vary greatly, depending on whether indexes are available to help narrow the search and if available, how effective those indexes are for multidimensional queries. In the *region growing phase*, the selected points are coalesced into connected regions. The region growing phase is usually performed with a connected component labeling algorithm in $O(N)$ time, where N is the number of points satisfying the query. Our experiments will show that the typical connected component labeling algorithm dominates the response time.

An example will provide insight into the more efficient techniques for region growing that we propose in this thesis. Figure 7.2 shows two regions of interest in a 49 point structured mesh.

Consider this mesh as a sequence of horizontal lines called *mesh lines*, which are numbered from 0 to 6. Along a single mesh line, *no two non-consecutive points are directly connected.* Moreover, inter-line connections are also *ordered*.That is, if the kth point on line l is directly connected to the kth point on line $l+1$, then the $k+1$st point on line l is connected to the $k+1$st point on line $l+1$. In addition, the points on line l are connected to points only in lines $l-1$ and $l+1$. We use these properties of structured meshes to speed up the consolidation of regions, and also show how these properties can be generalized to speed up the consolidation process in semi-structured and unstructured meshes.

The first step in the efficient lookup of points of interest is the creation of appropriate indexes. We use bitmap indexes because they are very efficient for high-dimensional scientific queries [WOS04], and also because they can efficiently form query lines, as explained later. Since bitmap indexes consist of unidimensional bit vectors, we need to linearize the mesh into a sequence of mesh lines before we can index the attributes of its points. In the 2D example in Figure 7.2, we choose to linearize the mesh in row-major order, so that neighboring points in a horizontal line become consecutive points in the bit vector. Other linearization approaches such as column-major ordering, Z-ordering, or space-filling curves could also be used. Once a mesh is linearized, we build and compress an ordinary bitmap index for each of its attributes for use in query processing. The worst case size of the index is $2N$ bytes where N is the size number of objects in the database.

We use our bitmap indexes to quickly find the mesh points that satisfy a particular query. The output from this constraint application phase is a set of *query lines*, which are directly connected points along a single mesh line (e.g. lines L1 through L4 in Figure 7.2). In the region growing phase, we use a novel labeling algorithm to efficiently find connected components (e. g. regions 1 and 2 in Figure 7.2). To merge the lines in Figure 7.2 into regions, we start with mesh line 0 and look for a query line on that mesh line, then move on to mesh line 1 and so on. In this example, the first mesh line with a query line is mesh line 2, which contains query line L1. Then we look along mesh line 3 for query lines neighboring L1. To do this, we examine the query lines in mesh line 3 in sorted order to find the first query line that overlaps with L1, which happens to

be L2 in this case. Then we unite L1 and L2 into a single region using a union-find data structure with path compression. As we have no other query lines in mesh lines 3 and 4, we move to mesh line 5 and find query line L3. We then move to mesh line 6 and find query line L4. Since L4 is connected to query line L3, we unite the query lines L3 and L4. Regions 1 and 2 are the final result of this algorithm. In the later sections of the chapter, we show how to generalize this approach to semi-structured and unstructured meshes.

In summary, we rely on compressed bitmap indexes to find regions of interest efficiently. The major steps involved of this process are:

1. *Constraint Application Phase:* Use bitmap indexes to find all points satisfying the query constraints.

2. *Region Growing Phase:*

   (a) *Finding query lines.* Convert the resulting bit vector from the constraint application phase into a set of *query lines*. A query line is a subsequence of a *mesh line* such that all points in the query line satisfy the query constraints. A mesh line is a sequence of points such that

      i. No two non-consecutive points in a mesh line are directly connected

      ii. Two consecutive points in a mesh line are directly connected.

      iii. Each point in a mesh line is on exactly one mesh line.

   (b) *Consolidate Lines.* Use an enhanced connected-component labeling algorithm to combine query lines into regions of interest.

In the rest of this chapter we

- Present FINDREGIONS, our algorithm for efficiently finding regions of interest in large scientific data sets based on user-specified attribute value constraints. The novelty of our approach lies in its exploitation of the properties of scientific meshes to reduce query process-

ing time, together with the use of compressed bitmap indexes and connected component labeling algorithms.

- Prove that FINDREGIONS runs in time less than $O(N)$, where N is the number of data points satisfying the user-supplied query constraints.
- Evaluate FINDREGIONS on two large real-world data sets. The first is a semi-structured mesh from the Princeton Plasma Center, and the second is an unstructured mesh from the Center for Simulation of Advanced Rockets. In these two sets of data we show over a 10 fold improvement in performance compared to traditional methods.

7.1 WAH Compressed Bitmaps

Scientific data are usually stored as a set of arrays with one array for each scientific variable. Each point in the array corresponds either to a real world observation point or a simulation point. The same index in different variable arrays corresponds to the same real-world or simulated point. Bit vectors are linear data structures and hence require a linear ordering of the objects they index. Appropriate orderings for common types of meshes and images are listed in Table 7.1. Henceforth, we refer to the linearized mesh as *mesh vector*; we use mesh vectors as the object ordering for the bit vectors in the bitmap index. Figure 7.3 shows one possible linearization of the mesh shown in Figure 7.2. The figure also shows the corresponding uncompressed and WAH compressed bit vectors (compression is done assuming 8 bit words for simplicity).

In this section, we discuss the details of WAH compression, which plays an important role in the sublinear retrieval of regions of interest. Our proofs are for single variable queries. No such proof can given for multi-variable queries; although our experiments show that multi-variable bitmap operations are also sublinear.

We use WAH compression, which previous work has shown to outperform other compression schemes for bitmap indexes and to be more amenable to analytical analysis. WAH compression is

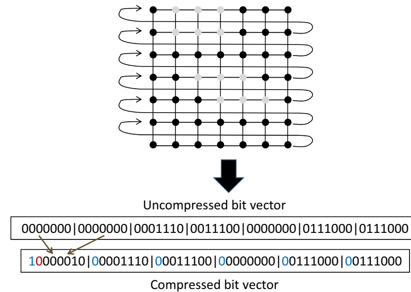


Figure 7.3. A possible linearization of the example mesh. The corresponding uncompressed and the compressed bit vectors shown below the mesh.

a combination of run-length encoding and uncompressed bitmaps. A WAH-compressed bitmap contains two kinds of words: *literal* and *fill* words. The word size should be chosen to give good in-memory performance on today's platforms, e.g., 32 bits. In this case, the uncompressed bitmap is divided into 31-bit segments. The most significant bit of each WAH-compressed word is used to differentiate between literal and fill words. A literal word contains one segment of the uncompressed bit vector verbatim, while a fill word represents a sequence of contiguous segments in the uncompressed bitmap comprise entirely of 0s or 1s. The body of a fill word tells us the number of contiguous segments that are all 0s or all 1s, as well as their value (0 or 1).

We now show analytically that bitmap OR and NOT operations have expected running time that are sub-linear in number of query result points, except in extreme cases. We consider only single attribute queries in this proof, so we do not look at the AND operation; however the experimental results presented in section 7.5.2 will show that in practice, multi-variable region retrieval is also sublinear in the number of points retrieved.

Let us call a fill word followed by a group of literal bits a *run*, and call the literal bits the *tail*. A run takes at most two words to represent: one fill word and one literal word assuming that the dataset has less than 31×2^{32} objects in case of a 32 bit word; otherwise we can split the data set into multiple parts each having less than 31×2^{32} objects. The only run that might not have a tail is the last run of a bitmap. Typically, the last few bits occupy only part of a literal word, though a whole word has to be used to store them. All together, the number of words in a WAH compressed bitmap is at most twice the number of runs. All runs except the last must contain at least one 1,

Images	Raster ordering
Structured Meshes	Row-major or column-major; z-order; other space-filling curves.
Semi-Structured	Usually the data are stored in a canonical order and we use that canonical order as the ordering of our points.
Unstructured	The points in an unstructured mesh are assigned unique identifiers. We linearize the mesh based on these IDs.

Table 7.1. Possible linearizations of different kinds of meshes.

else the run would not have a literal word in it. Hence each WAH-compressed bit vector has at most $n + 1$ runs, where n is the number of bits that are 1 in the bit vector, and has at most $2n + 2$ words. Hence a set of B bit vectors over N objects has a maximum size of $2N + 2B$. To respond to a range query over a range of B bit vectors with V points in the result, a maximum of of $2V + 2B$ bit vectors needs to be read in memory. This analysis was first shown in [WOS04].

Once the bit vectors are in memory, we need to perform a bitwise OR of the bit vectors. In our implementation, we follow the fourth approach described in Section 5 of [WOS04] for ORing bit vectors. This approach stores the intermediate results in an uncompressed bi vector. The approach to respond to range queries is linear in the total size of the bitmap index involved [WOS04]. The resulting bit vector is also a WAH compressed bitmap. In general $V \gg B$, hence the run time complexity of a range query using only OR's is $O(V)$ in the worst case.

For the negation operation, we need to change a 0 fill word to a 1 fill, and vice versa. For a literal word, we change each 0 bit to a 1 bit, and vice versa. The run time complexity of this negation operation is $O(V)$ at worst. When the range R of the query is less than $\frac{|C|}{2}$, where $|C|$ is the cardinality of the domain, we apply the OR operators in the specified range. Otherwise, we apply the OR operator over the bit vectors in the complement of the specified range and then negate the result bit vector. This reduces the overall number of bit vectors that need to be ORed to produce the result.

Let us analyze the data that leads to the worst case behavior. An occurrence of exactly one 1 in each run means that for every $31x + 30$ (where $x \geq 1$) zeros there is exactly one 1. This means that for each 1 there should be at least 61 zeros. Thus the cardinality of the attribute has to be at least $\frac{N}{61}$. Moreover, if we divide the mesh vector into segments of 31 points, then each value occurs at most once in two consecutive segments. Many attributes in scientific data, such as temperature and pressure, tend to be continuous in space. This means that points with similar values for the attribute tend to be clustered together in space. This continuity leads to non-trivial sized interesting regions, and is reflected as larger numbers of 1s in each run.

7.2 From Bitmaps to Query Lines

Mesh lines are properties of a mesh and thus are static across multiple queries. A query line, on the other hand depends on the query ,and hence changes from query to query. By definition, a query line is a sequence of query result points within a mesh line. Thus a query line has the following two properties:

- A query line consists of points that satisfy the query constraints, consecutive in the mesh vector, and hence correspond to consecutive 1s in the result bit vector.
- No query line spans multiple mesh lines.

Given mesh lines and the query result as a WAH compressed bit vector we can create query lines in a two step process in $O(S)$ time, where S is the number of query lines. The two steps are:

1. Convert the compressed bit vector into a sorted sequence of line segments, where each line segment consists of consecutive 1s in the bit vector. The corresponding algorithm, CREATELINESEGMENTS, is shown in Figure 7.4.
2. Split the line segments into query lines based on the mesh lines. The corresponding algorithm, GETQUERYLINES, is shown in Figure 7.5.

```
CREATELINESEGMENTS(BITVECTOR BV)
  LineSegs[]
  i ← 0
  pos ← findNextOne(BV, 0)
  LineSegs[0].start ← pos
  if (current 1 is a start of a fill word)
      LineSegs[0].end ← pos + fillSize
  else
      LineSegs[0].end ← pos + 1
  while (BV has more bits)
      pos = findNextOne(BV, LineSegs[i])
      if (LineSegs[i].end = pos)
          if (current 1 is a start of a fill word)
              LineSegs[i].end ← pos + fillSize
          else
              LineSegs[i].end ← pos + 1
          i ← i + 1
      else
          i ← i + 1
      if (current 1 is a start of a fill word)
          LineSegs[i].start ← pos
          LineSegs[i].end ← pos + fillSize
      else
          LineSegs[i].start ← pos
          LineSegs[i-1].end ← LineSegs[i].end + 1
  return LineSegs
```

Figure 7.4. Converting a bit vector into line segments.

For clarity, CREATELINESEGMENT uses the supporting method FINDNEXTONE(BV, position). This method returns the position of the next 1 in the bit vector after the position passed to this method. Each line segment is represented in $LineSegs$ by its start and end point. The start point of the first line segment is the position of the first 1 that is found in the bit vector. The algorithm sets the end of the line segment depending on whether the current word is part of a fill word or a literal word. The algorithm then iteratively finds the position of the next 1, until there are no more 1s. For each position, the algorithm checks if it is the same as the end of the end of the previous line segment. If so, it extends the previous line segment by the required number of points (decided based on whether the 1 is part of a fill or a literal word). This ensures that line segments that span multiple words are correctly recorded.

	Sequence	Example
1.	0 fill - 1 fill - 0 fill	10000011 - 11000101 - 10000101
2.	literal - 1 fill - literal	00111111 - 11000011 - 01111110
3.	0 fill - literal - literal - 0 fill	10000101 - 00111111 - 01111111 - 10000011
4.	0 fill - literal - 0 fill	10000101 - 00111100 - 10000011

Table 7.2. Sequences of words that can form a single query line.

Theorem 7.2.1. *Let S be the number of query lines in a WAH compressed bit vector. The time taken by* CREATELINESEGMENTS *algorithm is $O(S)$.*

Proof: To prove the above theorem, we prove that a single line segment can be retrieved in constant time. A single line segment can only be represented in a compressed bit vector as one of four possible sequences of words. Table 7.2 shows the four possible sequences with examples (assuming 8 bit words for simplicity).

We show here that the query line in each of the four sequences can be retrieved in constant time.

1. The line segment in sequence 1 can be retrieved in constant time by decoding the 1 fill.

2. The line segment in sequence 2 is spread over three different words, two literals with a fill in the middle. If either of the literals has more than 6 bits 1 ($n-1$ where n is the number of bits in a word, in the general case), then it would be compressed into the fill word. Thus we have to convert the fill word into a segment in constant time and then extend the line in both directions using the 1s in the literal words. Since the total number of 1s is a constant 12, we can retrieve this query line in constant time.

3. The line segment in sequence 3 is spread over two words. The maximum number of 1s in these two words is thirteen, as fourteen consecutive 1s would have been compressed into a fill word. This implies that a query line can be retrieved from this sequence in constant time.

4. The line segment in sequence 4 can have at most seven 1s and hence can be retrieved in constant time.

A new query line in a WAH compressed bitmap index can either begin in the word where the previous query line ends, in the next word, or be separated from the previous query line using a single 0 fill. In all three cases we can move from one query line to next in constant time.

We have shown that we can retrieve a query line in constant time and can move from one query line to other in constant time. Hence the total time to retrieve S query lines using GETLINESEGMENTS is $O(S)$. □

Once we have found all the line segments, we need to split them into query lines. Figure 7.5 shows the algorithm for performing this split operation. The algorithm takes a sorted sequence of line segments and mesh lines as input and returns a set of query lines as output. GETLINESEGMENTS iterates over the query lines and mesh lines, and whenever a line segment spans multiple mesh lines we split it to ensure that none of the resulting query lines spans multiple mesh lines. The query lines and the mesh lines are sorted in the same order, so that we can take advantage of the ordering of mesh lines as will be described in Section 7.4. The complexity of this algorithm is $O(L + M)$, where L is the number of line segments and M is the number of mesh lines. The number of mesh lines is independent of the number of points in the query result and hence is a constant. Thus the run time of the algorithm is $O(L)$.

Since both GETLINESEGMENTS and GETQUERYLINES are $O(L)$, the entire process for converting the bit vector to query lines is $O(L)$. Since $L \leq S$, where S is the number of query lines, we conclude that the algorithm to convert compressed bit vector query result to query lines is $O(S)$.

7.3 Region Growing Phase

To use a connected component labeling algorithm to consolidate query lines into regions, we need to first convert the mesh connectivity information into a representation that efficiently supports graph operations to find the neighbors of a vertex. We use an adjacency list representation of the mesh graph, because we can find the neighbors of a vertex in constant time. For unstructured

```
GETQUERYLINES(LSegs[L], MLines[M])
  QLines[M][] //three dimensional query lines array
  x, y, z ← 0
  for i ← 0 to L + M
      if (LSegs[x].start > MLines[y].end
          QLines[y] ← NULL
          y ← y + 1
      else if (LSegs[x].end < MLines[y].end)
          QLines[y][z].start ← LSegs[x].start
          QLines[y][z].end ← LSegs[x].end
          x ← x + 1
      else if (LSegs[x].end > MLines[y].end)
          if (LSegs[x].start ≥ MLines[y].start
              QLines[y][z].start ← LSegs[i].start
              QLines[y][z].end ← MLines[y].end
              z ← 0
              y ← y + 1
          else if (LSegs[x].end > MLines[x].end)
              QLines[y][z].start ← MLines[y].start
              QLines[y][z].end ← MLines[y].end
              z ← 0
              y ← y + 1
          else
              QLines[y][z].start ← MLines[y].start
              QLines[y][z].end ← LSegs[x].end
              z ← z + 1
              x ← x + 1
```

Figure 7.5. Splitting the line segments into query lines.

tetrahedral meshes, we must materialize the entire adjacency list beforehand, while for structured meshes the neighbors can be found (computed) quickly on the fly. For semi-structured meshes, some neighbors can be efficiently computed and some not. So we materialize an adjacency list containing only the neighbors that can not be computed on the fly. Creation of the adjacency list takes time $O(E)$, where E is the number of connections in the mesh. The adjacency list is materialized once per mesh and is used for all queries, so we do not include its computation time in query response time.

7.3.1 Union-Find

We make extensive use of a union-find data structure in our region growing phase. The union-find data structure represents a set of disjoint sets, with each set identified by a representative element. The union operation takes two elements and computes the union of the two sets that they belong to. The find operation takes an element, e, as input and returns the representative element of the set that e belongs to. The data structure is initialized with each element contained in a separate set.

Conceptually, we represent the union-find data structure as a forest with each tree representing one disjoint set, and the root of the tree being the representative element (label) for the set. To take the union of two sets, we point the root of one tree to the other. For the find operation, we return the root of the tree that contains the element being searched for. We implement the tree using an array as our storage data structure as shown in Figure 7.6. The smaller of the two roots is always made the root of the new united set, ensuring that $label[i] \leq i$. This constraint means that the Find algorithm moves in one direction in the array, improving cache performance. Also, by altering the label of all the points in the forest in the LABEL method we perform path compression. With path compression the amortized cost of doing N Unions is $O(N)$ [CLRSb]. We have chosen an array-based implementation because current day architectures are more conducive to arrays than pointer based structures, because pointers lead to pointer chasing which leads to poor cache behavior.

Figure 7.7 contains an example set of disjoint sets. The set is represented as a forest on the top and its array representation is shown at the bottom. Table 7.3 shows the state of the array during

```
CLASS UNIONFIND
  labels[1 ... nNodes]

  UNIONFIND(nNodes)
        for i ← 1 to nNodes
              labels[i] = i

  FIND(u)
        while (u < labels[u])
              u ← labels[u]
        return u

  LABEL(u, label)
        while (u < labels[u])
              temp ← labels[u]
              labels[u] = label
              u = label
        labels[u] = rootv

  UNION(u, v)
        rootu = Find(u)
        rootv = Find(v)
        if (rootu < rootv)
              label(v, rootu)
        else
              label(u, rootv)
```

Figure 7.6. The union-find algorithm.

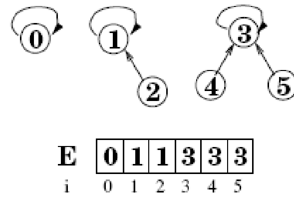


Figure 7.7. The union-find array.

Array Index	0	1	2	3	4	5
Initialization	0	1	2	3	4	5
Adding edge 1-2	0	1	1	3	4	5
Adding edge 3-4	0	1	1	3	3	5
Adding edge 4-5	0	1	1	3	3	3

Table 7.3. Operations on the Union-Find Array.

the labeling process from initialization to the final state. The first row shows the initialized state, while each subsequent row shows the state of the array after the addition of one edge.

7.3.2 Region Growing

Once we have converted the result bit vector into query lines, we are ready for our next phase, consolidating the query lines into regions. At this point the input to our region growing algorithm is a graph representation of the mesh G, and a set S_L of query lines L.

The output of the region growing phase is a set S of disjoint sets CL_i. Each set CL_i consists of the query lines that are connected in the underlying mesh and have label i. To compute the CL_i sets, we must solve the graph connected component labeling problem. We extend the basic connected component labeling algorithm shown in Figure 7.8 to three different versions, called POINTS, LINES and LINES&ORDERING. The POINTS is a simple restatement of the basic algorithm for our data structure. The LINES algorithm takes advantages of the query lines and mesh lines to improve query performance. The LINES&ORDERING algorithm takes advantage of mesh ordering properties along with the query and mesh lines to improve performance. The details of these algorithms will be discussed in the next few sections.

```
CONNECTEDCOMPONENTLABELING(G)
  for each vertex v ∈ V[G]
      MakeSet(v)
  for each edge(u, v) ∈ E[G]
      if Find(u) ≠ Find(v)
          Union(u, v)
```

Figure 7.8. The connected component labeling algorithm.

```
POINTS(G, Q)
  UnionFind u(||Q||)
  for each point p ∈ Q
      for each neighbor n of p in G
          if u.Find(p) ≠ u.Find(v)
              u.Union(u, v)
```

Figure 7.9. The modified connected component labeling algorithm.

POINTS: Using Query Result Points

Figure 7.9 shows the POINTS algorithm, which eschews the advantage of efficient query line retrieval afforded by WAH compressed bitmap indexes. Instead, the algorithm uses the query result points as the nodes for the union-find array. Since POINTS does not take advantage of the lines returned by the WAH compressed bitmap indexes, we treat it as the baseline algorithm in our experiments. The set Q is the set of nodes that are in the query result.

POINTS runs in time $O(E)$, where E is the number of edges contained in the graph formed by the query result points.

LINES: Using Query Result Lines

Figure 7.10 shows the LINES algorithm, which takes advantage of query lines and mesh lines to improve query performance. The input to the algorithm is the mesh graph G, the query lines L, and the mesh lines M. The union-find data structure is initialized with the query lines rather than the query points. This considerably reduces the size of the data structure and hence improves performance. For each point p in the current line, we look at all its mesh neighbors that precede the mesh line that p belongs to. We do not need to consider every neighbor of p, because no two

```
CLLLINES(G, L, M)
   UnionFind u(||L||)
   for each line l ∈ L
         for each point p ∈ l
               m ← mesh line that p belongs to
               for each neighbor n of p that is < m.start
                     if (n is not in the mesh line of p)
                           if u.Find(QL[n]) ≠ u.Find(QL[p])
                                 u.Union(QL[n], QL[p])
```

Figure 7.10. The CLLLines algorithm.

query lines in the same mesh line can be directly connected. The linearization of the mesh graph allows us to find all query lines that belong to the same region by just checking p's neighbors in lines that precede p's line. This shortcut affords us further performance improvement. LINES also maintains a hash table QL which when provided a point's ID returns the identifier of the query line that that point belongs to.

The running time of LINES algorithm is still $O(E)$ in the worst case, where E is the number of edges in the graph formed by the query result points. However, as the number of points in the result grows, so does the size of the query lines and hence the number of union operations goes down.

LINES&ORDERING: Using Query Result Lines & Mesh Ordering

The preceding algorithms use optimizations afforded by a simple linearization of the mesh to make region consolidation fast. Information about the ordering of connectivity of points between mesh lines can also be used to obtain further improvement in performance. For example, if we consider a single plane in a semi-structured toroidal mesh [CMH+04, Lee87] used in gyrokinetic simulations, we can garner further information that can be used to improve performance. A toroidal mesh consists of a series of circular planes connected to form a donut (see Figure 7.1(c)). Each plane consists of a series of concentric circles containing increasing numbers of points. Figure 7.11 shows a single plane in such a mesh, where each point p is connected to the following:

- The points closest to p in the next upper and lower circles in the same plane.
- The two points adjacent to p in the same circle.
- The points closest to p in the same circle in the previous and next planes.

Given the above connections, we define connectivity between the points in the first two rings in Figure 7.11. The set E of edges is $\{$(6, 1), (5, 0), (4, 0), (11, 3), (10, 3), (9, 2), (8, 2), (7, 1), (0, 1), (1, 2), (2, 3), (3, 0), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9), (9, 10), (10, 11), (11, 4)$\}$. We also define one circle to be a single mesh line. Thus, if we get a query result $\{$1, 2, 4, 7, 8, 9, 11$\}$, shown in Figure 7.11 as the two encircled regions, the corresponding query lines we would get would be $\{\{$1-2$\}$, $\{$4$\}$, $\{$7-9$\}$, $\{$11$\}\}$.

As in the LINES algorithm, LINES&ORDERING checks connectivity only for points in preceding mesh lines. For the innermost ring we have no such points. Hence our labeling starts with the second ring and checks for connectivity in the innermost ring. The first line in ring 2 is $\{$4$\}$, which is not connected to any query line in ring 1 and the only point it is connected to in ring 1 is point 0, which is smaller than the points in the query lines in the first ring. Hence we move to the next query line in ring 2, i.e., $\{$7-9$\}$. The smallest point connected to 7 is 1, and the largest point connected to 9 is 2. The only query line that the line $\{$7-9$\}$ connects to is $\{$1, 2$\}$. We unite $\{$7-9$\}$ and $\{$1, 2$\}$ using the union-find data structure. Since the largest point that query line $\{$1, 2$\}$ is connected to is 9, we can rest assured that no future query lines are connected to this query line. We are ready move to the next query line in ring 1, and find that there are no more. Thus by alternately moving forward in mesh lines 1 and 2, we only look up each pair of connected components at most once.

The previous example shows that when the connectivity between consecutive mesh lines is ordered (i.e., if node A precedes node B in mesh line 1 then the node connected to node A in mesh line 2 precedes the node connected to node B in mesh line 2) we can use the ordering to further improve performance. When we can define mesh lines and the ordering of connections between points in consecutive mesh lines, we can do region growing using the algorithm LINES&ORDERING shown in Figure 7.12. The method GETCONNECTEDMESHLINEIDS(i) returns a list of IDs of all the

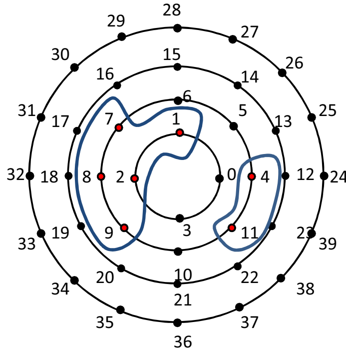


Figure 7.11. One plane in a gyrokinetic mesh, with two regions of interest.

mesh lines that are connected to mesh line i (meaning they have points that are connected to each other), and have IDs less than i. As before, QL is a hash table providing a unique identifier for each query line.

Since each mesh line is connected to a constant number of other mesh lines, the total run time of LINES&ORDERING is $O(\|L\|)$, where L is the set of query lines. Since $\|L\| \leq N$, we can say that the performance of LINES&ORDERING is at worst $O(N)$. In general $\|L\| < N$, and in such cases the LINES&ORDERING algorithm is sublinear in N.

7.4 Interesting Mesh Orderings

The LINES and LINES&ORDERING algorithms presented in the previous section take advantage of special mesh orderings to improve the performance of the region growing phase. The two salient features of these orderings are that they afford long mesh lines, and that connectivity between points in neighboring mesh lines is ordered, so that it is not necessary to look at all points in the query lines to decide whether two query lines belong to the same region.

We have been able to identify mesh lines with both of the above properties in structured meshes and semi-structured meshes, but have not been able to do so for arbitrary unstructured meshes. However, we believe that such orderings do exist even in unstructured meshes as unstructured meshes also model geometrical shapes. The properties of such shapes could lead to the creation of

```
CLLLines&Ordering(G, L, M)
  UnionFind u(‖L‖)
  for i ← 0 to ‖M‖
      mlIDs[] ← getConnectedMeshLineIDs(i)+
      for each mesh line ID mlID in mlIDs
          Q_cur ← L[mlID]
          Q_prev ← L[i]
          x, y ← 0
          for j ← 0 to ‖Q_cur‖ + ‖Q_prev‖
          n_start ← neighbor of Q_cur[x].start in M[i]
          n_end ← neighbor of Q_cur[x].end in M[i]
          start ← Q_prev[y].start
          end ← Q_prev[y].end
          if (n_end < start)
              x ← x + 1
          else if (end < n_start)
              y ← y + 1
          else if (end < n_end)
              Union(QL[L[i][x]], QL[L[mlID][y]])
              y ← y + 1
          else if (n_end < end)
              Union(QL[L[i][x]], QL[L[mlID][y]])
              x ← x + 1
```

Figure 7.12. The CLLLines&Ordering algorithm.

```
CREATELARGEMESHLINES(M)
   ML[]
   i ← 0
   while (M is not empty)
        ML[i] ← diameter of M
        M ← M − ML[i]
   return ML
```

Figure 7.13. Creating large mesh lines using diameter of a graph.

mesh orderings that possess both of the above properties.

Even in the absence of ordering of connectivity, we can reorder the points in a way that the mesh produces longer mesh lines than would be produced using the native ordering suggested in Section 7.1. To create long mesh lines, we treat the mesh as a graph. In a graph the largest acyclic set of connected points lies on the diameter of the graph (a graph's diameter is the largest number of vertices which must be traversed in order to travel from one vertex to another when paths which backtrack, detour, or loop are excluded from consideration). The algorithm CREATE-LARGEMESHLINES for creating large mesh lines based on graph diameters is shown in Figure 7.13. The operation $M - ML[i]$ removes of all points that are in $ML[i]$ from the graph M. Graph diameter computation algorithms are $O(V(E+V))$ and hence iteratively computing diameters of graphs becomes $O(V^2(E+V))$, which is too expensive for large meshes, even as a preprocessing step.

We approximate CREATELARGEMESHLINES by using a *modified depth first search (DFS)*. Figure 7.14 shows the algorithm REORDERMESH for reordering the graph. The algorithm does a DFS starting from the first point, and changes the point IDs to reflect the order in which they were visited. As soon as a point whose addition would create a cycle is visited, the DFS stops. The DFS then chooses a point that has yet not been visited before and continues until it has visited each point in the mesh. This algorithm visits each edge at most once. For an unstructured mesh with 763,395 points used by the Center for Simulation of Advanced Rockets at UIUC, the native ordering provided by the mesh produced 505,273 different mesh lines, with the largest mesh line size of 3 points. With such short mesh lines in the LINES algorithm offers little performance

```
GLOBAL VARIABLES adjList[N][]
adjListSizes[N]
reOrder[N] ← −1
counter ← 0

DFS(point, line)
   for j ← 0 to adjListSizes[point]
         if (reOrder[adjList[point][j]] = -1)
               if (line ∩ adjList[adjList[point][j]] = φ))
                     break;
   if (j ≠ adjListSizes[point])
         reOrder[adjList[point][j]] ← counter
         counter ← counter + 1
         line.push(adjList[point][j])
         DFS(adjList[point][j], line)

REORDERMESH()
   stack line
   for i ← 0 to N
         if (reOrder[i] = -1)
               line.push(i)
               DFS(i, line)
               line.empty()
```

Figure 7.14. Reordering the mesh using a modified DFS algorithm.

improvement over the POINTS algorithm. Once reordered using REORDERMESH, we had only 85,704 mesh lines (despite a large number of singleton lines). As will be shown later, this reordering alone brings up to a 10 fold speed up for LINES algorithm when compared to the POINTS algorithm.

7.5 Experiments

We ran our experiments on a dual core 2.4 GHz machine with 1 GB memory and two 500 GB disks, each with a manufacturer's claimed bandwidth of 100 MB/s maximum. Our experiments use real world data from two different sources.

- Our first set comes from a 250 processor gyrokinetic simulation of plasma at the Princeton

Plasma Physics Laboratory [CMH+04]. The data consists of mesh variables X, Y, Z, and Potential. The data had 4000 timesteps with 9,674,304 mesh points in each timestep. The mesh is a semi-structured toroidal mesh as described in Section 7.3.2.

- Our second data set came from a 64 processor rocket fluid simulation at the Center for Simulation of Advanced Rockets at UIUC. The data consists of 6 different nodal variables, X, Y, Z, Vel-X, Vel-Y, Vel-Z. The data has 46 different timesteps with 763,394 points per timestep. The mesh is an unstructured tetrahedral mesh.

7.5.1 Semi-Structured Mesh

In this experiment we evaluate the performance of the POINTS, LINES and LINES&ORDERING algorithms a set of queries over the plasma simulation. We also show that the LINES&ORDERING algorithm is sublinear with respect to the number of points in the query result.

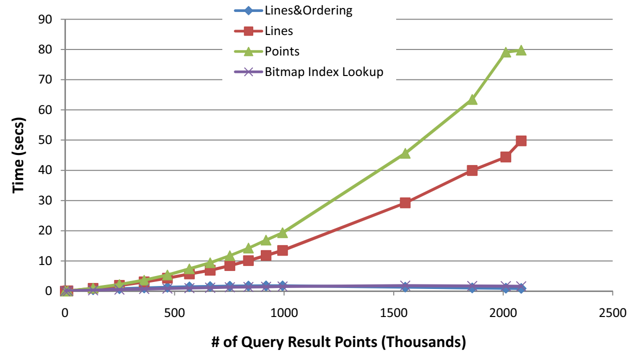
We created a bitmap index on the potential variable for one time step and stored it along with the mesh and the mesh line information. This mesh has three dimensions: the position in a single circle, position of the circle in the plane, and the position of the plane in the toroid. We linearize the mesh starting from the first point in the first circle in the first mesh. Then we move first along the circle then onto other circles in the same plane and then eventually to the next plane.

Figure 7.15(a) shows the total time taken for the three different labeling algorithms, POINTS, LINES and LINES&ORDERING. The figure also shows the time taken for the bitmap index lookup operations. When the number of objects returned by the index lookup algorithm is relatively low, the time taken by the three algorithms is similar and is close to the time taken for the bitmap index lookup operations. As the number of points in the query result increases, the time taken by the POINTS algorithm (the baseline) increases the fastest. The graph sshows that the POINTS algorithm is superlinear while the other two are linear. While LINES and LINES&ORDERING both appear linear, the constants for the two are very different. The graph for the LINES&ORDERING algorithm is almost flat when compared to the other two algorithms. The LINES is twice as fast as the POINTS

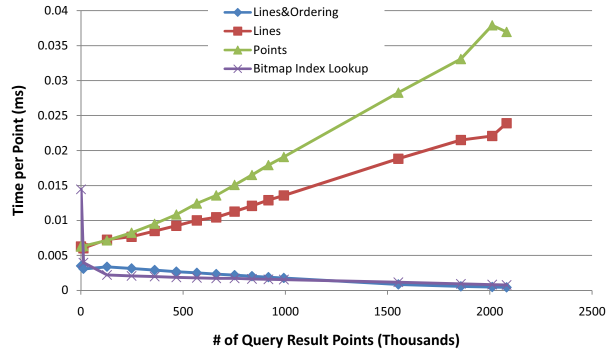
algorithm. The bitmap index lookup operations are also almost flat compared to the LINES and POINTS algorithms. Given that, in most cases the, time to label dominates the total time, improving the run time for the labeling algorithm considerably improves the performance of the entire query. The reduction in possible comparisons due to the mesh ordering in the Lines&Orderings algorithm contributes to its much improved performance. Figure 7.15(b) shows the per point cost for each of the three algorithms. This shows that while the POINTS and LINES algorithms are linear or super linear, the LINES&ORDERING algorithm and the bitmap computation is clearly sublinear in practice.

In the next set of experiments, we study the number of bytes of the bitmap index read into memory for the index lookup, which determine the time spent in index lookup. Figure 7.15(c) shows the size of the portion of the bitmap index for the Potential attribute, that were read into memory. In these queries, we see that the size of the index read is almost the same as predicted in Section 7.1. We see that the sizes of the bitmaps read nearly reach the maximum possible size, $8B + 8N$ for bitmap indexes. This is due to the linearization of the mesh, which is very good for region consolidation but not as good for bitmap compression. Figure 7.16 shows a visualization of the entire gyrokinetic mesh for the Potential variable. Different colors show the different values for Potential. Had we linearized along the lines formed by individual colors we would have gotten much better performance for the index lookup. But the largest possible line would be 64 points, as there are just 64 planes. Our current linearization gives us much larger possible lines in the region growing phase, but gives rather poor bitmap index compression. Overall, this linearization is excellent for finding regions of interest, because, the index lookup represents only a small portion of the total query run time. and the linearization is good for the more expensive region growing phase.

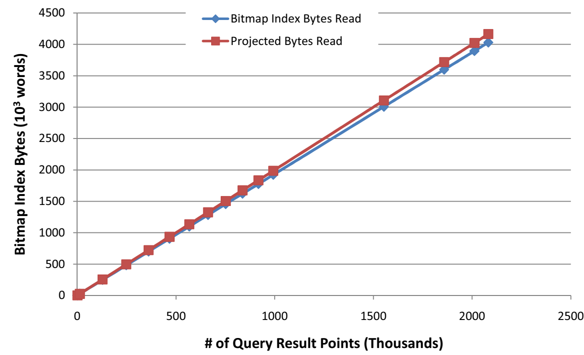
Figure 7.17(a) plots the number of lines in the query result versus the number of points in the query result. The figure shows that as the number of points increases, the number of lines also increases at first. Eventually, the lines start merging faster than new lines are introduced, so the number of lines starts decreases. Figure 7.17(b) shows the total time spent on the constraint



(a) Total query response time.



(b) Query response time per query point.



(c) Size of the index read into memory.

Figure 7.15. Response time for range queries over the Potential variable in the plasma simulations.

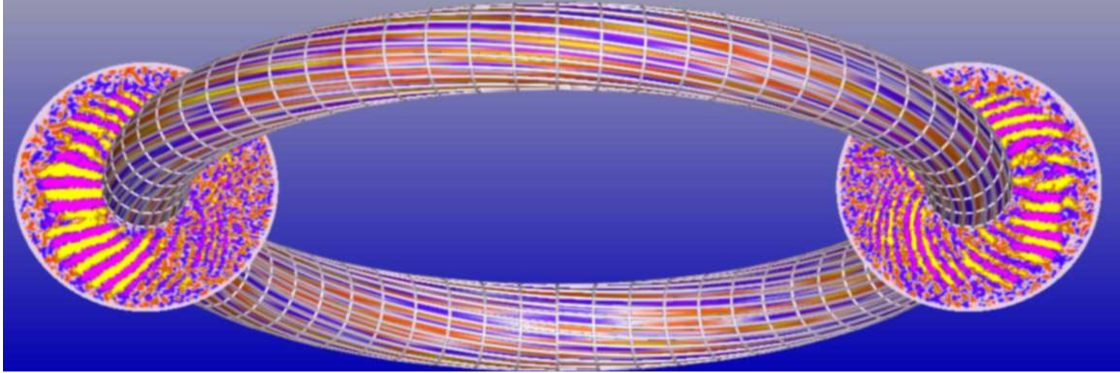


Figure 7.16. Complete Gyrokinetic Mesh.

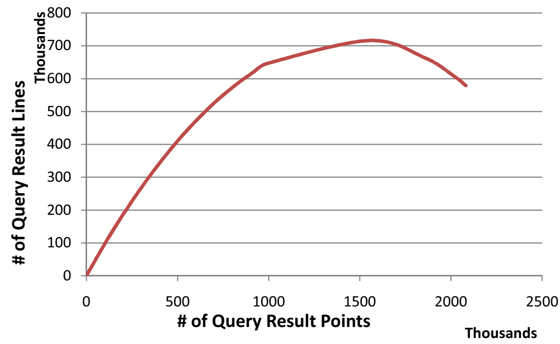
application phase and the LINES&ORDERING algorithm for the region growing phase. The figure shows that the query response time is linear with respect to the number of lines in the answer. Even though the number of result points continues to increase, the number of result lines starts to drop, and the time needed to apply constraints and create regions also drops.

7.5.2 Unstructured Meshes

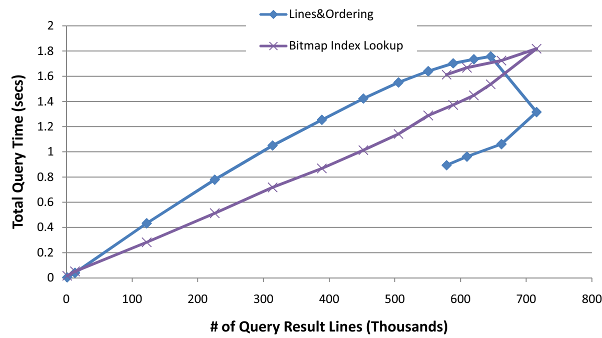
For the rocket simulations unstructured mesh data, we constructed a bitmap index on the Z coordinate for one timestep, stored the mesh explicitly as an adjacency list and materialized the mesh lines using, REORDERMESH algorithm shown in Figure 7.14.

Single Variable Queries

In these experiments we evaluate the difference in time for querying an unstructured mesh using the POINTS and LINES algorithms. We ran a series of range queries on the Z coordinate. The ranges all start at -1.6, and end at points between -1.4 and 1.6, in range increments of 0.1. The results from the queries are summarized in Figure 7.18. As mentioned before, we could not generate an ordering of the connectivity between the mesh lines. Hence the applicable algorithms for region growing were the POINTS (baseline) and LINES algorithms. Figure 7.18(a) shows the time spent on index lookup (same for the POINTS and LINES algorithms), and the time spent on labeling using the POINTS and LINES algorithms. The LINES algorithm far outperforms the POINTS algorithm.



(a) Number of query lines versus number of query points.



(b) Query response time versus number of query lines.

Figure 7.17. Line information for queries over the Potential attribute from the plasma simulation.

This is because the degree of each point in an unstructured mesh is much higher than in a structured or semi-structured mesh, and hence the LINES algorithm's reduction of the number of points to be examined improves the computation time considerably. As the number of query result points approaches the number of points in the mesh, the size of the query lines increases considerably. The choice of region growing algorithm has the greatest impact at this point, for two reasons:

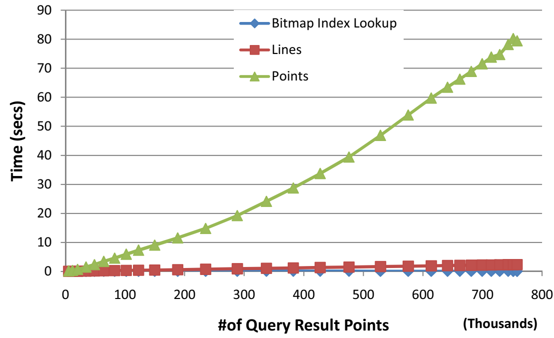
- The union-find data structure is much smaller if it is initialized with query lines rather than query points.
- The number of neighbors outside the query line decreases considerably.

Thus, although unstructured meshes lack an ordering of connectivity between mesh lines, we can still get significant performance benefits from the use of query and mesh lines.

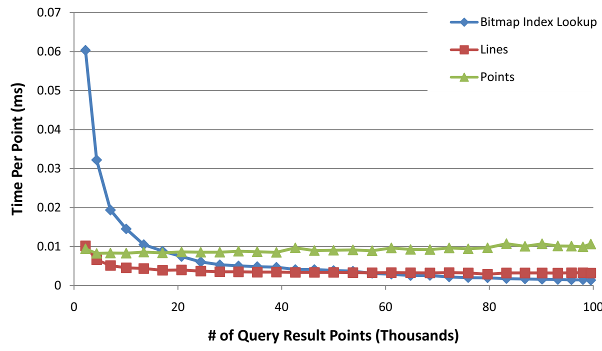
The other important component in query response time is the time spent on bitmap index lookup, which is quite small (as was also the case in the plasma physics experiments). In Figure 7.18(b), the LINES algorithm and bitmap index lookups are sublinear in the number of query result points (more clearly when the number of points are smaller). The POINTS algorithm is clearly superlinear.

Figure 7.18(c) shows a very interesting aspect of bitmap indexes. While the maximum predicted size of the index grows almost linearly, somewhere around the halfway point the amount of index read starts to drop off. At this point, the bitmap index lookup code reads the bit vectors required to answer the negation of the query, and negates the result. This means that even for very large ranges, the index lookup do not take too much time.

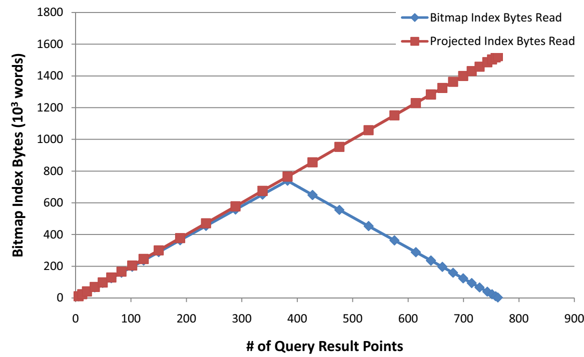
Figure 7.19 shows the performance of region retrieval algorithms for the X and Y coordinates. As before, the green line shows the time taken for the region growing phase using the POINTS algorithm, the brown line shows the time taken for the region growing phase using the LINES algorithm and the blue line shows the time taken for the index lookup. As with the X coordinate this experiment illustrates the considerable performance improvement of the LINES algorithm over the POINTS algorithm. The presence of values from both the X and the Ys coordinate makes the



(a) Total query response time.



(b) Query response time per query point.



(c) Size of the index read into memory.

Figure 7.18. Response time for range queries over the Z coordinate, with rocket simulation data.

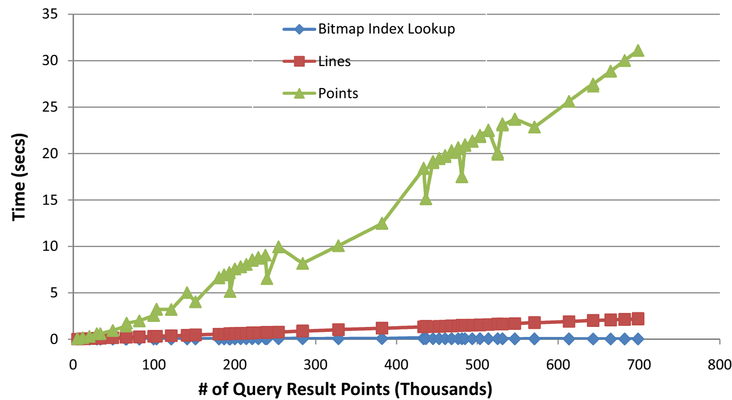


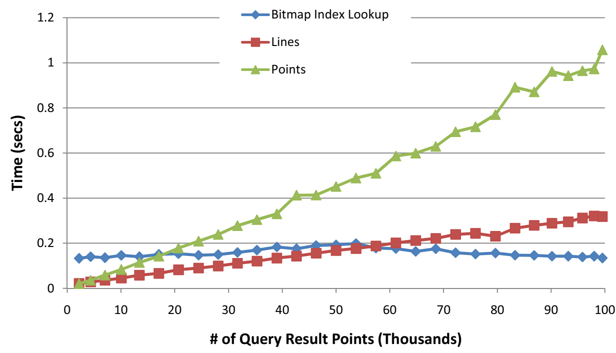
Figure 7.19. Response time for range queries over the X and Y coordinates, with rocket simulation data.

POINTS curve into a composite curve consisting of two distinctive curves. Such behavior is also present in the LINES algorithm and the index lookup, but is masked due to the scale of the graph.

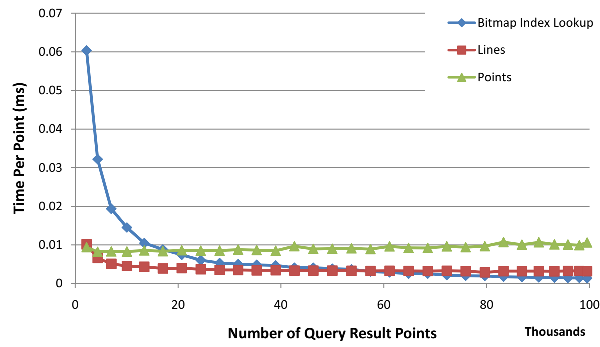
Multiple Variable Queries

Our discussion so far has focussed on single attribute queries. In this set of experiments, we evaluate the performance of queries over multiple attributes. As number of attributesincreases, the bitmap index lookup time will also increase and may become the dominant portion of the entire region retrieval algorithm, making the improvement in performance of the labeling algorithm superfluous.

We keep the ranges in the Z coordinates the same as in the single variable case, and add the ranges [0, 1.2], and [-.4, .4] for the X and Y coordinates, respectively, for each query. Figure 7.20(b) shows the results of this experiment. The graph shows that when the number of points returned by the query is low, the bitmap index lookup time dominates the query response time, but as the number of points increases the POINTS algorithm performance drops very rapidly, while the LINES algorithm drops at a much slower rate. Figure 7.20(b) shows the rate of growth of the query response time as a function of the number of points. Figure 7.20(b) shows that both the index lookup and the LINES algorithm are sublinear in the number of result points, while the POINTS algorithm is superlinear.



(a) Time spent over the entire query.



(b) Time spent per query point.

Figure 7.20. Response time for range queries over X, Y, and Z attributes, for rocket simulation data.

Chapter 8

Conclusions

NASA's Earth Observing Satellite produces 3 terabytes of data in a single day. The next generation of satellites launched by NASA will have 10 times higher resolution (in each of the X, Y and Z dimensions), producing petabytes of data in a single day. If we turn our gaze heavenwards, the Sloan Digital Sky Survey has exported a 12 terabyte data set. Simulations do not lag too far behind in terms of data produced. A single complex simulation at the Rocket Center produces 2 terabytes of data in a few days. Similarly, plasma simulations at TechX-Corp can produce upwards of 2 terabytes of data in a week.

With such large amounts of data, scientists are already drowning in the sea of data they have created. Analyzing these data sets to produce novel scientific innovations requires algorithms that can deal with large amounts of data and find subtle changes in the data. To make the problem harder, often scientists need to combine data from a variety of sources to produce interesting discoveries. Trying to extract information from such large amounts of disparate data is like looking for a needle in a haystack.

Luckily, the scientists know what kinds of needles (interesting objects) they would like to find, although they do not know where the needles are. The needles can be defined in terms of the values of their associated variables, e.g., length, width, metal. Thus the haystack of scientific data can be organized into smaller stacks, based on the values of the variables, and then with a magnet the scientists can locate the stacks containing a needle. Finding the needle in one of these smaller haystacks is much easier. In scientific computation, indexes play the role of the magnet that allows the scientists to zoom into the region of data that most interests them.

8.1 Summary

In this thesis we have presented a bitmap based indexing scheme with the following features:

- *Efficient response to multidimensional range queries* - Traditional bitmap indexes have been shown to scale well for queries that restrict many variables. The experiments in this thesis have validated their efficiency at very high dimensionality. However, our early experiments showed that traditional bitmap indexes performed poorly for large range queries. In our interactions with scientists and experience with scientific data and queries, we found that these queries are very common. Our work presents a new multi-resolution version of bitmap indexes that performs well on large range queries too.

- *Small footprint* - While talking to scientists who must manage very large data warehouses (such as those of the Sloan Digital Sky Survey), we found the storage space for indexes in such warehouses is relatively limited. Traditional bitmap indexes are smaller than other indexes, requiring only 4-8 bytes per indexed object attribute. However, today's scientific data warehouses can typically afford only 2 bytes per indexed attribute. In response, we have presented adaptive bitmap indexes, which provide efficient lookups even when occupying only 2 bytes per indexed object attribute.

- *Efficient retrieval of region* - Scientific data often consist of numerous data points that are implicitly or explicitly connected in a mesh. While data are stored as individual points, scientists are more interested in regions, which are sets of interconnected points that share a common interesting characteristic. Thus scientists need an efficient method to retrieve regions of interest in large scientific data sets. Previous methods have at best been linear in the number of interesting data points retrieved, which with such large data sets becomes untenable. We have created an algorithm that scales sublinearly with the number of points in the query result.

In our analysis of scientific data and queries, we found three main features that allow us to efficiently use bitmap indexes to respond to queries from scientists. These features are:

- *Dependency on ORs* - To respond to range queries, bitmap indexes first read the bit vectors corresponding to the values in the range into memory. Then bitwise OR operations are performed on these bit vectors until the final bit vector has 1s at positions which correspond to the points satisfying the query. With large range queries, the number of these bitwise OR operations is very high, leading to unacceptable query costs. Reducing the number of bitwise OR operations is the key to the success of multi-resolution bitmap indexes.

- *Query Patterns* - In our study of the Sloan Digital Sky Survey query log, we found that consecutive scientific queries have very strong locality in terms of range restrictions. Scientists target a particular region of the sky, and iteratively move through this region, looking for new galaxies and stars. This implies that through aggressive caching of query results, we can considerably improve the performance of bitmap operations. This insight underlies the good performance of adaptive bitmap indexes.

- *Word Aligned Hybrid Compression and Mesh Connectivity properties* - WAH compression allows us to efficiently retrieve certain subsets of interconnected points in the query answer, called *query lines*. When combined with specific connectivity properties from the underlying mesh, these query lines allow efficient retrieval of regions of interest from scientific data. This insight underlies our approach to finding regions of interest.

These three insights have been our key to devising effective methods for indexing scientific data.

- *Multi-resolution bitmap indexes* solve the problem of answering large range queries by materializing bit vectors corresponding to large ranges at index creation time. By binning the data into larger ranges and building individual bit vectors on these ranges, we considerably decrease the number of OR operations needed to evaluate the range restrictions. But if we

store just the bit vectors corresponding to these large bins in general, we cannot answer queries without fetching underlying data to filter out false positive matches caused by too-wide bins. Our multi-resolution approach, in which we keep bit vectors corresponding to individual values as well as wide bins, allows us to answer range queries without visiting the underlying data. This makes query processing much faster.

Bitmap indexes handle multidimensional queries by producing a result bit vector for each individual attribute restriction and doing a bitwise AND operation over them. The number of AND operations scales linearly with the dimensionality of the query. Hence as the number of dimensions increases, the performance of multidimensional queries degrades much slower than traditional multidimensional indexes.

We provided a theoretical framework to assist users in choosing the number of resolutions and choosing the bin sizes at each level. Our experiments with two distinct real world data sets and more than 40 different real world queries showed an almost 10 times improvement in query performance for large range queries, compared to traditional bitmap indexes.

- *Adaptive multi-resolution bitmap indexes* - In a large scientific data warehouse where space is at a premium, we can only build an index that is a fraction of the size of the data. We chose to build multi-resolution indexes with locally optimal bin boundaries. Even the lowest level of these indexes is binned. In order to remove false positives during index lookups, we go to the underlying data and create projection indexes for the individual bins that may contain false positives. The strong temporal locality in the values queried by scientists allows us to improve query performance by aggressively caching these projection indexes in main memory. Because of the strong temporal locality, the false positives of subsequent queries can often be found using the cached projection indexes. We build and drop projection indexes based on the query workload and an LRU caching algorithm.

  We also generated a model for scientific queries and used it to validate our aggressive caching mechanism. We ran experiments using query logs and data from the Sloan Digi-

tal Sky Survey as well as synthetic query logs (based on our model) and synthetic data sets (based on modeling of WAH compression). Our experiments showed up to 6 times speedup for queries, compared to a non-adaptive multi-resolution index of the same size. One of the most interesting results of these experiments was that with well-clustered data, the adaptive index that was half the size of the indexed data performed better than an adaptive index of the same size as the indexed data.

- *Retrieving interesting regions* - Linearization of meshes or other spatiotemporal data sets in a manner that produces long mesh lines, i.e., long sequence of points that are connected within the mesh, can allow us to find regions of interest very quickly. If the mesh lines connect to one another in a systematic way, we can exploit this to get even better performance. We can find query lines quickly using WAH compressed bitmap indexes, and combine them quickly into regions by exploiting the properties of mesh lines. We proved that our entire scheme for finding regions of interest is in general sublinear in the the number of points satisfying the query. We reported experiments from two real world simulation mesh data sets. Our results showed that our approach is up to 10 times faster than the traditional methods for finding regions of interest, which scale linearly with the number of points retrieved.

The bitmap index approaches proposed in this thesis are a part of the Maitri data management framework. Maitri is a format agnostic scientific data management system. As proposed in this thesis, at the heart of Maitri lies the block manager, which encapsulates all the format specific information about the data. The other modules of Maitri perform indexing, buffering, metadata management, concurrency control and query optimization and are unaware of the format of the data. The minimal interfaces between the modules allow scientists the freedom to pick and choose the modules they want or replace current implementations with the implementations of their choice.

This thesis has provided a solid indexing approach that is effective for a large class of scientific queries. An earlier thesis has provided a buffer manager for Maitri. Future theses can build on these two components, to realize the entire Maitri vision.

8.2 Future Work

Previous work on bitmap indexes has treated queries as independent of each other and has viewed the index as independent of the other components of the data management system. In fact, however, most queries on scientific data have strong spatiotemporal locality [8]. This can be used to improve query performance by caching and sharing the intermediate bit vectors produced by different queries. Moreover, with high dimensional queries, each additional dimension restriction decreases the number of objects satisfying the query. Yet the cost of finding the objects that satisfy each additional range restriction is the same no matter how many dimensions are queried. If relatively few objects satisfy the restrictions so far, it may be faster to go directly to the data to check the remaining restrictions, instead of using the bitmap indexes.

Thus one aim for future work is to build a query optimizer for bitmap indexes that assumes the following:

- Intermediate results of bitmap operations can be shared across queries.
- Effective interaction of the index with other modules such as the buffer manager can improve the efficiency of data retrieval. For example, given partial index lookup results, we can ask the buffer manager to retrieve the underlying data before the rest of the lookup is complete.

Bitmap indexes may be able to outperform current indexing techniques like the R-tree and oct-tree in responding to nearest neighbor queries. These can be formulated as range queries in the requisite dimensions, coupled with relaxing or constricting the range in each dimension to find the nearest neighbors. Open problems include how much to relax/constrict and which dimensions to relax/constrict. Moreover, in the case of a multi-resolution index, we also need to figure out which resolution should be used for the lookup.

Our results on processor behavior during bitmap index lookups show poor counts for instructions per cycle and poor cache performance. This provides scope for improvement of the cache and processor performance. Most tree based index lookups need to parse a block of data to determine

the next disk block to read into memory or the next memory page to read into cache. In contrast, the next data needed for a bitmap index lookup is fairly predictable, because we know in advance which bit vectors need to be brought into memory. An interesting avenue for future research is to develop bitmap indexes that are able to take advantage of this cache predictability.

References

[ACFT06] Tan Apaydin, Guadalupe Canahuate, Hakan Ferhatosmanoglu, and Ali Saman Tosun. Approximate encoding for direct access and query processing over compressed bitmaps. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 846–857. VLDB Endowment, 2006.

[Ant95] G. Antoshenkov. Byte-aligned bitmap compression. In *DCC '95: Proceedings of the Conference on Data Compression*, page 476, Washington, DC, USA, 1995. IEEE Computer Society.

[AYJ00] Sihem Amer-Yahia and Theodore Johnson. Optimizing queries on compressed bitmaps. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 329–338, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[Bac73] C. Bachman. The programmer as navigator. *Communications of the ACM*, 16(11):653–658, 1973.

[Cha03] Guang-Ho Cha. Bitmap indexing method for complex similarity queries with relevance feedback. In *MMDB '03: Proceedings of the 1st ACM International Workshop on Multimedia Databases*, pages 55–62, New York, NY, USA, 2003. ACM Press.

[CI98] Chee-Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 355–366, New York, NY, USA, 1998. ACM Press.

[CI99] Chee-Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 215–226, New York, NY, USA, 1999. ACM Press.

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 268–279, New York, NY, USA, 1985. ACM Press.

[CKN+99] A. Choudhary, M. Kandemir, H. Nagesh, J. No, X. Shen, V. Taylor, S. More, and R. Thakur. Data management for large-scale scientific computations in high performance distributed systems. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 263–272, Washington, DC, USA, 1999. IEEE Computer Society.

[CLRSa] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw Hill, second edition.

[CLRSb] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 21. Data Structures for Disjoint Sets, pages 498–524. McGraw Hill, second edition.

[CMH+04] David Crawford, Kwan-Liu Ma, Min-Yu Huang, Scott Klasky, and Stephane Ethier. Visualizing gyrokinetic simulations. In *VIS '04: Proceedings of the Conference on Visualization '04*, pages 59–66, Washington, DC, USA, 2004. IEEE Computer Society.

[CT] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, second edition.

[Dep04] The Department of Energy Office of Science Data Management Challenge. http://www.sc.doe.gov/ascr/Final-report-v26.pdf, 2004.

[DST92] Michael B. Dillencourt, Hannan Samet, and Markku Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM*, 39(2):253–280, 1992.

[FG96] Christophe Fiorio and Jens Gustedt. Two linear time union-find strategies for image processing. *Theory of Computer Science*, 154(2):165–181, 1996.

[FIT] http://fits.gsfc.nasa.gov/.

[GG98] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GLNS+05] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[GPF] http://www-03.ibm.com/systems/clusters/software/gpfs.html.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[HDF] http://www.hdfgroup.org.

[HJ05] Charles D. Hansen and Chris R. Johnson. *The Visualization Handbook*. Academic Press, 2005.

[IIS] http://www.iist.unu.edu/ãlumni/software/other/inria/rocq/eng9.htm.

[JGL+92] Rowland R. Johnson, Mandy Goldner, Mitch Lee, Keith McKay, Robert Shectman, and John Woodruff. USD - a database management system for scientific research. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 1–12, New York, NY, USA, 1992. ACM Press.

[JJ06] Magesh Jayapandian and H. V. Jagadish. Automating the design and construction of query forms. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 125–135, Washington, DC, USA, 2006. IEEE Computer Society.

[JL01] Marcus Jürgens and Hans-Joachim Lenz. Tree based indexes versus bitmap indexes: A performance study. *International Journal on Cooperative Information Systems*, 10(3):355–376, 2001.

[Joh99] Theodore Johnson. Performance measurements of compressed bitmap indices. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 278–289, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[KCC+01] T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Salz. Exploration and visualization of very large data sets with the Active Data Repository. *IEEE Computer Graphics and Applications*, 30(2), 2001.

[KF92] Ibrahim Kamel and Christos Faloutsos. Parallel R-trees. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 195–204, New York, NY, USA, 1992. ACM Press.

[Kou00] Nick Koudas. Space efficient bitmap indexing. In *CIKM '00: Proceedings of the Ninth International Conference on Information and Knowledge Management*, pages 194–201, New York, NY, USA, 2000. ACM Press.

[Lee87] W. W. Lee. Gyrokinetic particle simulation model. *Journal of Computational Physics*, 72(1):243–269, 1987.

[LHJ99] Eric C. LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *IEEE Visualization*, pages 355–362, 1999.

[LUS] http://www.clusterfs.com/.

[LYJ06] Yunyao Li, Huahai Yang, and H. V. Jagadish. Constructing a generic natural language interface for an XML database. In *EDBT '06: Proceedings of the 10th International Conference on Extending Database Technology*, pages 737–754, 2006.

[MC99] Ron Musick and Terence Critchlow. Practical lessons in supporting large-scale computational science. *SIGMOD Rec.*, 28(4):49–57, 1999.

[MMNM03] Mikolaj Morzy, Tadeusz Morzy, Alexandros Nanopoulos, and Yannis Manolopoulos. Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes. In *East-European Conference on Advances in Databases and Information Systems*, pages 236–252, 2003.

[MSWJ05] Soumyadeb Mitra, Rishi Rakesh Sinha, Marianne Winslett, and Xiangmin Jiao. An efficient, non intrusive, log based I/O mechanism for scientific simulations on clusters. In *Cluster '05: IEEE Cluster Conference*, 2005.

[MWN+04] Xiaosong Ma, Marianne Winslett, John Norris, Xiangmin Jiao, and Robert Fiedler. Godiva: Lightweight data management for scientific visualization applications. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 732, Washington, DC, USA, 2004. IEEE Computer Society.

[NET] http://www.unidata.ucar.edu/software/netcdf/.

[NTC00] Jaechun No, Rajeev Thakur, and Alok Choudhary. Integrating parallel file I/O and database support for high-performance scientific data management. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, number 57, Washington, DC, USA, 2000. IEEE Computer Society.

[O'N87] Patrick O'Neil. Model 204 architecture and performance. In *HPTS '87: Proceedings of Conference on High Performance Transaction Systems*, pages 40–59, 1987.

[OQ97] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 38–49, New York, NY, USA, 1997. ACM Press.

[PAL+06] Stratos Papadomanolakis, Anastassia Ailamaki, Julio C. Lopez, Tiankai Tu, David R. O'Hallaron, and Gerd Heber. Efficient query processing on unstructured tetrahedral meshes. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 551–562, New York, NY, USA, 2006. ACM Press.

[PAW+05] Beth Plale, Jay Alameda, Bob Wilhelmson, Dennis Gannon, Shawn Hampton, Al Rossi, and Kelvin Droegemeier. Active management of scientific data. *IEEE Internet Computing*, 9(1):27–34, 2005.

[PN06] Joohyoun Park and Jongho Nang. A hierarchical bitmap indexing method for content based multimedia retrieval. In *IMSA'06: Proceedings of the 24th IASTED International Conference on Internet and Multimedia Systems and Applications*, pages 223–228, Anaheim, CA, USA, 2006. ACTA Press.

[PPP] http://www.pppl.gov.

[ROM] http://www-unix.mcs.anl.gov/romio/.

[ROO] http://root.cern.ch/.

[RSW05a] Doron Rotem, Kurt Stockinger, and Kesheng Wu. Optimizing candidate check costs for bitmap indices. In *CIKM '05: Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, pages 648–655, New York, NY, USA, 2005. ACM Press.

[RSW05b] Doron Rotem, Kurt Stockinger, and Kesheng Wu. Optimizing I/O costs of multidimensional queries using bitmap indices. In *DEXA '05: Proceedings of the 16th International Conference on Database and Expert System Applications*, pages 220–229, 2005.

[SCI] http://www.sc-aip.com/.

[SCN+93] Michael Stonebraker, Jolly Chen, Nobuko Nathan, Caroline Paxson, and Jiang Wu. Tioga: Providing data management support for scientific visualization applications. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 25–38, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[SDHS00] Kurt Stockinger, Dirk Düllmann, Wolfgang Hoschek, and Erich Schikuta. Improving the performance of high-energy physics analysis through bitmap indices. In *DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 835–845, London, UK, 2000. Springer-Verlag.

[SDSa] http://www.sdss.org/.

[SDSb] http://www.sdss.org/dr1/.

[SJ02] Qingmin Shi and Joseph JáJá. Efficient techniques for range search queries on earth science data. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 142–151, 2002.

[SMW05] Rishi R. Sinha, Soumyadeb Mitra, and Marianne Winslett. Maitri: A format independent data management system for scientific data. In *SNAPI '05: International Workshop on Storage Network Architecture and Parallel I/Os*, pages 49–57, 2005.

[SMW06] Rishi R. Sinha, Soumyadeb Mitra, and Marianne Winslett. Bitmap indexes for large scientific data sets: A case study. In *IPDPS '06: Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006.

[SR86] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 340–355, New York, NY, USA, 1986. ACM Press.

[SSBW05] Kurt Stockinger, John Shalf, E. Wes Bethel, and Kesheng Wu. Dex: Increasing the capability of scientific data analysis pipelines by using efficient bitmap indices to accelerate scientific visualization. In *SSDBM '05: Proceedings of 17th International Conference on Scientific and Statistical Database Management*, pages 35–44, 2005.

[SSWB05] Kurt Stockinger, John Shalf, Kesheng Wu, and E. Wes Bethel. Query-driven visualization of large data sets. In *IEEE Visualization*, page 22, 2005.

[STMW07] Rishi R. Sinha, Arash Termehchy, Soumyadeb Mitra, and Marianne Winslett. Maitri demonstration: Managing large scale scientific data. In *CIDR '07: Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, pages 219–224, 2007.

[Sto01] Kurt Stockinger. Design and implementation of bitmap indices for scientific data. In *IDEAS '01: International Database Engineering & Applications Symposium*, pages 47–57, 2001.

[SW98] Deborah Silver and Xin Wang. Tracking scalar features in unstructured datasets. In *VIS '98: Proceedings of the Conference on Visualization '98*, pages 79–86, 1998.

[SW07] Rishi R. Sinha and Marianne Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Transactions on Database Systems*, 32(3):46–84, 2007.

[SWS04] Kurt Stockinger, Kesheng Wu, and Arie Shoshani. Evaluation strategies for bitmap indices with binning. In *DEXA '04: Proceedings of the International Conference on Database and Expert System Applications*, pages 120–129, 2004.

[TG] http://www.truegrid.com/pistonl.html.

[VTu] http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm.

[WB98] Ming-Chuan Wu and Alejandro P. Buchmann. Encoded bitmap indexing for data warehouses. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 220–230, Washington, DC, USA, 1998. IEEE Computer Society.

[WKA05] C.-L. Wu, J.-L. Koh, and P.-Y. An. Improved sequential pattern mining using an extended bitmap representation. In *Proceedings of the International Conference on Database and Expert System Applications*, pages 776–785, 2005.

[WLO+85] Harry K. T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit transposed files. In *VLDB '85: Proceedings of the 11th International Conference on Very Large Data Bases*, pages 448–457, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

[WLO+86] Harry K. T. Wong, Jianzhong Li, Frank Olken, Doron Rotem, and Linda Wong. Bit transposition for very large scientific and statistical databases. *Algorithmica*, 1(3):289–309, 1986.

[WOS02] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 99–108, Washington, DC, USA, 2002. IEEE Computer Society.

[WOS04] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 24–35, San Francisco, CA, USA, 2004. Morgan Kaufmann Publishers Inc.

[WOS06] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[WOSN02] Kesheng Wu, Ekow J. Otoo, Arie Shoshani, and Henrik Nordberg. Notes on design and implementation of compressed bit vectors. Technical report, Lawrence Berkeley National Laboratory, 2002.

[Wu99] Ming-Chuan Wu. Query optimization for selections using bitmaps. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 227–238, New York, NY, USA, 1999. ACM Press.

[WY98] Kun-Lung Wu and Philip S. Yu. Range-based bitmap indexing for high cardinality attributes with skew. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, pages 61–67, Washington, DC, USA, 1998. IEEE Computer Society.

[YKM99] Haruo Yokota, Yasuhiko Kanemasa, and Jun Miyazaki. Fat-Btree: An update-conscious parallel directory structure. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, page 448, Washington, DC, USA, 1999. IEEE Computer Society.

Appendix A

Theorem 8.2.1. $\frac{\log \binom{n}{\frac{n}{2}}}{2*\log \binom{n}{\frac{n}{m}}}$ *can be approximated as* $\frac{\log 2*m}{\log m}$.

Proof: In order to reduce $\frac{\log \binom{n}{\frac{n}{2}}}{2*\log \binom{n}{\frac{n}{m}}}$ we use Stirling's approximation, which states that

$$n! \approx \sqrt{2\pi n}(\tfrac{n}{e})^n.$$

Using this approximation, we can approximate $\log \binom{n}{k}$ to the following:

$$\log \frac{\sqrt{2\pi n}(\frac{n}{e})^n}{\sqrt{2\pi(n-k)}\sqrt{2\pi k}(\frac{n-k}{e})^{n-k}(\frac{k}{e})^k}$$
$$= \log \sqrt{\tfrac{n}{2\pi k(n-k)}} + n\log n - (n-k)\log(n-k) - k\log k$$

For the numerator, substituting $\frac{n}{2}$ for k we get:

$$= \log \sqrt{\tfrac{2}{\pi n}} + n\log n - \tfrac{n}{2}\log\tfrac{n}{2} - \tfrac{n}{2}\log\tfrac{n}{2}$$
$$= \log \sqrt{\tfrac{2}{\pi n}} + n\log n - n\log\tfrac{n}{2}$$
$$= \log \sqrt{\tfrac{2}{\pi n}} + n\log n - n\log n + n\log 2$$
$$= n + \log \sqrt{\tfrac{2}{\pi n}}$$
$$\approx n$$

For the denominator, substituting n/m for k we get:

$$= \log \sqrt{\tfrac{m}{2\pi n}} + n \log n - \tfrac{n}{m} \log \tfrac{n}{m} - (n - \tfrac{n}{m}) \log (n - \tfrac{n}{m})$$
$$= \log \sqrt{\tfrac{m}{2\pi n}} + n \log n - \tfrac{n}{m} \log \tfrac{n}{m} - n \log (\tfrac{n(m-1)}{m}) + \tfrac{n}{m} \log (\tfrac{n(m-1)}{m})$$

Since m is very large, $\frac{m-1}{m}$ can be approximated as 1. Thus the above formula becomes:

$$= \log \sqrt{\tfrac{m}{2\pi n}} + n \log n - \tfrac{n}{m} \log n + \tfrac{n}{m} \log m - n \log n + \tfrac{n}{m} \log n$$
$$\approx \tfrac{n}{m} \log m$$

Thus the original expression is reduced to:

$$\frac{n}{\frac{n}{m} \log m}$$

which is

$$\frac{m}{\log m}.$$

□

Author's Biography

Rishi Rakesh Sinha was born in Ranchi, India. After studying for two years at IIT Roorkee he transferred to the State University of New York at Stony Brook in 2000. At Stony Brook he graduated as the valedictorian of the class of May 2002, with a B. S. degree in Computer Science. He then moved to the University of Illinois at Urbana–Champaign to pursue graduate education and joined the Database and Information Systems group in August 2002. In December of 2004, he finished his Master's thesis on entity integration. Thereafter, his work on scientific data management led to him becoming a CSE Fellow at UIUC for the academic year 2006-2007. For his Ph.D. thesis, Rishi proposed, implemented and evaluated indexing schemes to retrieve interesting objects from large scientific datasets using small footprint indexes. He spent two semesters at the Data Management Group at Lawrence Berkeley National Lab working on indexing schemes. He also worked on extending the HDF5 data access library to improve its data retrieval efficiency.