Bitmap Indexes for Large Scientific Data Sets: A Case Study

Rishi Rakesh Sinha, Soumyadeb Mitra, Marianne Winslett
University of Illinois, Urnana Champaign
Dept. of Computer Science
201 N Goodwin Ave, Urbana, IL 61801-2302
{rsinha, mitra1, winslett}@cs.uiuc.edu

Abstract

The data used by today's scientific applications are often very high in dimensionality and staggering in size. These characteristics necessitate the use of a good multidimensional indexing strategy to provide efficient access to the data. Researchers have previously proposed the use of bitmap indexes for high-dimension scientific data as a way of overcoming the drawbacks of traditional multidimensional indexes such as R-trees and KD-trees, which are bulky and whose performance does not scale well as the number of dimensions increases. However, the techniques proposed in previous work on bitmap indexes are not sufficient to address all problems that arise in practice. In experiments with real datasets, we experienced problems with index size and query performance. To overcome these shortcomings, we propose the use of adaptive, multilevel, multiresolution bitmap indexes, and evaluate their performance in two scientific domains. Our preliminary experiments with a parallel query processor and index creator also show that it is very easy to parallelize a bitmap index.

1 Introduction

Scientific applications can be broadly classified into observation/experiment driven, simulation driven, and information driven [1]. All three types can easily produce or consume terabytes of data. For example, observational applications like the Moderate Resolution Imaging Spectroradiometer (MODIS), one of many instruments on board the Earth Observing Satellite (EOS), produces 144 MB/hour just for its calibrated radiances 5km sub-sampled data. At the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois at Urbana-Champaign (UIUC), a complex parallel simulation run produces hundreds of gigabytes of output data. Other UIUC scientists use 7GB working sets of NASA satellite data to study vegetation and climate patterns in the Appalachians [5]. Future scientific applications will be even more data-intensive, due to improved observation-gathering technology and more complex and larger-scale simulation codes.

To address their data management issues, domain scientists have collaborated with computer scientists to develop a series of file formats and associated user-level I/O libraries. These formats store data together with their associated metadata, and the I/O libraries provide APIs for efficient storage and retrieval of the data. Today most scientific data is stored in such formats and accessed through the associated I/O libraries. The formats range from the generic (e.g., HDF[1] and netCDF[2]) to the domain-specific (e.g., ROOT[3] for high energy physics and FITS[4] for astronomy). The most popular libraries support both parallel and sequential access.

Unfortunately, data retrieval in these libraries is still navigational [2]. In other words, to retrieve a particular subset of the data, the scientist must write code to read through all the related files and determine which part of the data is needed. The code to perform this search is very specific to the format, and must be rewritten if the scientist moves to a new format. If the scientist uses data from multiple sources, the code must use the appropriate API for each source, which imposes a significant learning curve. Further, these APIs are for operations on individual files, but scientists usually have to deal with a huge number of files. (For example, a simulation of medium complexity at CSAR produces 12069 HDF files.) The scientists must manage the file-level metadata themselves, typically through a complex file naming framework. The libraries associated with

[1] http://hdf.ncsa.uiuc.edu.
[2] http://my.unidata.ucar.edu/content/software/netcdf.
[3] http://root.cern.ch.
[4] http://heasarc.gsfc.nasa.gov/docs/heasarc/fits.html.

1-4244-0054-6/06/$20.00 ©2006 IEEE

these APIs provide facilities for buffering and caching inside a single file, but not across multiple files. To the best of our knowledge, none of the libraries supports indexes even for a single file of data, although the HDF group will provide single-file support in a future release.

Other researchers have recognized the need for multi-file, multi-site, and even multi-format data management facilities to assist scientists. For example, the Globus Alliance[5] is looking into issues related to data migration and replication in large grid applications. The Storage Resource Broker (SRB) project[6] and the openDAP infrastructure[7] both offer architectures to allow scientists to collaborate and manage remote, distributed data through a unified framework. Both provide simplified interfaces to query stored data, although their interfaces restrict queries to range over the contents of a single file. These systems do not provide buffering or indexing facilities. Other researchers have investigated metadata management issues for scientific computing, both for a single site and for Grid applications [6, 4, 9]. When a relational database is used to store the metadata, the database indexing facilities can be used to index the metadata (although not the data itself), which can be very helpful in speeding up user queries.

In this paper, we focus on the problem of indexing scientific data stored in its native formats. As previously documented by others [1, 12], desiderata for indexing approaches for scientific data include the ability to index very large amounts of data; good run-time performance for index creation, and (high-dimensional) queries; and reasonable size. To this list, we add the ability to index across individual file, directory and data site boundaries, so that scientists can find the desired data no matter where it is located; support for both parallel and sequential access, because parallel codes are important in scientific computing; and data format independence, so that scientists can use a single indexing facility to find all data of interest, regardless of how data are stored. In the remainder of this paper, we analyze how well traditional indexing schemes satisfy these desiderata, and conduct an in-depth examination of the most promising candidate, bitmap indexing, in two scientific applications: rocket simulation and vegetation pattern studies. We compare our empirical results with the predictions of the previous work on bitmap indexing for scientific data [12, 11, 10], which was primarily analytical in focus. In particular, we found problems with bitmap size and runtime query performance that were not addressed by previous work. To overcome those problems, we propose the use of adaptive, multilevel, multi-resolution bitmap indexes, and present the experimental results from the first steps toward using these refinements in the rocket simulation and vegetative pattern domains. We also provide an elementary framework for parallel creation and querying of bitmap indexes, and evaluate the scalability of the framework in experiments with Voyager, a parallel visualization tool used by the rocket scientists at CSAR.

[5] http://www.globus.org/solutions/tgcp/.
[6] http://www.npaci.edu/DICE/SRB/.
[7] http://www.opendap.org/.

2 Background and Related Work

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 queries use time and space as high selectivity attributes, such queries are not very common. For example, high-energy physics produces data sets with 500 *searchable* attributes over billions of particles [1], where no individual attribute has much discriminatory power and a typical query might impose conditions on 40 attributes. As another example, Figure 1 shows the distribution of the Normalized Difference Vegetation Index (NDVI) attribute values in the vegetative cover data set, where the queries are usually over a subset of the 5000 to 10000 range of the NDVI attribute. The figure shows that the selectivity of this attribute is low for typical queries. The other attributes used as typical query selection criteria in this data set have similarly unselective distributions; but when taken all together, the combined selectivity of the attributes is usually high enough[8] for indexes to offer significant potential performance improvement, compared to a sequential scan of the data.

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, KD-trees, quad trees, oct trees, and so on are effective for indexing a handful of dimensions simultaneously (e.g., three geographic coordinates plus a

[8] Under the independence assumption the final selectivity is a product of individual selectivities

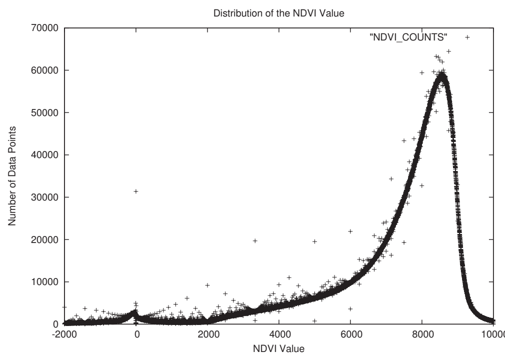


Figure 1. Data Distribution for NDVI attribute in Vegetation Index Dataset. Most of the queries occur over a subset of [5000, 10000].

time dimension). Unfortunately, none of these schemes scales well once more than 15 dimensions are to be indexed, and none of them is very good at handling queries that impose conditions on only (say) half of the attributes in the index. 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.

Moreover, scientific queries typically involve identification of regions of interest, in a two step process [1]. The first step involves searching for objects that satisfy user-defined criteria; the second step involves growing the objects into regions by locating nearby objects. 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 grow those points to include the nearby 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 helps to make the second step fast. Fortunately, bitmap indexes do not require the data to be in any particular order to obtain good performance.

These characteristics have led researchers to suggest the use of bitmap indices for scientific data [12, 11, 10]. Bitmap indexes are known to give good performance for high-dimensionality queries in large data warehouses [7]; are typically smaller in size than B-trees and other traditional indexes over the same amount of data [12]; and can be efficiently updated when new data is appended (e.g., when a new day of satellite observations arrives). Bitmap indexes are known to be poor performers for update operations other than appending new data; that is not a concern in scientific domains, because data is normally read-only once created.

Bitmap indexes store the index information as a sequence of bit vectors and use logical boolean operators to respond to queries. Paul O'Neil is credited with the popularization of these indexes with his work on the Model 204 Data Management System [7, 8]. 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 2(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 un-encoded 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. 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.

The example index contains just 12 objects and 9 different values for the attribute A. A real scientific database might have billions of objects, and the number of values that the object could take would be in the thousands—or much higher, if the value is floating point. For example, the vegetation data set mentioned earlier contains $1.6128 * 10^8$ objects for the Appalachian mountain region alone. This means that each bit vector is approximately 20 MB in size. The value of the Estimated Vegetation Index attribute is stored as short integers that vary between -2,000 and 10,000. Hence there are 12,000 possible values, making the total size of an uncompressed bitmap index 222.69 GB for this attribute, while the amount of data for that attribute is only 305 MB. These numbers show that such a simplistic approach to indexing requires too much space. Moreover, our previous claim that queries will run fast because of low level Boolean operations on bit vectors would not be valid, as reading 1,000 different 20 MB bit vectors and doing Boolean operations on them takes a substantial amount of time.

Analysis of bitmaps shows that there two different directions in which improvements could be made: reducing the size of each bit vector and reducing the number of bit vectors that need to be read to answer

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

Figure 2. Bitmap index basics.

a query. Another important enhancement for scientific data is that of binning the values, so that each bit represents a bin instead of a value. This is very important for very high cardinality attributes and is absolutely vital for floating point data, which are widely used in science.

Researchers have proposed a variety of enhancements in the three directions just described. For example, many encodings have been proposed to decrease the number of bit vectors that must be read for different types of queries. The most interesting of these for our purposes is interval encoding [3], which allows us to answer any range or equality query on one attribute by reading a maximum of two bit vectors. Figure 2(b) shows the interval encoded bitmap for the example in Figure 2(a). In this type of encoding, every bit vector represents a range of numbers instead of a single number. For example, I^0 represents $[0, 4]$, I^1 represents $[1, 5]$, and so on. Any query with range greater than four can be answered by a union of two bitmaps, and a range smaller than four can be answered by intersecting two bitmaps.

Range encoding of bit vectors allows the system to answer an *open ended* range query (e.g., date $<$ 10/1/2005) by accessing at most two bit vectors, in the case when each vector represents one discrete value. This technique can be extended to ensure that the index still works efficiently when each vector represents a range (bin) of values rather than a single value [10]. The observation behind this generic range encoding is that in an open ended range query, only one bit vector (called the candidate vector) has a range where part of the data does not satisfy the target range of the query (a false positive). The query processor reads the objects in that bit vector and removes those that fall outside the target range. 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.

The work in [10] was extended in [11] 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 the three strategies and show that the third one is the best overall.

Compression is an important factor in minimizing the size of bitmap indexes. As explained in [13], the first approach to compression involves using standard compression techniques such as gzip and bzip2 to compress the data when it is written and decompress it while it is being read. This reduces the amount of disk space required; but does not change the total amount of memory required to process the data and increases the time for processing by adding the cost of decompression to it. Other methods have been developed that not only decrease the size of the bit vector but also improve the performance of the querying process. These can be further divided into bit based, byte based, and word based compression. Run-length encoding is

a common type of bit based encoding; PackBits and Byte-aligned Bitmap Code are examples of byte based encoding; and hybrid run-length encoding and word-aligned hybrid (WAH) run-length encoding are examples of word based encodings. A detailed study of these techniques can be found in [13]. Overall, WAH encoding appears to be the best choice for compressing scientific bitmap indexes.

In the vegetation mapping data set mentioned earlier, most of the queries are range queries with ranges greater than 1000. If we store a single bit vector for each of these attribute values, the index performance will be very poor. Even if we used interval encoding to reduce the number of bitmaps that need to be read, we still need to store all the bitmaps, which will be very large. In such a case, the recommended method is to divide the data values into bins, and index each bin with a single bit vector. This considerably reduces the number of Boolean operations that must be performed to answer a range query. The problem with binning data is that not all queries will specify ranges that align with the bin boundaries; misalignment requires us to read the actual data to remove the false positives. While this is expensive, the resultant reduction in requirement of storage space compensates for the expense.

In [12], the authors study the behavior of WAH and BBC compressed bitmaps. The authors suggest a variety of methods to operate on bit vectors to improve performance of range queries over attributes with high cardinality. The authors show that bitmap indexes are very effective even in high cardinality attributes, as long as the indexes are properly compressed and an appropriate evaluation strategy is used while executing the query. The authors also prove that the size of a WAH-compressed bitmap index is $8N + 8b$ bytes, where N is the number of objects being indexed and b is the number of bit vectors. In contrast, in commercial database systems, a B-tree index usually occupies $10N$ to $15N$ bytes, which is larger than the theoretical maximum for a WAH compressed bitmap index.

Researchers have reported the size of bitmap indexes for a combustion data set [12, 11], but to date no detailed application study has been performed. Our goal in this paper is to fill that gap.

3 Bitmap Index Manager Design and Implementation

Our bitmap index manager implementation supports indexing individual objects and also user-defined logical agglomerations of objects, called *blocks* in this paper. We included both levels of abstraction because we want to be able to index across multiple files and multiple sites, and we recognize that the granularity of indexing will vary across files and sites. For example, one site may choose to index only at the file level, while another may include detailed indexes for some or all of the internal contents of its files. In the long run, we envisage a hierarchy of bitmap indexes that a scientist can use to identify the sites that may contain relevant data, then the sections of those sites that may contain relevant data, followed by specific directories and then specific files that may contain relevant data. If an index is available at the finest level of detail, a scientist can find the specific objects that satisfy the query. The data owners may choose not to index their data beyond a certain level of detail.

In block level indexing, we treat the entire block as though it were a single object with multiple values. In this case the number of objects and hence the number of bits in the bit vectors goes down in proportion to the number of objects in the block. Usually each block will contain thousands of objects, causing the block level index to be much smaller than the object level index. The problem with such an index is that the encodings described in the previous section, which were developed to reduce the number of bit vectors that need to be read to answer a single query, are no longer valid. All these encodings assume that each object has only one value for the attribute, thus causing each row in the unencoded bitmap to have just one bit active. This is no longer the case with block level indexing, causing the previous encodings to be no longer valid.

Block-level indexing also complicates query processing, because a block satisfying a conjunctive condition $p_1 \wedge p_2$ may contain no single object that satisfies both p_1 and p_2. Thus either a lower-level index for the block must be consulted to determine whether any individual objects match the query conditions, or else the block itself must be parsed and its individual objects examined to determine whether any individual object satisfies $p_1 \wedge p_2$.

We use an extended version of bitmap indexes proposed by [12] that can index floating point data, which are common in scientific data sets, as well as categorical data. Floating point data are first binned, and then one bit vector is created for each bin. We adopt WAH encoding to compress the bit vectors. The index manager is written in C++ and interfaces with the buffer manager and metadata manager components of our Maitri data management system, whose overall goal is to provide efficient, easy-to-use data-format-independent data management facilities for scientists.

4 Experiments

4.1 Vegetation Pattern Domain

Our experiments employed a data set used by UIUC scientists studying vegetation patterns in the United States. The data come from the NASA EOS Data Gateway[9], by selecting the vegetation index (VI) files collected on 8/2/2003 and 8/16/2003 over the Appalachians. The resulting 3.4 GB data set has a simple schema with 10 attributes, each of which is stored in a separate HDF dataset (a 2D array of 4800 * 4800 points) in one of seven HDF files, with each file corresponding to a different geographical region. The data also includes eight quality control attributes that are mapped to a single unsigned short integers and stored as a single dataset. All other attributes were stored as short integers. The experiments were run on a 1.4 GHz Pentium 4 CPU with 512 MB RAM, with data on a local disk with up to 100 MB/sec transfer rate (according to the Maxtor web site). These experiments were run on a uniprocessor machine and used the sequential version of the Maitri query manager to run the queries.

We ran a total of 10 different queries, each of which was recommended by domain scientists. Five of these were two-dimensional queries over the ND Vegetation Index (NDVI) attribute and Vegetation Index Usefulness (VIU) attribute. (Note: the word *index* is part of the attribute name chosen by the scientists, and does not refer in any way to bitmap indexes.) The other five imposed conditions on attributes NDVI, VIU, Vegetation Index Quality , Aerosol Quantity , and Land Water Mass . All the attributes except the NDVI attribute are quality control attributes, which are mapped to a single short and stored in a single 333 MB HDF dataset. The NDVI attribute is also stored in a single 333 MB HDF dataset.

To answer the two-dimensional and five-dimensional queries without using an index, we do a sequential scan over all the files, read both the datasets and respond to the query. This sequential scan takes 75 seconds to complete.

The NDVI value varies between -2000 and 10000, making NDVI a high cardinality attribute. To efficiently respond to queries to this attribute, we used a multi-resolution index. In this case, we built two indexes. In the first index, each bit vector corresponds to one discrete value. In the second index, the values are divided into 1,000 bins and each bit vector represents one of these bins. The quality control attributes had only 16 different values, so no binned index was created for that dataset. The size of the NDVI index with no binning is 865 MB, while the size of the index with 1,000 bins is 718 MB. Both these indexes are much smaller than the theoretical maximum of 8N bytes, which here would be $8 * 1.612 * 10^8 = 1.2$ GB.

[9]http://edcimswww.cr.usgs.gov/pub/imswelcome.

Query Name	Query Condition
Q1	NDVI IN [6000 6100] & VIU = 4
Q2	NDVI IN [6000 6200] & VIU = 4
Q3	NDVI IN [6000 6300] & VIU = 4
Q4	NDVI IN [6000 6400] & VIU = 4
Q5	NDVI IN [6000 6500] & VIU = 4

Figure 3. The set of two-dimensional vegetation queries.

To show the effect of having two levels of indexing, we ran the queries in two different modes. In the first mode, the queries were answered using only vectors from the index with no binning (in the figure these are labeled as "No Binning"). In the second mode, the queries were answered using a combination of vectors from the binned index and the non-binned indexes. For example, for the range 6,000-6,100, the bit vectors for 6,000-6,004 were read from the non-binned index while those for 6,004-6,100 were read from the binned index (referred to as the multi-resolution index in the rest of the paper). The latter mode used many fewer bit vectors, giving much better query performance.

The two-dimensional queries that we executed are shown in Figure 3. The running times of these queries are shown in Figure 5(a). As described before, the queries were run in two different modes, where the first mode answered the queries using only the index with no binning. In this case, as the range of the query increases, the response time also increases dramatically. This is because the logical operations in a bitmap index take time quadratic in the number of vectors being operated upon [12]. In the Multi-resoultion Index approach, we see that the numbers are much improved. This is because the number of bit vectors being operated upon in this case is much lower. The total number of bit vectors being operated upon in each of the queries is shown in Figure 5(b). We see that for Q3, the number of bit vectors being read in the Multi-resolution Index is 37, while the No Binning Index uses 301 bit vectors. Theoretically this reduction in bit vectors would imply a 100-fold improvement in query performance, but the performance gain is of a magnitude less. This is because bit vectors with a total size of 22.22MB (for the Non Binning index) and 21.18 MB (for the Multi-Resolution Index) have to be read before the actual operations can take place. This I/O time reduces the

Query Name	Query Condition
Q1’	NDVI IN [6000 6100] & VIU = 4 & VIQ = 2 & LWM = 2 & AEQ = 0
Q2’	NDVI IN [6000 6200] & VIU = 4 & VIQ = 2 & LWM = 2 & AEQ = 0
Q3’	NDVI IN [6000 6300] & VIU = 4 & VIQ = 2 & LWM = 2 & AEQ = 0
Q4’	NDVI IN [6000 6400] & VIU = 4 & VIQ = 2 & LWM = 2 & AEQ = 0
Q5’	NDVI IN [6000 6500] & VIU = 4 & VIQ = 2 & LWM = 2 & AEQ = 0

Figure 4. The set of five-dimensional vegetation queries.

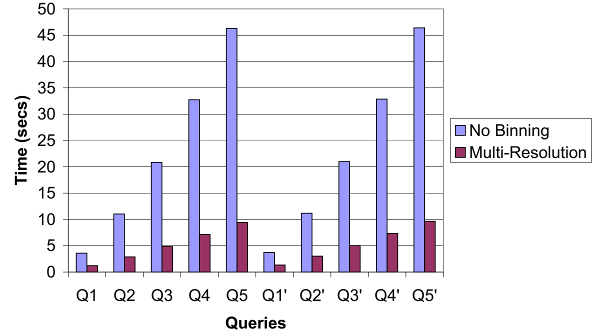
performance improvement.

The total sizes of the indexes that were read for the queries are given in Figure 5(c). Despite the reduced performance improvement compared to the theoretical maximum, the performance improvement from using a multilevel index is still substantial. This implies that an approach of combining indexes at different levels of binning is likely to lead to improved performances of the bitmap indexes for range queries over high cardinality attributes.

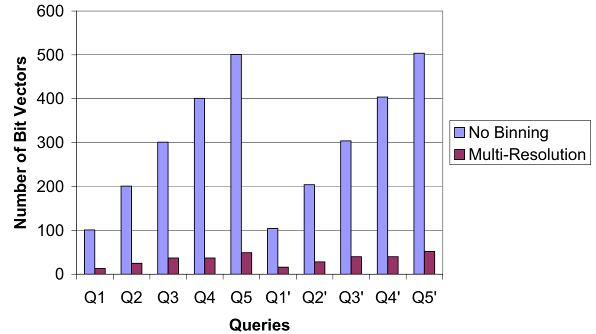
To show the effectiveness of bitmap indexes with higher-dimensional queries, we also ran a set of five-dimensional queries suggested by domain scientists. This set of queries is listed in Figure 4, and their running times are shown in Figure 5(a). These queries again show that the running times of the queries are dominated by the cost of evaluating the range condition. As the range increases, the time needed to answer the queries also increases dramatically. Again the Multi-resolution Index provides a much better running time, due to the reduced number of Boolean index operations it requires. The running times for the queries are almost the same as the running time for the queries with the corresponding ranges in the two-dimensional queries. This indicates that the efficiency of the indexes in a multi-dimensional environment hinges on the number of bit vectors involved at run time, and is not sensitive to the number of query dimensions per se. In other words, the limiting factor for bitmap index performance is not the number of attributes with query conditions, but rather the ranges of the individual attributes in the query conditions.

4.2 Rocket Simulation Domain

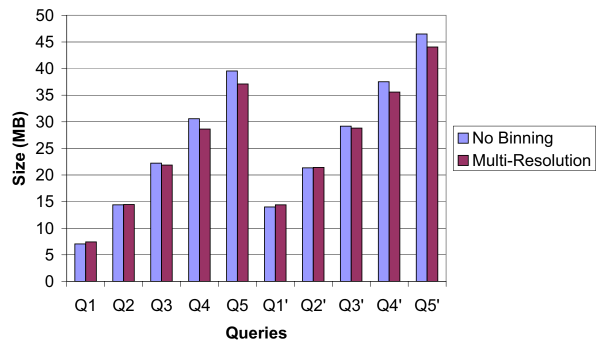
These experiments employed an 11.1 GB data set generated by a medium complexity simulation at CSAR. The data set had 11 attributes, with 10 of type double and one of type integer. Each attribute was stored in 7,599 separate HDF datasets (a 3D array of 12 * 12 * 28) points, with one dataset for each attribute in each of the 7,599 files. The amount of data for each attribute is approximately 220 MB, and a se-



(a) Time Taken



(b) Number of Bit Vectors read



(c) Size of Bit Vectors read

Figure 5. Results for Vegetation Index Dataset.

quential scan over any one of the attributes takes approximately 1,003 seconds, owing to the need to open and close 7,599 different files for each attribute.

The sequential versions of these experiments were run on the same platform as for the vegetation experiments. The parallel versions of the rocket experiments were conducted on the Computational Science and Engineering Turing cluster at UIUC, which has 640 Apple dual processor machines running at 2GHz, a Myrinet interconnect for MPI, and a 100 Mbps Ethernet for other network traffic, including the filesystem traffic. The Turing cluster has a single Apple XRaid storage server with 7 TB of storage space, connected to a Linux NFS server over a 2 Gbps Fibre Channel Interface. The Linux server exports the volume over NFS to the rest of the cluster, using a 100 Mbps Ethernet link. These experiments used the parallel version of the Maitri Query Manager, creating the indexes in parallel and manipulating them in parallel as well.

The queries we execute on this database were suggested by scientists, and they impose conditions on the speed of sound (af) and temperature (Tf) attributes. (The speed of sound varies for the gas in the rocket, according to the temperature and pressure.) Both these attributes are of type double; af ranges from 0 to 2,000, and Tf ranges from -250 to 24,750. Since these are both doubles, we need to bin the data. Since the CSAR visualization tool has a query interface that restricts most queries to integral ranges, we chose to make the smallest bin an integer. Thus Tf has 25,000 distinct bins, while af has 2,000 bins. Both of these are high cardinality attributes, so to give good performance we once again use a multi-resolution index. For the Tf attribute, each bit vector in the lower resolution index corresponds to a bin of 25 different values, and in af every bit vector corresponds to 10 different values. Thus the number of bit vectors for the lower resolution index is 1,000 for Tf and 200 for af. The total size of the high resolution bitmap index is 174 MB for Tf and 121 MB for af. The sizes of the low resolution bitmap indexes for Tf and af are 53 and 42 MB respectively.

We ran ten queries, five of which are single dimensional queries of increasing range. These queries demonstrate that as the range size increases, more and more bit vectors must be read and operated upon, causing considerable slow down. The remaining five queries are two-dimensional, and they show that the performance penalty of introducing an additional dimension is very low as long as the range restriction within that dimension is narrow. The queries are listed in Figures 6(a) and 6(b).

The performance results for the one-dimensional queries on a sequential machine are shown in Figure 7(a). We see that with increasing range of the query, the performance of the query degrades considerably. As in the vegetation domain, we see that the multi-resolution index outperforms the single-resolution index, because of the reduction in the number of bit vectors that must be read to answer the query. The total size of the bit vectors read for each query is given in Figure 7(c), and the total number of bit vectors read for each query is summarized in Figure 7(b).

The two-dimensional query index operation times on a sequential machine are summarized in Figure 7(a). Here we see that the response time for a two dimensional query is much higher than that for a single dimension query. This is not due to the introduction of a new dimension per se; it is due to the fact that the second dimension condition is a range condition with a large range. The processing of this range slows down the whole query. This effect is also visible when we run the queries using the multi-resolution index. The multi-resolution index greatly improves the query performance by reducing the time needed to evaluate the range query in the second dimension. In this case, the performance of the multi-resolution index for the two-dimensional query is almost as good as the performance for the one-dimensional query.

Parallel Index Creation and Querying. Many scientists choose to write parallel scientific applications, so that they can simulate larger phenomena or process larger amounts of data in a reasonable amount of time. Thus an index manager for scientific data should be parallelizable, in the sense that the performance of creating and querying the index should scale well as the number of nodes in the system increases. Our preliminary experiments with a parallel query processor and index creator show that it is very easy to parallelize a bitmap index.

The CSAR rocket simulation codes are parallel, and CSAR also has parallel visualization tools. To evaluate query performance in a parallel environment, we also ran experiments on the CSE Turing cluster described earlier. These experiments used the parallel version of the Maitri Query Manager, creating the indexes in parallel and manipulating them in parallel as well.

In our implementation of parallel index creation, as each node reads a subset of the data, it creates a bitmap index for that subset. When the data is read in later on, the index provides efficient access to that subset of the data. Performance results from experiments on a small 228 MB dataset from the CSAR rocket simulations are presented in Figure 8(a), while results from a much larger 11 GB dataset are shown in Figure 8(b).

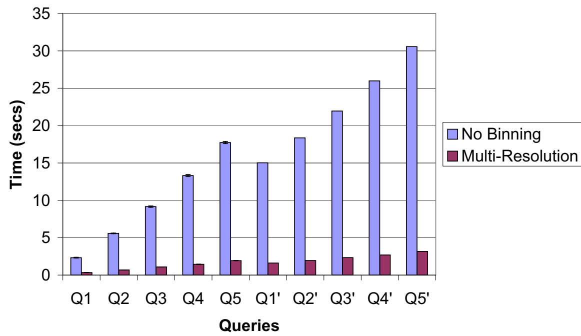
Query Name	Query Condition
Q1	af IN [500 600]
Q2	af IN [500 700]
Q3	af IN [500 800]
Q4	af IN [500 900]
Q5	af IN [500 1000]

(a) 1 Dimensional Queries

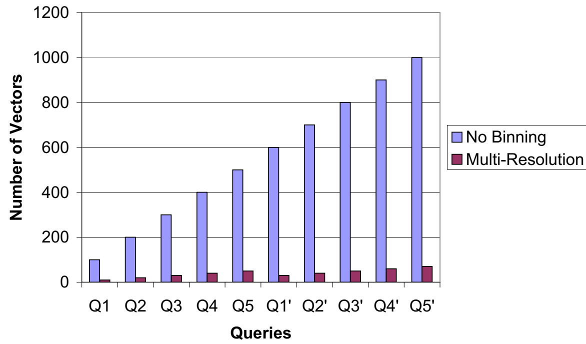
Query Name	Query Condition
Q1’	af IN [500 600] & Tf IN [1000 1500]
Q2’	af IN [500 700] & Tf IN [1000 1500]
Q3’	af IN [500 800] & Tf IN [1000 1500]
Q4’	af IN [500 900] & Tf IN [1000 1500]
Q5’	af IN [500 1000] & Tf IN [1000 1500]

(b) 2 Dimensional Queries

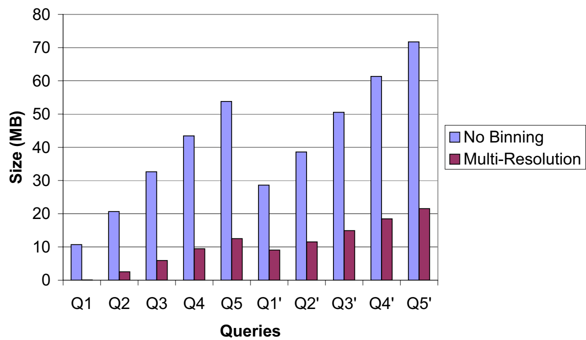
Figure 6. Queries Used for the Vegetation Dataset.



(a) Time Taken

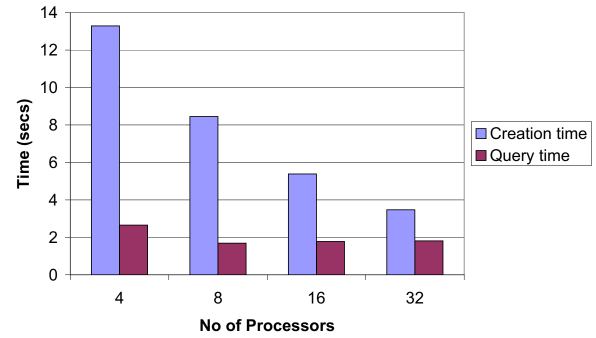


(b) Number of Bit Vectors Read

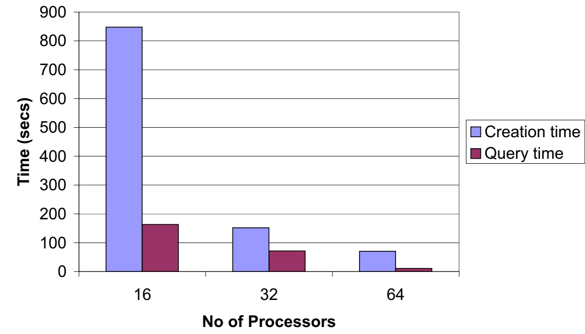


(c) Size of Bit Vectors Read

Figure 7. Results from CSAR Dataset.



(a) 228 MB Dataset



(b) 11 GB Dataset

Figure 8. Parallel Creation and Query Results.

No of Processors	Without Index	With Index
1	195	142
4	45	41
8	22	21

Figure 9. Time taken (in secs) for two-dimensional queries (CSAR dataset) used for parallel runs.

The query condition executed was Tf IN [3500 3900], in all experiments. The index is created at the block level, i.e., each object that is being indexed is not an individual database object, but rather a collection of lower-level database objects. In Figure 8(a) we see that as the number of processors increases, performance at first improves, but then it degrades after 4 processors. This is because after this stage the contention for reading the startup metadata becomes the dominating factor in performance, rather than the actual reading of the data. Figure 8(b) shows results for a much larger data set; here we see that with the increase in the number of processors, the query performance improves tremendously. On a closer look we see that the query performance is increasing super-linearly. This is because in the 16 processor case the load is not propoerly balanced, it so happens that the files to 2 of the processors are much larger in size and hence these two processors end up being a bottleneck. A proper I/O load balancer would take care of this problem.

Indexing Support for Voyager. Voyager is the batch mode version of Rocketeer, a visualization tool developed at CSAR. Voyager processes a series of snapshots from a simulation, in parallel. We integrated our index manager with Rocketeer and ran experiments to see how much improvement we could achieve with indexing support in a parallel real world application. We visualized the temperature of the rocket for a 300 MB data set. The time taken for the visualizations is recorded in Table 9. We see that as the number of processors increases, the improvement afforded by the index goes down. This is because once we have more than four processors, the initialization of the visualization widgets and the metadata manipulation start to dominate the run time of the visualization tool. For a considerably larger data set where the actual reads are the bottleneck (as happens here in the single-processor case), the indexing offers considerable benefit over the non-indexed case.

5 Conclusion and Future Work

We have presented one of the first detailed case studies of the use of bitmap indexes for large scientific data sets. Our sequential and parallel experiments with vegetation data and rocket simulation data showed that bitmap indexes can greatly improve execution time, but only if the index structures are enhanced in a new way. In particular, a hierarchy of multi-resolution indexes is needed in order to limit the number of bit vectors that must be read during query processing. We also found that bitmap indexes can be created and operated upon in parallel very efficiently, with good scale-up in performance as the number of processors increases. We conclude that the purveyors of scientific I/O libraries should consider including bitmap indexes in future releases of those libraries.

6 Acknowledgments

The research is supported by the Department of Energy under subcontracts B341494, DOE DEFC02-01ER25508, and by NSF grant NSF ACI 02-05611. We would also like to thank John Norris for his help with the Voyager visualization toolkit.

References

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

[2] C. Bachman. The programmer as navigator. *Comm. ACM*, 1973.

[3] C. Chan and Y. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD*, 1999.

[4] A. Choudhary, M. Kandemir, and H. N. et al. Data management for large-scale scientific computations in high performance distributed systems. In *High Performance Distributed Computing*, 1999.

[5] P. Kumar, P. Bajcsy, and D. T. et. al. Using D2K data mining platform for understanding the dynamic evolution of land-surface variables. In *Earth-Sun System Technology Conference*, 2005.

[6] J. No, R. Thakur, and A. Chaudhary. Integrating parallel file I/O and database support for high-performance scientific data management. In *Proceedings of SC2000: HPNC*, 2000.

[7] P. E. O'Neil. Model 204 architecture and performance. In *HPTS*, pages 40–59, 1987.

[8] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, 1997.

[9] B. Plale, J. Alameda, and B. W. et. al. Active management of scientific data. In *IEEE Internet Computing*, 2005.

[10] K. Stockinger. Design and implementation of bitmap indices for scientific data. In *IDEAS*, 2001.

[11] K. Stockinger, K. Wu, and A. Shoshani. Evaluation strategies for bitmap indices with binning. In *DEXA*, 2004.

[12] K. Wu, E. J. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB*, 2004.

[13] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical report, Lawrence Berkley National Laboratory, 2001.