Strategies for Processing ad hoc Queries on Large Data Warehouses*

Kurt Stockinger
CERN
Geneva, Switzerland
Kurt.Stockinger@cern.ch

Kesheng Wu
Lawrence Berkeley Nat'l Lab
Berkeley, CA, USA
KWu@lbl.gov

Arie Shoshani
Lawrence Berkeley Nat'l Lab
Berkeley, CA, USA
Shoshani@lbl.gov

ABSTRACT

As data warehousing applications grow in size, existing data organizations and access strategies, such as relational tables and B-tree indexes, are becoming increasingly ineffective. The two primary reasons for this are that these datasets involve many attributes and the queries on the data usually involve conditions on small subsets of the attributes. Two strategies are known to address these difficulties well, namely vertical partitioning and bitmap indexes. In this paper, we summarize our experience of implementing a number of bitmap index schemes on vertically partitioned data tables. One important observation is that simply scanning the vertically partitioned data tables is often more efficient than using B-tree based indexes to answer ad hoc range queries on static datasets. For these range queries, compressed bitmap indexes are in most cases more efficient than scanning vertically partitioned tables. We evaluate the performance of two different compression schemes for bitmap indexes stored is various ways. Using the compression scheme called Word-Aligned Hybrid Code (WAH) to store the bitmaps in plain files shows the best overall performance for bitmap indexes. Tests indicate that our bitmap index strategy based on WAH is not only efficient for attributes of low cardinality, say, < 100, but also for high-cardinality attributes with 200,000 or more distinct values.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database managementSystem[Query processing]; H.2.2 [**Information Systems**]: Database managementSystem[Access methods]

General Terms

Algorithms, Performance

1. INTRODUCTION

Many large scale science projects are generating or planning to generate terabytes of data per year. For example, a high-energy physics project called STAR is currently online and is capable of generating a terabyte a year[1]. Emerging scientific applications such as the Large Hadron Collider (LHC)[2] at CERN and the NASA EOS[3] satellites are capable of producing data even faster. In the commercial world, similar amounts of data are being gathered. On-line analytical processing (OLAP) on these datasets presents a great challenge to most software systems built around them.

Currently, the predominant model of data processing is to store data as tables in relational database management systems (DBMS) and use a variant of the B-tree index [8] to facilitate the searching operations. This basic strategy is inadequate for ad hoc queries on large datasets. The B-tree index is effective if the user queries involve only one attribute or always involve similar conditions on the same set of attributes. However, many OLAP queries are ad hoc in nature; they involve different subsets of attributes at different time. There are a number of multidimensional indexing schemes that are designed to speed up the processing of multidimensional queries [9]. However, they usually suffer from the curse of dimensionality, i.e., as the number of dimensions (the number of attributes in the dataset) increases, either the size of the index grows superlinearly or the search time grows superlinearly. In many instances, both the size and the time grow exponentially as the number of dimensions increases. These indexing schemes are generally considered more efficient than sequentially scanning a relational table when the number of dimensions are less than ten and each query involves all the attributes. On queries involving only a subset of the attributes or for datasets with higher dimensions, using these indexing schemes takes more time than performing sequential scan.

Since simply scanning the data values is one of the most efficient strategies to answer complex queries, the perfor-

*This work was supported by the Director, Office of Science, Office of Laboratory Policy and Infrastructure Management, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DOLAP'02, November 4–9, 2002, McLean, Virginia, USA.
Copyright 2002 ACM 1-58113-492-4/02/0011 ...$5.00.

[1]More information about the STAR project is available at `http://www.star.bnl.gov/STAR`.
[2]More information about the LHC project is available at `http://lhc.cern.ch`.
[3]More information about EOS is available at `http://eos.nasa.gov`.

mance of the these scanning operations is important for data warehousing applications. Typically the raw data changes infrequently in these applications, thus it is appropriate to partition the data vertically, i.e., keeping the values of each attribute close together. Some authors have suggested projecting out commonly queried attributes and store the values of each attribute separately. This is called the projection index [14]. Instead of doing this for some attributes, we believe it is better to store all attribute values this way.

Other indexing strategies that are often considered effective for processing ad hoc queries on large data warehouses include the bitmap index [11, 13] and the UB-tree [4, 12]. In this paper, we report our experiences from implementing various versions of the bitmap index. Some of the lessons learned include that (i) vertical partitioning is useful, (ii) the conventional wisdom about IO cost dominating total query processing time is incorrect for compressed bitmaps, (iii) compressed bitmap indexes for high cardinality attributes are usually smaller than B-tree indexes, and (iv) more importantly, these compressed bitmap indexes are quite efficient.

We have also studied two strategies for storing bitmap indexes persistently, using plain files or using an ODBMS system. Compared to a bitmap indexing scheme implemented in a commercial relational DBMS, the one based on plain files shows similar performance when using the simple indexing method of the commercial system. Our simple bitmap index implementation based on the ODBMS is slower, as expected. However, since the file-based program implements vertical partitioning and utilizes better bitmap indexing schemes, it significantly outperforms the relational DBMS in processing range queries.

2. RELATED WORK

2.1 Bitmap indexes

Bitmap indexes were first implemented in a commercial DBMS called Model 204 [13] although they were already used in the 60s. This index data structure is mostly used for On-Line Analytical Processing (OLAP) and data warehouse applications [7], which are mainly characterized by complex queries in read-only or append-only environments. Bitmap indexes offer the best query processing performance for such environments. They are not optimized for typical transaction operations such as insert, delete or update.

A detailed discussion on designing bitmap indexes based on different encoding schemes is presented by Chan and Ioannidis [5, 6]. Equality encoding, which is also referred to as the *basic Bitmap Index*, can be regarded as the most fundamental method that consists of $|A|$ bitmaps where $|A|$ is the cardinality of the attribute to be indexed. This type of index is optimal for *exact match queries. One-sided range queries* show the best performance characteristics with *range encoded bitmap indexes*, which only consist of $|A|-1$ bitmaps. The third type of encoding called *interval encoding* generates $|A|/2$ bitmaps only and is optimal for *two-sided range queries*.

All bitmap encoding techniques mentioned so far are optimized for typical commercial applications with categorical values. Bitmap indexes are used in Oracle, Sybase and Informix as an alternative to the more conventional tree-based data structures for handling multidimensional queries on low cardinality attributes, say cardinality < 100. There are many datasets that contain attributes with high cardinality. For example, a typical scientific dataset usually contains many attributes with cardinalities much larger than 100. On these datasets, the bitmap index is not effective either because the index size is too large or the query processing time is too long to be practical.

There are a number of approaches to reduce the index size and increase the performance of the bitmap index for high cardinality attributes. These approaches include multicomponent encoding [5, 6], binning the attribute values [15, 16] and compressing the bitmaps. This paper concentrates on our the effects of compression.

2.2 Bitmap compression

Compression is one of the ways to make the bitmap index useful for high cardinality attributes. By compressing the bitmaps, less disk space is required to store the indexes and thus they can be read faster from disk into memory. With smaller indexes, more of them can be kept in the memory cache and boolean operations between compressed bitmaps might be faster as well. An efficient bitmap compression scheme not only has to reduce the size of bitmaps but also has to perform bitwise boolean operations efficiently.

A number of the well known compression schemes were studied by Johnson and colleagues [1, 10]. Among them, the scheme named the Byte-aligned Bitmap Code (BBC) [2, 3] shows the best overall performance characteristics. Recently, we have derived another scheme that can greatly reduce the overall query processing time compared to BBC. This scheme is named the Word-Aligned Hybrid (WAH) scheme [18]. It is significant faster because it always operates on words rather than bytes or bits and because it requires very little work to encode and decode the compressed bitmaps. We reported that WAH can be an order of magnitude faster than BBC in performing boolean operations [18]. In this paper, we support this claim by comparing WAH against implementations of BBC from different sources. We also identify how IO operations affect the relative performance of the two compression schemes.

2.3 Why build new bitmap index software

Many commercial database systems have implemented some versions of the bitmap index. For example, Oracle has implemented a compressed bitmap index with equality encoding [3], Sybase IQ provides the bit-sliced index [14], and IBM DB2 UDB implements the Encoded Vector Index[4]. All of them are targeted for typical commercial data warehouses where most of the attributes have relatively low cardinalities. They are not appropriate for scientific data with mostly high cardinality attributes.

The scientific datasets are also larger than those in a typical commercial data warehouse and most of the users are likely to be experts that issue complex ad hoc queries. These reasons lead us to develop our own bitmap index software [15, 16]. One fundamental issue to be addressed in this software is how to store the bitmaps. Since it is unlikely that we can develop a full-featured DBMS ourselves, two obvious options are to store the bitmaps in plain files or to store them as objects in an object-oriented DBMS (ODBMS). We explore both of these options.

[4]Some information about the Encoded Vector Index is available at `http://www-919.ibm.com/developer/bi/evi.html`.

3. VERTICAL PARTITIONING

Most DBMS organize their data as relational tables or objects, where the values of each record are grouped next to each other. Imagine the entries of the table are laid out on a plane; this organization puts rows of the table together and places a number of rows in a page. We call this horizontal partitioning of the table. An obvious alternative is to partition the table vertically and group the values of each column (attribute) together.

Table size	Time [sec]
one-attribute table	0.51
12-attribute table	5.8

Figure 1: Time used by a commercial DBMS to perform sequential scan on tables with 2.2 million records.

Clearly, if most of the queries are scanning only a small number of attributes, vertical partitioning reduces the number of pages accessed. Figure 1 contains a set of timing results on a set of sample data. As expected, scanning the 12-attribute table takes nearly 12 times as long as scanning an one-attribute table. Clearly, if only one attribute is to be scanned, scanning it separately is much more efficient than scanning the whole table. The main drawback of this strategy is that updating the table is more time-consuming. However, since most updates to the data warehouses are performed in bulk and usually they are only appended, this deficiency is not significant.

To make insertion of new records more efficient, often 20% or even 30% of the bytes in a page are left unused in a typical relational table. This also increases the number of pages accessed during the scanning operation. If a data warehouse only appends new data in bulk, it is also appropriate to densely pack the pages of the files containing the data. Densely packing the pages also makes it easier to directly access an individual data item. This makes masked scan operations, also known as sequential skip scans, more efficient. Figure 2 shows the timing results of using masked scan to answer multidimensional range queries of the form, $(3 \leq A < 5) \& (1 < B \leq 10)$, for example.

The test dataset consists of 2.2 million records of STAR[1] data. The most commonly queried 12 attributes are included here. The dataset contains a total of 500 attributes. The query box is the hypercube formed by the boundaries of the query conditions [12]. Its size is measured as a ratio of the hypercube volume and the total volume of the domain of all attributes. For example, let the values of `Energy` be integers in the range of 0 to 30 and `NumParticles` in the range of 1 to 15, the query box size of "`Energy > 15 GeV and 7 <= NumParticles < 13`" is $15/31 \times 6/15 = 0.19$. In our tests, we limit the query box size to be less than 0.1 because it is unlikely that a user would routinely process more than 10% of a large dataset with terabytes of data.

For many queries, say, those with query box size < 0.01, only one attribute needs to be fully scanned and the masked scan operations on the remaining attributes only touch a small number of entries. This makes the total execution time usually quite close to the time of scanning one attribute. From Figure 2, we see that scanning the vertically partitioned table often takes about 0.5 seconds. In contrast

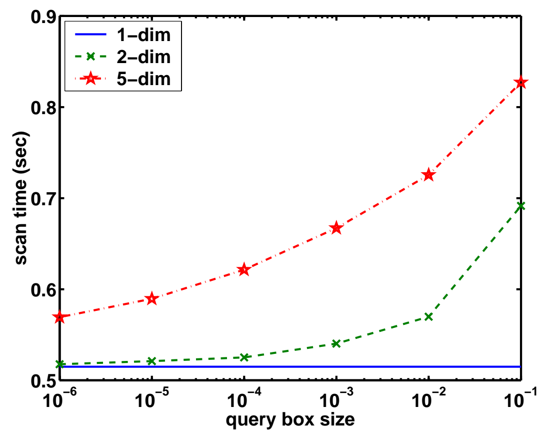

Figure 2: Average time needed to process multidimensional queries using masked scan on a table with 2.2 million records.

if the same queries are answered using a B-tree index, the average time is 2.2 seconds. The time is about the same for two and five dimensional queries because in most cases the DBMS is only able to utilize the B-tree index of one attribute and has to scan the table to resolve the condition on the remaining attributes.

4. COMPRESSION SCHEMES

In previous work [10, 18, 19], a number of good bitmap compression schemes have been identified. Here we select two of them, WAH and BBC, for further testing. They are selected because WAH is usually the fastest and BBC compresses well and is faster than most others. Our tests use two different BBC implementations, namely, *BBC-s* which is a simplified version of the 2-sided BBC, and *BBC-f* which is a 2-sided version of BBC based on the implementation by Theodore Johnson [10]. BBC-f is the version that was used by Stockinger [17] and BBC-s was used by Wu et al. [18].

The set of bitmaps used in the tests reported in this section consists of 20 random bitmaps with 10 different bit densities d ranging from 10^{-4} to 0.5. For instance, a bit density of 10^{-4} means that on average one bit out of 10,000 is set to 1. All other bits are set to 0. Since WAH, BBC-s and BBC-f are symmetric compression algorithms, bitmaps with bit densities above 0.5 show the same results as those with densities of $1 - d$.

Each uncompressed bitmap consists of 10^8 bits and is stored as a single Unix file. The total file sizes of the uncompressed bitmaps and compressed bitmaps using BBC and WAH are listed in Figure 3. The corresponding sizes of the individual bitmaps are displayed in Figure 4. Overall, BBC-s compresses nearly as well as BBC-f and WAH uses about 50% more space than the two versions of BBC.

4.1 Computational Complexity

In order to measure only the computational complexity of the bitmap compression schemes, we loaded the compressed bitmaps into main memory. The tests are executed on three different machines (called *tin*, *dms* and *dm*) with various CPU speeds and I/O subsystems, see Figure 5.

We group the set of 20 bitmaps into 10 pairs according

Compression schema	Size [MB]
uncompressed	250
WAH	139
BBC-s	94
BBC-f	90

Figure 3: Total sizes of the test bitmaps (20 x 10^8 bits).

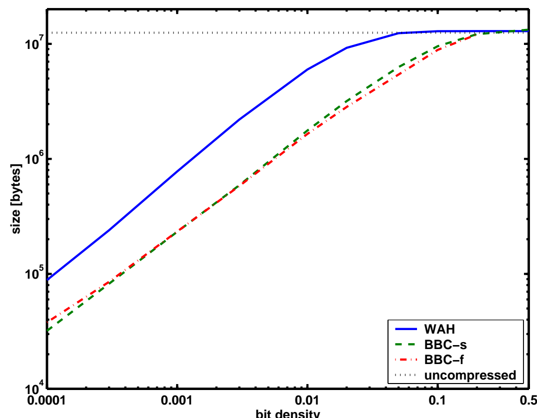

Figure 4: Compressed sizes (bytes) of 10^8 bits of different bit densities.

to their bit densities. We then measure the time required to perform a bitwise logical AND operation after the two bitmaps are loaded into memory. Figure 6 shows the measured time on the machine called *dm*. Summary information about the in-core performance of the three different compression schemes is displayed in Figure 7. These timing results show that WAH is more efficient in performing boolean operations than BBC. In some cases WAH can be an order of magnitude faster than BBC.

4.2 Reading plain files

When answering a query, the most time consuming part is to read the bitmaps from disk and perform bitwise boolean operations on them. In the next set of tests, we emulate this process by first reading two files containing two compressed bitmaps and then perform a single logical operation on the two bitmaps. Generally, the BBC files are smaller than WAH files. If the disks are slow, the total time for BBC is lower; and if the disks are faster, the total time for WAH is smaller. Figure 8 shows the total time required on two different machines called *dm* and *tin*. On dm, where the disk is relatively fast, the total time for WAH are generally smaller than the two versions of BBC. When the disk is slow, as on tin, the total time for WAH is often larger than that of BBC. This is especially true for bitmaps with very low bit densities. In these cases, BBC-f uses about half the time required by WAH because the corresponding files for BBC-f are about half of those for WAH.

The measurements recorded in Figure 8 are obtained with *cold files*. In this case, the disk has just been unmounted and mounted again. The OS has no information about the specific directory containing the files or any information about the files. Had we accessed the files before, the OS would

Name	Disk MB/s	CPU type	MHz	OS
tin	2	Pentium 3	500	Linux 2.2.12
dms	5	Pentium II	300	Solaris 5.8
dm	20	UltraSPARC	450	Solaris 5.8

Figure 5: Information about the test machines.

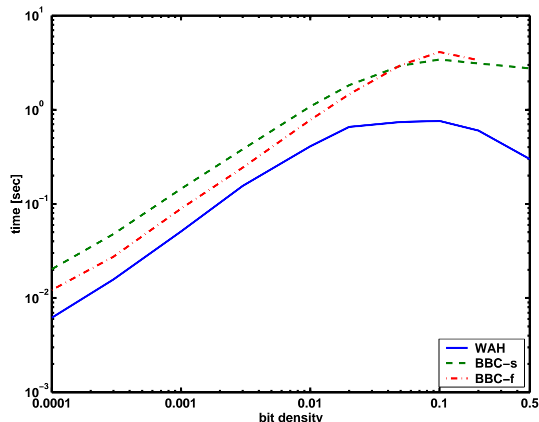

Figure 6: Time (seconds) to perform bitwise AND on dm. The compressed bitmaps are in memory.

have cached directory information about the files and the IO time maybe significantly reduced. In the following discussions we refer to these files as *warm files*. Since database systems also cache indexes, the total time reported in Figure 8 is weighted more heavily with IO time than in a typical database system.

The conventional wisdom regarding index size and query processing time is that IO time dominates the overall processing time, thus the smaller the index size the less the query processing time. For compressed bitmap indexes, this observation appears to be wrong. Figure 9 shows the ratio of IO time and the total execution time. Even on a relatively slow disk, the BBC schemes spend not much more than half of the total time in reading the files from disks. On both machines called *dms* and *dm* where the disk speeds are higher, less than half of the total time is spent on reading the files. In selecting a compression scheme for bitmap indexes, we clearly have to take into account both the CPU efficiency as well as the IO efficiency.

4.3 Worst case sizes

In earlier tests [18, 19], we have observed that the boolean operation time is nearly proportional to the sizes of the compressed operands. This leads us to examine what is the expected size of the bitmaps in a bitmap index. Given the same attribute cardinality, the bitmap index for the one with uniform distribution requires the largest amount of space. Figure 10 shows these worst case sizes of attributes with different cardinalities.

As shown in Figure 10, the index size increases as the cardinality increases. Since there are a total of 10^8 records, the maximum attribute cardinality is 10^8. In this extreme case, the size of the bitmap index is 4×10^8 words. If each

	average time			ratio of average time			average of ratios		
	WAH	BBC-s	BBC-f	$\frac{\text{BBC-s}}{\text{WAH}}$	$\frac{\text{BBC-f}}{\text{WAH}}$	$\frac{\text{BBC-f}}{\text{BBC-s}}$	$\frac{\text{BBC-s}}{\text{WAH}}$	$\frac{\text{BBC-f}}{\text{WAH}}$	$\frac{\text{BBC-f}}{\text{BBC-s}}$
tin	0.36	1.36	2.44	3.8	6.8	1.8	4.2	6.1	1.3
dms	0.50	1.85	2.80	3.7	5.6	1.5	4.0	4.7	1.1
dm	0.37	1.57	1.64	4.2	4.4	1.0	4.0	3.7	0.85

Figure 7: Summary information about the in-core performance of the different compression schemes.

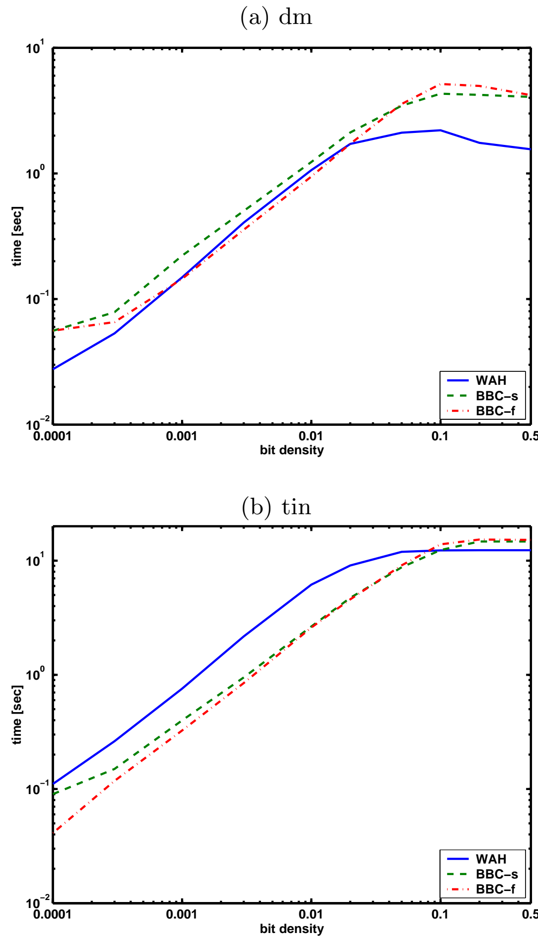

Figure 8: Total time (seconds) needed to read two files and perform one bitwise AND operation.

attribute value can be stored in one word, the index size is about four times the data size. A B-tree index for the same attribute typically takes 3–4 times the data size, this maximum bitmap index size is acceptable. For a large range of cardinality values, the compressed index size is about twice the data size which is smaller than the corresponding B-tree index. In practice, most attributes have non-uniform distribution and the size of their compressed bitmap indexes would be even smaller.

5. HOW TO STORE BITMAPS

In terms of how to store the bitmaps generated for an index, there are a number of options. This section discusses our experiences using two such options, files and an ODBMS. This section is divided into two subsections, one for the file based program and one for the program based on ODBMS. The goal is to identify a reasonable strategy for further testing of different indexing schemes.

5.1 File based bitmap index

This file based program was originally developed for a high-energy physics application [15]. It manages its own file accesses with locks and tracks its memory usage with a trivial memory management scheme. For portability reasons, we have selected standard Unix read and write functions for IO operations which may be slower than lower level functions that directly interact with disk controllers. Since a typical DBMS implements more sophisticated memory management scheme and uses low level IO functions, we are interested in finding whether our file based program have reasonable performance. This file based program is labeled as IBIS in the following discussions.

In the next set of tests, we measure the time required to process queries. We expect these time to be dominated by the time to read bitmaps and to perform logical operations. To verify this, we constructed a pair of special attributes each with cardinality of 11. Among the 11 values, 10 of them follow a distribution so that the basic bitmap indexes consist of the same bitmaps as used in the previous tests, i.e. the bitmaps have bit density ranging from 10^{-4} to 0.5. A set of queries are constructed so that one bitwise boolean AND operation between two bitmaps with equal densities are required. The queries are given to two versions of IBIS and the DBMS. For ease of comparison, we plotted the timing results using the bit density as the x-axis, see Figure 11 for the measurements on *dm*. As expected, the time require to process these special queries are about the same as the time required to read the bitmaps and perform logical operations, see Figure 8(a) and Figure 11.

Figures 12 show the sizes of the bitmap index, the creation time and the average query processing time on *dm*. Because the commercial DBMS implements a BBC compressed bitmap index, the size of its index is close to the BBC-s compressed index generated by IBIS. The average query processing time of the DBMS and IBIS with BBC-s are also about the same. This indicates that our file based program has reasonable performance. The index creation time of IBIS is much better than that of the DBMS. More work is required to fully understand its causes.

5.2 Bitmap index on top of an ODBMS

The second program named BMI stores its bitmaps in an object-oriented database management system (ODBMS) [17] and was also initially motivated by a high-energy application [16]. As far as we know, this program is the first bitmap implementation on top of an ODBMS. We selected

[5]The DBMS actually allocated 247 MB for the bitmaps.

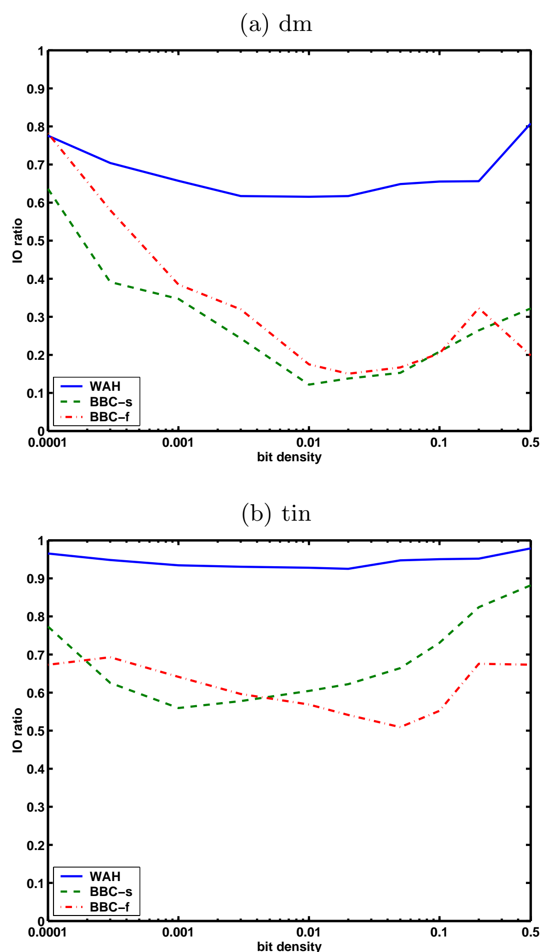

Figure 9: The ratio of IO time and the total time reported in Figure 8.

to use an ODBMS because it is easier to manage bitmaps as objects in an ODBMS than treating them as tables or opaque objects in a relational DBMS.

In this program, each bitmap is implemented with a general purpose persistent scalable variable array which is stored in segments of 8 KB. Bitmaps can be accessed via so-called smart pointers but also via standard iterators. Due to this design decision, it is fairly simple to plug in various compression algorithms.

For the next set of tests, we have decided to used is a set of 10 random attributes with exponential distribution since this kind of distribution is fairly common for scientific data [16]. Figure 13 shows the query processing time on the machine called *tin*. The test data used consists of 5 million records. In the tests, each query involves five attributes selected from a total of 10 attributes. After the attributes are selected, a common boundary b is chosen to form the test query $(a0 > b)\&(a1 > b)\&(a2 > b)\&(a3 > b)\&(a4 > b)$. The query boundary b is used as the horizontal axes in Figure 13 which shows the time required to process this type of query by BMI with BBC-f and WAH, and IBIS with WAH. In all tests, BMI with WAH performs better than BMI with BBC-f and IBIS with WAH is the fastest overall.

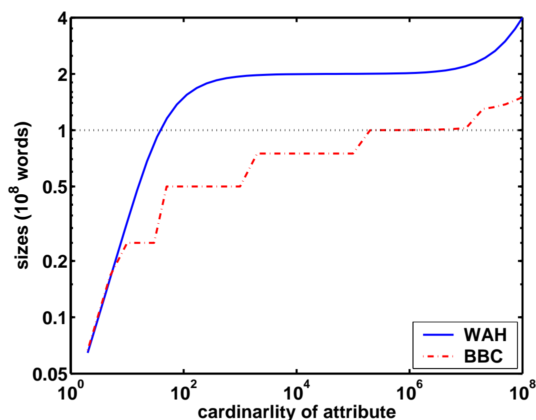

Figure 10: Compressed sizes of the basic bitmap index for an attribute with uniform distribution (10^8 records).

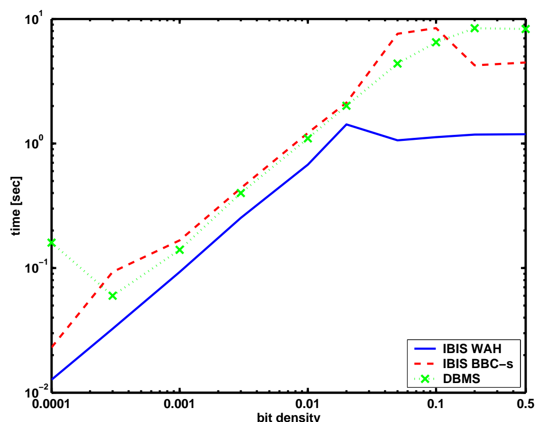

Figure 11: Time (seconds) required to perform bitwise logical operations through IBIS and a commercial DBMS.

Figure 14 shows the average query processing time. The data indicates that switching from BBC-f compression to WAH compression improves the overall performance of BMI by a factor of three. In a number of cases, BMI with WAH is more than an order of magnitude faster than with BBC-f.

When the files are cold, BMI with WAH compression is almost a factor of five slower than IBIS with the same compression scheme. This suggests a significant overhead for using bitmaps in the ODBMS. We were expecting some overhead for building a bitmap index on top of an ODBMS, however we did not expect the overhead to be this significant. A minor point to note here is that since IBIS relies on the OS to perform IO operations, it is able to significantly reduce the IO time when operating on warm files. Since the ODBMS performs its own file caching, it does not benefit from directory information cached by the OS.

6. PERFORMANCE WITH REAL DATA

This section presents some performance results on a set of data taken from the STAR experiment[1]. This dataset

	size(MB)	create(sec)	query(sec)
IBIS WAH	166	91	0.7
IBIS BBC-s	117	116	2.9
DBMS	123^5	2890	3.1

Figure 12: Total sizes (MB) of the bitmap indexes, the time (seconds) needed to create them, and the average query processing time (seconds) (2 attributes, 10^8 records).

contains about 2.2 million records and the tests are conducted on 12 commonly queried attributes. Our goal is to demonstrate that compressed bitmap indexes can perform well even on very high cardinality attributes. For comparison, we also include time results for scanning the vertically partitioned tables and the time results of using both a bitmap index and a B-tree index of the DBMS.

Figure 15 shows the total sizes of the bitmap indexes compared with the size of the B-tree. IBIS is tested with both WAH and BBC-s and the DBMS is tested with both a bitmap index and a B-tree index. The total number of bitmaps generated for the 12 attributes is nearly 2.7 million. The average attribute cardinality is over 222,000. Without compression, the bitmap index size is more than 720 GB. Both BBC and WAH are very effective in reducing the sizes of the bitmap indexes because the majority of the bitmaps are very sparse. It is important to note that the compressed bitmap indexes are significantly smaller than the B-tree indexes. The total size of the files containing the attribute values is about 113 MB. Based on Figure 10, we expect that the total size of WAH compressed indexes to be about twice this value, 226 MB. However, the actual total size is less than 226 MB because the attributes are not uniform random numbers.

Figure 16 shows the average query processing time of three compressed bitmap indexes. The partial range queries are generated by randomly selecting a number of attributes and constructing a query with the specified query box size [12]. Given a query box size, the shape of the query box is allowed to vary. For simplicity, we only use conjunctive queries; that is the conditions on each attribute are joined together using the AND operator. Typically, as the query box size increases and the number of attributes increases, it takes more time to process the query. The time used by scanning the vertically partitioned table is marked as "p scan" in this figure.

In these tests, BBC compressed indexes require about as much time as scanning the vertically partitioned table, but WAH compressed indexes are always faster. On the average, IBIS with WAH compression is about four times faster than IBIS with BBC-s compression and more than 10 times faster than the bitmap index from the DBMS. It uses 0.01 – 0.2 seconds to answer the test queries, which are significantly less than the 2.2 seconds used by the B-tree indexes of the DBMS.

7. CONCLUSIONS

Motivated by the need to efficiently process ad hoc queries on high dimensionality data, we started to implement a number of different bitmap index strategies. In this paper, we first verified that for ad hoc queries on read-only data, simply scanning the vertical partitioned tables is better than using a variant of the B-tree index in a commercial DBMS.

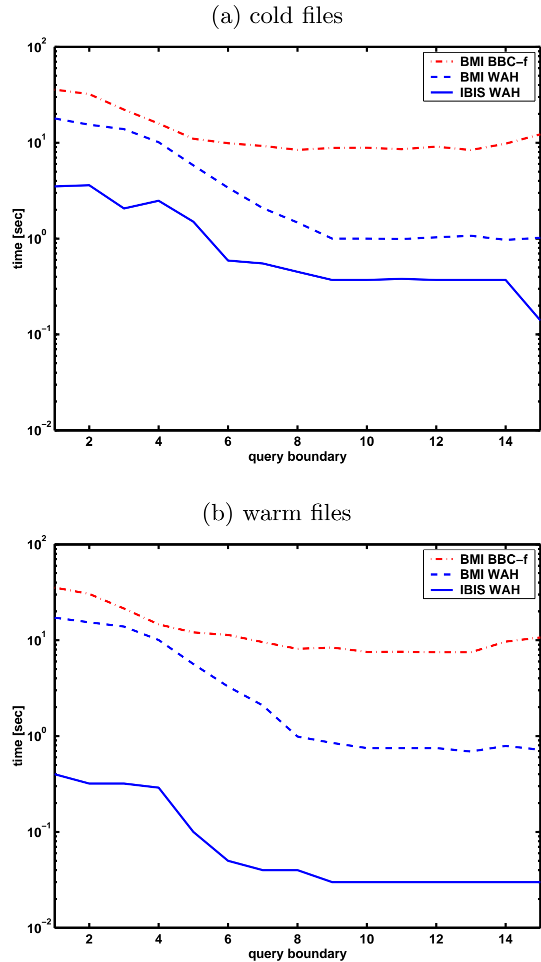

Figure 13: Time (seconds) required to process 5-dimensional queries on the machine *tin* (5 million records with exponential distribution).

Among the various options for bitmap indexes, we mainly examine two aspects in this paper, the compression scheme and the persistent storage method. Regarding the persistent storage method, our observation is that implementing bitmap indexes on top of an ODBMS incurs too much overhead. Even with sophisticated caching scheme, the implementation of the bitmap indexes on files shows good performance compared to a commercial DBMS when the same options are used.

Contrary to the conventional wisdom, smaller compressed bitmap indexes are not necessarily more efficient. Among the various bitmap compression schemes examined, the word-aligned hybrid (WAH) scheme does not generate the smallest indexes, but WAH compressed indexes are always the most efficient in answering queries because it is much more CPU-efficient.

In the worst case the size of a WAH compressed bitmap index is comparable to that of a B-tree index. For most high cardinality attributes, the compressed bitmap indexes are smaller than the corresponding B-tree indexes. This is verified on a set of data from a scientific application where

	cold files	warm files
BMI BBC-f	14.0	13.5
BMI WAH	5.1	4.9
IBIS WAH	1.1	0.12

Figure 14: Average time (seconds) to processing 5-dimensional queries on the machine *tin* (5 million records with exponential distribution).

	IBIS		DBMS	
N	WAH	BBC-s	bitmap	B-tree
2,673,646	186	117	111	408

Figure 15: Sizes (MB) of the bitmap indexes on 2.2 million of STAR data. "N" is the total number of bitmaps in the indexes.

the cardinalities of many of the attributes are more than 222,000. Even on these very high cardinality attributes, WAH compressed bitmap indexes can be orders of magnitudes faster than alternative schemes such as B-tree indexes or scanning the vertically partitioned tables.

8. ACKNOWLEDGMENTS

The authors wish to express our sincere gratitude to Theodore Johnson of AT&T Research for permitting us to use his BBC implementation in our bitmap indexing program.

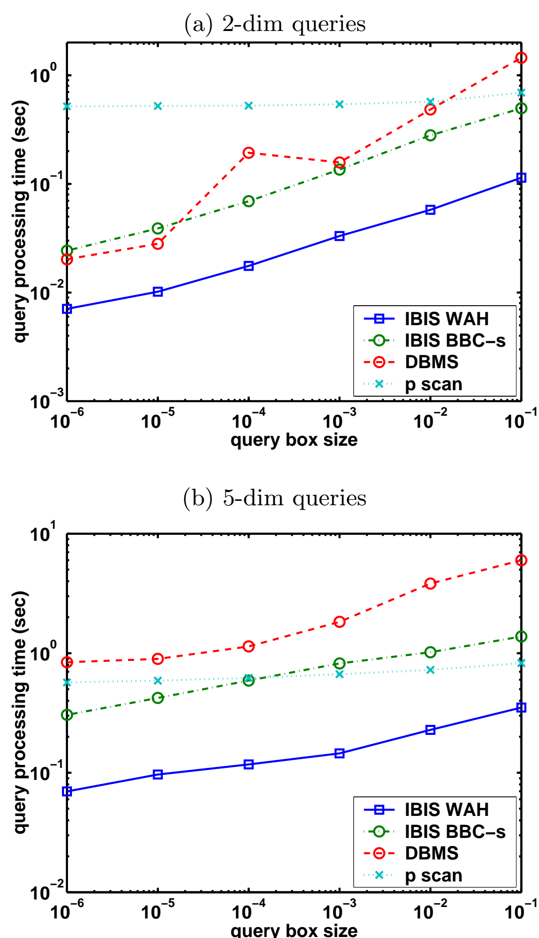

Figure 16: The average query processing time of random range queries on the STAR data.

9. REFERENCES

[1] S. Amer-Yahia and T. Johnson. Optimizing Queries on Compressed Bitmaps. In *Proceedings of VLDB 2000*, pages 329–338. Morgan Kaufmann, 2000.

[2] G. Antoshenkov. Byte-Aligned Bitmap Compression. Technical Report, Oracle Corp., 1994. U.S. Patent number 5,363,098.

[3] G. Antoshenkov and M. Ziauddin. Query Processing and Optimization in ORACLE RDB. *The VLDB Journal*, 5:229–237, 1996.

[4] R. Bayer. The Universal B-tree for Multidimensional Indexing. In *Proc. of Intl. Conf. on World-Wide Computing and Its Applications*, pages 98–112. Springer-Verlag, 1997.

[5] C.-Y. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *Proceedings of SIGMOD 1998*. ACM Press, 1998.

[6] C. Y. Chan and Y. E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries. In *Proceedings of SIGMOD 1999*. ACM Press, 1999.

[7] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record*, 26(1):65–74, March 1997.

[8] D. Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, 1979.

[9] V. Gaede and O. Günther. Multidimension access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[10] T. Johnson. Performance Measurements of Compressed Bitmap Indices. In *Proceedings of VLDB'99*, pages 278–289. Morgan Kaufmann, 1999.

[11] M. Jürgens and H.-J. Lenz. Tree Based Indexes vs. Bitmap Indexes - a Performance Study. *International Journal of Cooperative Information Systems*, 10(3):355–376, 2001.

[12] V. Markl and R. Bayer. Processing Relational OLAP Queries with UB-Trees and Multidimensional Hierarchical Clustering. In *Proceedings of DMDW 2000*, June 5-6, 2000.

[13] P. O'Neil. Model 204 Architecture and Performance. In *2nd International Workshop in High Performance Transaction Systems, Asilomar, CA*, pages 40–59, September 1987.

[14] P. O'Neil and D. Quass. Improved Query Performance With Variant Indices. In *Proceedings of SIGMOD'97*, pages 38–49. ACM Press, 1997.

[15] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional Indexing and Query Coordination for Tertiary Storage Management. In *Proceedings of SSDBM'99*, pages 214–225. IEEE Computer Society Press, 1999.

[16] K. Stockinger, D. Duellmann, W. Hoschek, and E. Schikuta. Improving the Performance of High-Energy Physics Analysis through Bitmap Indices. In *DEXA 2000*. Springer-Verlag 2000.

[17] K. Stockinger. Bitmap Indices for Speeding Up High-Dimensional Data Analysis. In *DEXA 2002*. Springer-Verlag, 2002.

[18] K. Wu, E. J. Otoo, and A. Shoshani. A Performance Comparison of Bitmap Indexes. In *Proceedings of CIKM 2001*, pages 559–561. ACM Press, 2001.

[19] K. Wu, E. J. Otoo, A. Shoshani, and H. Nordberg. Notes on Design and Implementation of Compressed Bit Vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.