LBNL-49627

Compressing Bitmap Indexes for Faster Search Operations

Kesheng Wu, Ekow J. Otoo and Arie Shoshani
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
Email: {kwu, ejotoo, ashoshani}@lbl.gov

Abstract

In this paper, we study the effects of compression on bitmap indexes. The main operations on the bitmaps during query processing are bitwise logical operations such as AND, OR, NOT, etc. Using the general purpose compression schemes, such as gzip, the logical operations on the compressed bitmaps are much slower than on the uncompressed bitmaps. Specialized compression schemes, like the byte-aligned bitmap code (BBC), are usually faster in performing logical operations than the general purpose schemes, but in many cases they are still orders of magnitude slower than the uncompressed scheme. To make the compressed bitmap indexes operate more efficiently, we designed a CPU-friendly *scheme which we refer to as the word-aligned hybrid code (WAH). Tests on both synthetic and real application data show that the new scheme significantly outperforms well-known compression schemes at a modest increase in storage space. Compared to BBC, a scheme well-known for its operational efficiency, WAH performs logical operations about 12 times faster and uses only 60% more space. Compared to the uncompressed scheme, in most test cases WAH is faster while still using less space. We further verified with additional tests that the improvement in logical operation speed translates to similar improvement in query processing speed.*

1. Introduction

This research was originally motivated by the need to manage the volume of data produce by a high-energy experiment called STAR[1] [25, 26]. In this experiment, information about each potentially interesting collision event is recorded and multi-terabyte (10^{12}) of data is generated each year. One important way of accessing the data is to have the data management

OID	**X**	bitmap index =0	=1	=2	=3
1	0	1	0	0	0
2	1	0	1	0	0
3	3	0	0	0	1
4	2	0	0	1	0
5	3	0	0	0	1
6	3	0	0	0	1
7	1	0	1	0	0
8	3	0	0	0	1
		b_1	b_2	b_3	b_4

Figure 1. A sample bitmap index.

system retrieve the events satisfying some condition such as “`Energy > 15 GeV and 7 <= NumParticles < 13`” [5, 25]. The physicists have identified about 500 attributes that are useful for this selection process and a typical condition may involve a handful of attributes. This type of queries are known as the partial range queries. Since the attributes are usually read not modified, the characteristics of the dataset are very similar to those of commercial data warehouses. In data warehouse applications, one of the best known indexing strategies for processing the partial range queries is the bitmap index [6, 8, 21, 30]. For this reason, we have selected to use the bitmap index for the data management software [25].

Generally, a bitmap index consists of a set of bitmaps and queries can be answered using bitwise logical operations on the bitmaps. Figure 1 shows a set of such bitmaps for the attribute **X** of a tiny table (**T**) consisting of only eight tuples (rows). The attribute **X** can have one of four values, 0, 1, 2 and 3. There are four bitmaps each corresponding to one of the four choices. For convenience, we have labeled the four bit sequences $b_1, \ldots, b_4$. To process the query “`select * from T where X < 2`,” one performs the bitwise logical operation b_1 OR b_2. Since

1 Information about the project is also available at `http://www.star.bnl.gov/STAR`.

bitwise logical operations are well supported by computer hardware, bitmap indexes are very efficient to use [21]. In many data warehouse applications, bitmap indexes are better than the tree based schemes [6, 21, 30], such as the variants of B-tree [9] or R-tree [11]. According to the performance model proposed by Jürgens and Lenz [14], the bitmap indexes are likely to be even more competitive in the future as the disk technology improves. In addition to supporting complex queries on one single table as shown in this paper, researchers have also demonstrated that bitmap indexes can accelerate complex queries involving multiple tables [23]. Realizing the value of the bitmap indexes, most major DBMS vendors have implemented them. The example shown in Figure 1 is the simplest form of the bitmap index we call the basic bitmap index.

A bitmap index is typically generated for each attribute. The basic bitmap index produces one bitmap for each distinct attribute value and it may perform the logical OR operation on as many as half of the bitmaps when answering a query involving the attribute. For attributes with low cardinality, a bitmap index is small compared to one of the tree based indexes and it can answer a query faster as well. To process the query "`Energy > 15 GeV and 7 <= NumParticles < 13`," a bitmap index on attribute `Energy` and a bitmap index on `NumParticles` are used separately to generate two bitmaps representing objects satisfying the conditions on `Energy` and `NumParticles`. A bitwise logical AND operation is sufficient to combine the two bitmaps to generate the final answer. These features make the bitmap index ideal for processing partial range queries. However, as in many real applications, the domain of many of the STAR attributes are continuous and the number of different values actually appear in the datasets are very large, in other words, the cardinalities of these attributes are very high. In these cases, the basic bitmap index generates too many bitmaps and operations on the bitmaps may also take too long.

In this paper, we propose to improve the effectiveness of the basic bitmap index by compression. Other ways of improving the bitmap index include binning and using different encoding. With binning, multiple values are grouped into a single bin and only the bins are indexed [15, 25, 28]. This strategy reduces the number of bitmaps used but it also introduces inaccuracies. In order to accurately answer a query, one has to scan some of the attribute values after operating on the indexes. Many researchers have studied the strategy of using different encoding schemes [6, 7, 22, 27, 30]. One well-known scheme is the bit-sliced index, that encodes k distinct values using $log_2 k$ bits and creates a bitmap for each binary digit [22]. This is related to the binary encoding scheme discussed elsewhere [6, 27, 30]. A drawback of this scheme is that to answer each query, most of the bitmaps have to be accessed, and possibly multiple times. There are also a number of schemes that generate more bitmaps than the bit-sliced index but access less of them while processing a query, for examples, the attribute value decomposition [6], interval encoding [7] and the K-of-N encoding [27]. We choose to concentrate on compression in this paper because it can be applied on any bitmap. Once we have identified some efficient compression schemes, we can improve all bitmap indexes. Additionally, a number of other common indexing schemes such as the signature file [10, 12, 16] and the bit transposed files [27] may also benefit from efficient bitmap compression algorithms.

Other high-dimensional indexing schemes yet to be mentioned include the projection index [22] and the UB-tree [4, 18, 19]. The projection index can be viewed as a different way of organizing the attribute values of a table. It can be implemented easily and efficiently by using bitmaps to store the intermediate results, and we use it as the bases for measuring the performance of our compressed bitmap index. The UB-Tree is a promising technique, regrettably we have to leave it out because of space limitations.

To compress the bitmap indexes, a simple option is to use one of the text compression algorithms, such as LZ77 (used in gzip) [17]. These algorithms are well-studied and effective in reducing file sizes. However, performing logical operations on the compressed data are usually significantly slower than on the uncompressed data. To address this performance issue, a number of special algorithms have been proposed. Johnson and colleagues have conducted extensive studies on their performances [13, 1]. From their studies, we know that the logical operations using these specialized schemes are usually faster than those using gzip. One such specialized algorithm, called the *Byte-aligned Bitmap Code* (BBC), is known to be very efficient. It is used in a commercial database system [2, 3]. However, even with BBC, in many cases logical operations on the compressed data still can be orders of magnitudes slower than on the uncompressed data.

In this paper, we propose a simple algorithm for compressing the bitmap indexes that improves the speed of logical operations by an order of magnitude at a cost of small increase in space. We call the method the Word-aligned Hybrid (WAH) compression scheme. This algorithm not only supports faster logical operations but also enables the bitmap index to be applied to attributes with high cardinalities. Our tests show that by using WAH compression, we can achieve good performance on scientific datasets where most attributes have high cardinalities. From their performance studies, Johnson and colleagues came to the conclusion that one has to dynamically switch among different compression schemes in order to achieve the best performance [1]. We found that since WAH is significantly faster than earlier compression schemes, there is no need to switch compression schemes in a bitmap indexing software. The new compression scheme not only improves the performance of the bitmap indexes but also simplifies the indexing software.

The remainder of this paper is organized as follows. In Section II we review three commonly used compression schemes and identify their key features. These three were selected as representatives in our performance comparisons. Section III contains the description of the word-aligned hybrid code (WAH). Section IV contains some timing results of the bitwise logical operations. Some timing information on processing range queries are presented in section V. A short summary is given in Section VI.

2. Review of byte based schemes

In this section, we briefly review three well known schemes for representing bitmaps and introduce the terminology needed to described our new scheme. These three schemes are selected as representatives from a number of schemes studied previously [13, 29].

A straightforward way of representing a bitmap is to use one bit of computer memory for each bit of the bitmap. We call this the *literal* (LIT) *bit vector*[2]. This is the uncompressed scheme and logical operations on uncompressed bitmaps are extremely fast.

The second type of scheme in our comparisons is the general purpose compression scheme such as gzip [17]. They are highly effective in compressing data files. We use gzip as the representative because it is usually faster than others in decompressing the data files.

As mentioned earlier, there are a number of compression schemes that offer good compression and also allow fast bitwise logical operations. One of the best known schemes is the Byte-aligned Bitmap Code (BBC) [2, 3, 13]. The BBC scheme performs bitwise logical operations efficiently and it compresses almost as well as gzip. We use BBC as the representative for these types of schemes. Our implementation of the BBC scheme is a version of the two-sided BBC code [29, Section 3.2]. This version performs as well as the improved version by Johnson [13]. In both Johnson's tests [13] and ours, the time curves for BBC and gzip (marked at LZ in [13]) cross at about the same position.

Many of the specialized bitmap compression schemes, including BBC, are based on the basic idea of run-length encoding that represents consecutive identical bits (also called a *fill* or a *gap*) by their bit value and their length. The bit value of a fill is called the fill bit. If the fill bit is zero, we call the fill a *0-fill*, otherwise it is a *1-fill*. Compression schemes generally try to store repeating bit patterns in compact forms. The run-length encoding is among the simplest of these schemes. This simplicity allows logical operations to be performed efficiently on the compressed bitmaps.

Different run-length encoding schemes commonly differ in their representations of the fill lengths and the short fills. A naive run-length code may use a word to represent all fill lengths. This is ineffective because it uses more space to represent short fills than in the literal scheme. One common improvement is to represent the short fills literally. The second improvement is to use as few bits as possible to represent the fill length. Given a bit sequence, the BBC scheme first divides it into bytes and then groups the bytes into *runs*. Each BBC run consists of a fill followed by a *tail* of literal bytes. Since a BBC fill always contains a number of whole bytes, it represents the fill length as the number of bytes rather than the number of bits. In addition, it uses a multi-byte scheme to represent the fill lengths [2, 13]. This strategy often uses more bits to represent a fill length than others such as ExpGol [20]. However it allows for faster operations [13].

Another property that is crucial to the efficiency of the BBC scheme is the byte alignment. This property limits a fill length to be an integer multiple of bytes. More importantly, it ensures that during any bitwise logical operation a tail byte is never broken into individual bits. Because working on individual bits is much less efficient than working on whole bytes on most CPUs, byte-alignment is crucial to the operational efficiency of BBC. Removing the alignment may lead to better compression. For example, the ExpGol scheme [20] can compress better than BBC partly because it does not obey the byte alignment. However, bitwise logical operations on ExpGol bit vectors are often much slower than on BBC bit vectors [13].

3. Word based schemes

Most of the known compression schemes are byte based, that is, they access computer memory one byte

2 We use the term bit vector to describe the data structure used to represent the compressed bitmaps.

at a time. On most modern computers, accessing one byte takes as much time as accessing one word [24]. A computer CPU with MMX technology offers the capability of performing a single operation on multiple bytes. This may automatically turn byte accesses into word accesses. However, because the bytes in a compressed bit vector typically have complex dependencies, logical operations implemented in high-level languages are unlikely to take advantage of the MMX technology. Instead of relying on the hardware and compilers, we developed a new scheme that accesses only whole words. It is named the *word-aligned hybrid code* (WAH). We have previously considered a number of word-based schemes and this is the most efficient one in our tests [29].

The word-aligned hybrid (WAH) code is similar to BBC in that it is a hybrid between the run-length encoding and the literal scheme. Unlike BBC, WAH is much simpler and it stores compressed data in words rather than in bytes. There are two types of words in WAH: *literal* words and *fill* words. In our implementation, we use the most significant bit of a word to distinguish between a literal word (0) and a fill word (1). This choice allows one to easily distinguish a literal word from a fill word without explicitly extracting the bit. The lower bits of a literal word contain the bit values from the bitmap. The second most significant bit of a fill word is the fill bit and the lower bits store the fill length. WAH imposes the word-alignment requirement on the fills, it requires that all fill lengths be integer multiples of the number of bits in a literal word. The word-alignment ensures that logical operation functions only need to access words not bytes or bits.

Figure 2 shows a WAH bit vector representing 128 bits. In this example, we assume each computer word contains 32 bits. Under this assumption, each literal word stores 31 bits from the bitmap and each fill word represents a fill with a multiple of 31 bits. If the machine has 64-bit words, each literal word would store 63 bits from the bitmap and each fill would have a multiple of 63 bits. The second line in Figure 2 shows how the bitmap is divided into 31-bit groups and the third line shows the hexadecimal representation of the groups. The last line shows the values of the WAH words. The first three words are normal words, two literal words and one fill word. The fill word `80000002` indicates a 0-fill of two-word long (containing 62 consecutive zero bits). Note that the fill word stores the fill length as two rather than 62. In other word, we represent the fill length as multiples of the literal word size. The fourth word is the *active word* that stores the last few bits that can not be stored in a normal word, and another word (not shown) is needed to stores the number of useful bits in the active word.

The logical operation functions are easy to implement but are tedious to describe. To save space, we refer the interested reader to a technical report [29]. Here we only briefly describe one example, see Figure 3. In this example, the first operand of the logical operation is the one in Figure 2. To perform a logical operation, we basically need to match each group of 31 bits from both operands and generate the groups for the result using the hardware support to perform the operations between groups of 31 bits. Each column of the table is reserved to represent one such group. A literal word occupies the location for the group and a fill word is given at the space reserved for the first group it represents. The first 31-bit group of the result C is the same as that of A because the corresponding group in B is part of a 1-fill. The next three groups of C contain only zero bits. The active words are always treated separated.

Figure 3 shows a decompressed version of the three bitmaps involved in the operation for the purpose of illustration only. The logical operations can be directly performed on the compressed bitmaps and the time needed by one such operation on two operands is related to the sizes of the compressed bitmaps. Let the compression ratio be the ratio of size of a compressed bitmap and its uncompressed counterpart. When the average compression ratio of the two operands are less than 0.5, the logical operation time is expected to be proportional to the average compression ratio [29].

4. Performance of the logical operations

In this section, we discuss the performance of the logical operations. Ultimately we are interested in enhancing the speed of query processing. However, because logical operations are the main operations on the bitmaps and their performances are directly affected by the compression schemes, we discuss the performances of the logical operations first.

The WAH compression scheme are compared against the three schemes reviewed in Section 2. The tests are conducted on three sets of data, a set of random bitmaps, a set of bitmaps generated from a Markov process and a set of bitmap indexes on some real application data. Each synthetic bitmap has 100 million bits. The synthetic data are controlled through two parameters, the *bit density* and the *clustering factor*. In a bitmap, the bit density is the fraction of bits that are one and the clustering factor is the average length of the 1-fills. The random bitmaps are generated according to the bit density

128 bits	1,20*0,3*1,79*0,25*1				
31-bit groups	1,20*0,3*1,7*0	62*0		10*0,21*1	4*1
groups in hex	40000380	00000000	00000000	001FFFFF	0000000F
WAH (hex)	40000380	80000002		001FFFFF	0000000F

Figure 2. A WAH bit vector. Each WAH word (last row) represents a multiple of 31 bits from the bit sequence, except the last word that represents the four leftover bits.

	decompressed				
A	40000380	00000000	00000000	001FFFFF	0000000F
B	7FFFFFFF	7FFFFFFF	7C0001E0	3FE00000	00000003
C	40000380	00000000	00000000	00000000	00000003
	compressed				
A	40000380	80000002		001FFFFF	0000000F
B	C0000002		7C0001E0	3FE00000	00000003
C	40000380	80000003			00000003

Figure 3. A bitwise logical AND operation on WAH compressed bitmaps, C = A AND B.

and the Markov process generates bitmaps with a specified bit density and clustering factor. The goal of this test is to examine the performance of the different compression schemes under various conditions. However to limit the number of test cases, we restrict all synthetic bitmaps to have bit density no more than 1/2. Since all compression schemes can compress 0-fills and 1-fills equally well, the performance on high bit density bitmaps should be the same as on their complements. When necessary to distinguish the two type of synthetic bitmaps, we refer to them as the random bitmaps and the Markov bitmaps according to how they are generated. The real application is a high-energy physics experiment called STAR [25, 26]. The data used in our tests can be viewed as one relational table consisting of about 2.2 million tuples and 500 attributes. The bitmaps used in this test are bitmap indexes on a set of 12 most frequently queried attributes.

We have conducted a number of tests on different machines and found that the relative performances among the different compression schemes are independent of the specific machine architecture. This characteristic was also observed in a different performance study [13]. The main reason for this is that most of the clock cycles are consumed by branching operations such as "if" tests and "loop condition" tests. These operations only depend on the clock speed. For this reason, we only report the timing results from a Sun Enterprise 450[3] that is based 400 MHz UltraSPARC II CPUs. The test data were stored in a file system striped across five disks connected to an UltraSCSI controller and managed by a VERITAS Volume Manager[4]. The VERITAS software distribute files across the five disks to maximize the IO performance. The machine has four gigabytes (GB) of RAM which is large enough to store each of our test cases in memory. The secondary cache size is 4 MB. In most cases, this cache is too small to store the two operands and the result of a logical operation.

Because of space limitations, we only show performance of the logical OR operations in the following discussions. On the same machine, a logical AND operation typically takes slightly less time than a logical OR operation on the same bit vectors, and a logical XOR operation typically takes slightly more time. In general, if WAH is X times faster than BBC in performing a logical OR operation, the same would also be true for the two other logical operations.

The most likely scenario of using these bit vectors in a database system is to read a number of them from disks and then perform bitwise logical operations on them. In most cases, the bit vectors simply need to be read into memory and stored in the corresponding in-memory data structures. Only the gzip scheme

3 Information about the E450 is available at `http://www.sun.com/servers/workgroup/450`.

4 Information about VERITAS Volume Manager is available at `http://www.veritas.com/us/products`.

needs a significant amount of CPU cycles to decompress the data files into the literal representation before actually performing the logical operations. In our tests involving gzip, only the operands of logical operations are compressed; the results are not. This is to save time. Had we compressed the result as well, the operations would take several times longer than those reported in this paper because the compression process is more time-consuming [29]. We use the *direct* method for both BBC and WAH. In other word, a logical operation directly operates on two compressed operands and produces a compressed result. It is one of the four strategies studied by Johnson [13]. We have chosen the direct method because it requires less memory and is often faster than the alternative methods.

Figure 4 shows the time it takes to perform the bitwise logical OR operations on the random bitmaps. Each data point shows the time to perform a logical operation on two bitmaps with similar bit densities. Figure 4(a) shows the logical operation time and Figure 4(b) shows the total time including the time to read the two bitmaps from files. In most cases, the IO time is a relatively small portion of the total time for BBC and WAH. Neglecting the IO time does not significantly change the relative performance between WAH and BBC. In an actual application, once the bitmaps are read into memory, they are likely to be used more than once. The average cost of a logical operation would be close to what is shown in Figure 4(a). From now on when showing the logical operation time, we will not include the IO time.

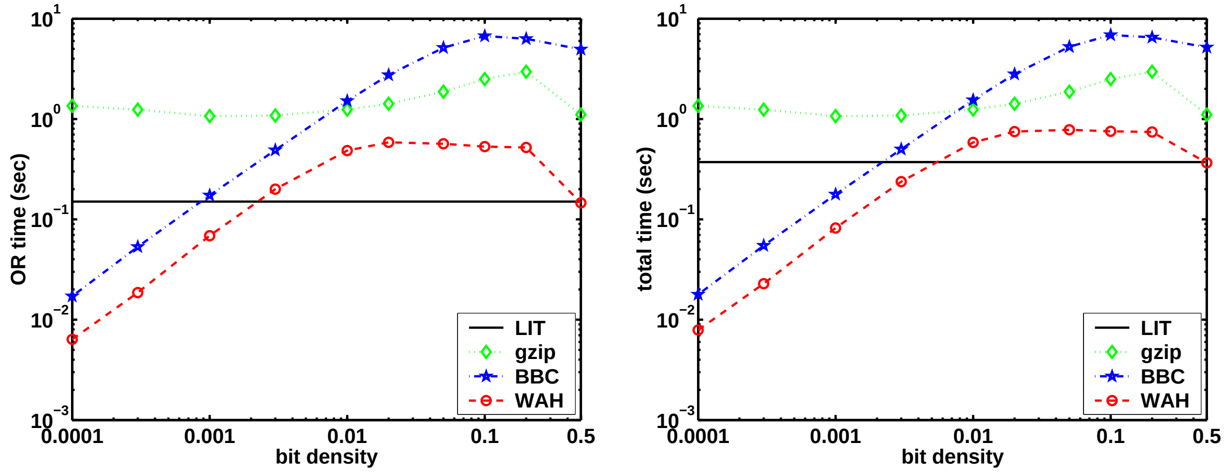
Among the schemes shown, it is clear that WAH uses much less time than either BBC or gzip. In all test cases, the gzip scheme uses at least three times more time than the literal scheme. In almost half of the test cases, BBC takes more than ten times longer than WAH.

When the bit density is about 1/2, the random bitmaps are not compressible by WAH. For convenience, we refer to the bit vectors only literal words as the decompressed bit vectors. Usually, each logical operation function takes two compressed bit vectors and generates a compressed result, but the functions that perform logical operations on decompressed bit vectors always generate decompressed results. It's easy to see that the logical operations on decompressed WAH bit vectors is nearly as fast as on the literal bit vectors. Unless one explicitly decompress a BBC bit vector, it is very unlikely to have a decompressed BBC bit vector. Even with bit density of 1/2, a BBC bit vector still contains a number of short fills. Even if we explicitly decompress the bit vectors, operations on decompressed BBC bit vectors are not as efficient as on literal bit vectors. In Figure 4, the line for WAH falls on top of the one for the literal scheme at bit density of 1/2 but the line for BBC only shows a slight dip.

In Figure 4 we see that when bit density is above 0.01, WAH performs logical operations slower than the literal scheme. Since on the uncompressed bitmaps WAH can perform logical operations as well as the literal scheme, we might store those dense bitmaps without compression and expect the logical operations to be as fast as in the literal scheme. However, doing so significantly increases the space requirement and it does not even guarantee the speed of logical operation is always the fastest. This leads us to take a more careful look at the compression effectiveness and factors that determine the logical operation speed.

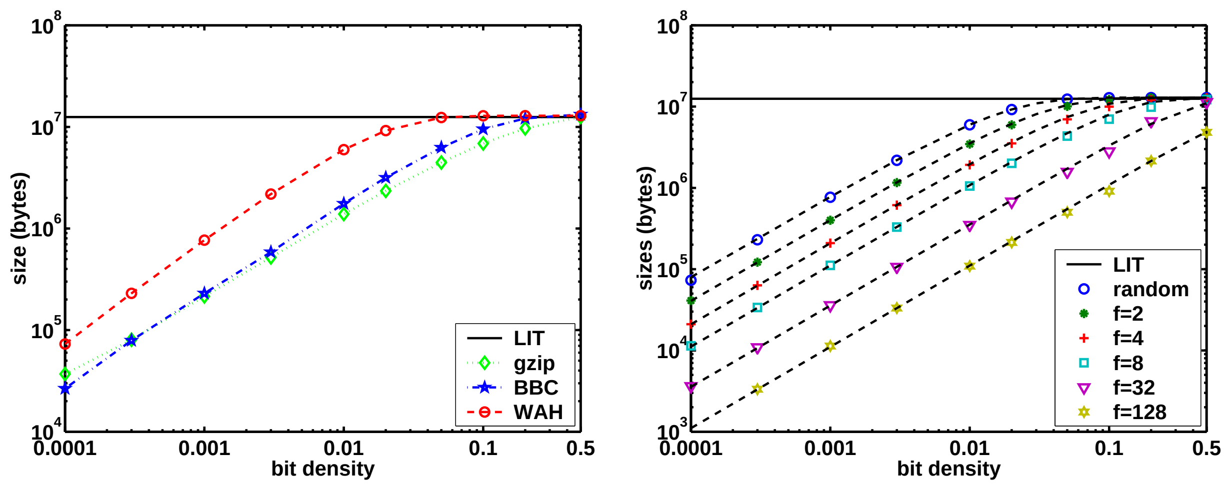
Figure 5 shows the sizes of the four types of bit vectors. Each data point in this figure represents the average size of a number of bitmaps with the same bit density and clustering factor. As the bit density increases from 0.0001 to 0.5, the bit sequences become less compressible and it takes more space to represent them. When the bit density is 0.0001, all four compression schemes use less than 1% of the disk space required by the literal scheme. At a bit density of 0.5, the test bitmaps become incompressible and the compression schemes all use slightly more space than the literal scheme. In most cases, WAH uses more space than the two byte based schemes, BBC and gzip. For bit density between 0.001 and 0.01, WAH uses about 2.5 ($\sim 8/3$) times the space as BBC bit vectors. In fact, in extreme cases, WAH may use four times as much space as BBC. Fortunately, these cases do not dominate the total space required by a bitmap index. In a typical bitmap index, the set of bitmaps contains some that are easy to compress and some that are hard to compress, and the total size is dominated by the hard to compress ones. Since most schemes use about the same amount of space to store these hard to compress ones, the differences in total sizes are usually much smaller than the extreme cases. For example, on the set of STAR data, the bitmap indexes compressed using WAH are about 60% bigger than those compressed using BBC, see Figure 7. This is a fairly modest increase in space compared to the increase in speed.

To verify that the logical operation time is proportional to the sizes of the operands, we plotted the timing results of the two sets of synthetic bitmaps together in Figure 6(a) and the results on the STAR bitmaps in Figure 6(b). In both cases, the compression ratio is used as the horizontal axes. Since in each plot, the bitmaps are of the same length, the sizes are directly proportional to the compression ratios. In each plot, a sym-



(a) logical OR time (b) total time (including IO)

Figure 4. CPU seconds needed to perform a bitwise OR operation on two random bitmaps.



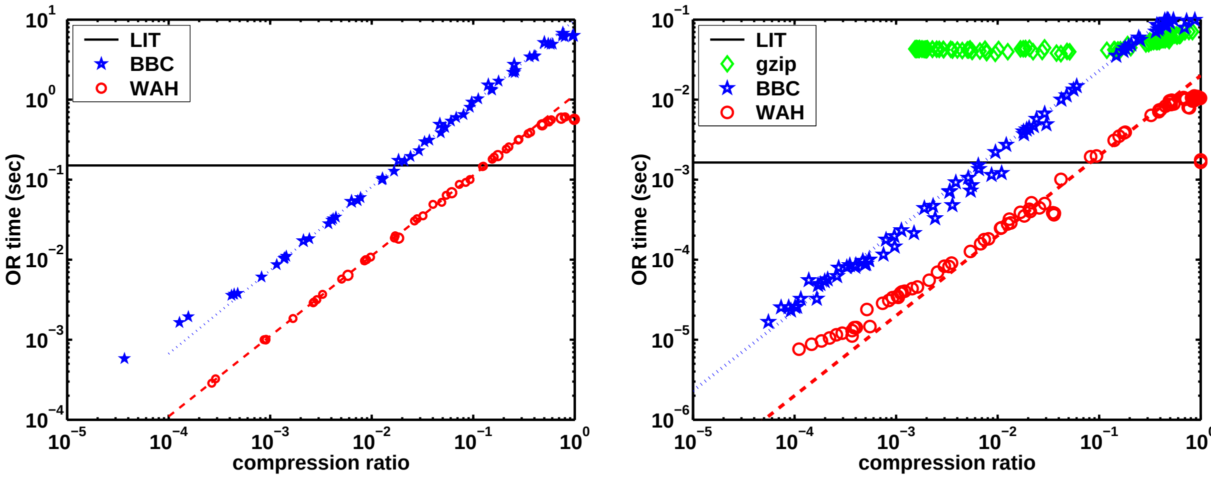
(a) random (b) Markov (WAH only)

Figure 5. The sizes of the compressed bit vectors. The symbols for the Markov bitmaps are marked with their clustering factors.

bol represents the average time of logical operations on bitmaps with the same size. The dashed and dotted lines are produced from linear regressions. Most of the data points near the center of the graphs are close to the regression lines. Those logical operations involving bit vectors with high compression ratios are nearly constant. For very small bit vectors, where the logical operation time is measured to be a few microseconds, the logical operations time deviates from the linear relation because of the overheads such as the timing overhead, function call overhead and other lower order terms in the complexity expression. The regression lines for WAH and BBC are about a factor of ten apart in both plots.

The performance differences between WAH and BBC can be attributed to three main factors.

1. The encoding scheme of WAH is much simpler than BBC. WAH has only two kinds of words and one test is sufficient to determine the type of any given word. In contrast, our implementa-



(a) all synthetic bitmaps (b) STAR bitmap indexes

Figure 6. Logical operation time is almost proportional to compression ratio. The STAR bitmap indexes are on the 12 most queried attributes.

tion of BBC has four different types of runs, other implementations have even more [13]. It may take up to three tests in order to decide the run type of a header byte. After deciding the run type, many clock cycles may still be needed to fully decode a run to determine the fill length or the tail value.

2. During the logical operations, WAH always accesses whole words, while BBC accesses bytes. On most bitmaps, BBC needs more time to load its data from the main memory to CPU registers than WAH.

3. BBC can encode shorter fills more compactly than WAH, however, this comes at a cost. Each time BBC encounters a short fill, say a fill with less than 8 bytes, it starts a new run. WAH typically represent such a short fill literally. It is much faster to operate on a WAH literal word than on a BBC run. This situation is common when bit density is greater than 0.01 in random bitmaps.

If we sum up the execution time of all logical operations performed on the STAR bitmaps for each compression scheme, the total time for BBC is about 12 times that of WAH. Much of this difference can be attributed to factor 3 discussed above. There are a number of bitmaps that can not be compressed by WAH but can be compressed by BBC. When operating on these bitmaps, WAH is nearly 100 times faster than BBC. On very sparse bit vectors, WAH is about four to five times faster than BBC.

Compared to the literal scheme, BBC is faster in a fraction of the test cases, however, WAH is faster in more than 60% of the test cases. In the worst case, BBC can be nearly 100 times slower than the literal scheme, but WAH is only 6 times slower. It might be desirable to use the literal scheme in some cases. To reduce the complexity of the software, we suggest one to use WAH but only use the literal words. Regarding whether to store random bitmaps with bit density greater than 0.01 without compression, we recommend that the bitmaps be compressed.

5. WAH improves bitmap index effectiveness

In this section, we use a set of real application data from STAR to demonstrate the effectiveness of WAH compressed bitmap index. The frequently queried attributes can be organized as a relational table consisting of millions of tuples and hundreds of attributes. A typical query is a range query involving a handful of attributes. If `Energy` and `NumParticles` are two attributes of the table, a query on them might be "`Energy > 15 GeV and 7 <= NumParticles < 13`". In addition, most user queries may involve different attributes and different number of them. Queries of this form, which we call *partial range queries*, are particularly difficult for most database systems. For example, if a B-tree index is created for each attribute, a commercial DBMS usually selects one of them to resolve part of the query and then scans the table to fully resolve the query. This approach often takes more time than simply scanning the table without using an index.

Commonly used multidimensional indexing schemes such as variations of R-tree [11] are not effective for two reasons. Most of these schemes are only effective when the number of attributes are no more than ten, but the STAR dataset has hundreds of attributes. In addition, if a query does not involve all attributes indexed, these multidimensional indexes are not effective in processing the query. A number of researchers have confirmed that the projection index and the bitmap index are among the fastest schemes in processing partial range queries [14, 21, 22]. The projection index is simply another name for vertical partitioning a relational table, we store the values of an attribute consecutively rather than storing the values of a tuple consecutively. In this case, queries are processed by simply compare on the values. In later discussions, we will refer to this as the *projection scan* or *p scan* for short.

Our goal is to demonstrate that WAH compression can improve the performance of the bitmap indexing scheme. To do this, we perform two sets of tests. The first one is on some low cardinality attributes and the second is on some high cardinality attributes. The bitmap index is usually thought to be efficient for low cardinality attributes. In this case, we show that the WAH compressed indexes are not only smaller than the uncompressed ones but are also more efficient in answering range queries. When the cardinalities are high, it is impractical to generate the uncompressed indexes. In this case, we show that the WAH compressed indexes are still of reasonable sizes and can process range queries faster than the BBC compressed indexes and the projection index. The high cardinality case are of particular interests to us because the most frequently queried attributes of the STAR data have high cardinality.

In our tests, the low cardinality attributes are the 12 attributes with the lowest cardinalities from the STAR data, and the high cardinality attributes are the 12 attributes that are most likely to be queried by a physicist. All low cardinality attributes are four-byte integers; the frequently queried attributes are mostly four-byte integers and floating-point values except one attribute is eight-byte floating-point value. The total size for the first set is about 104 MB and the second one is 113 MB.

Figure 7 shows the sizes of the bitmap indexes. Four columns are displayed in each table. Column 'c' shows the cardinalities of the attributes. Columns marked 'WAH' and 'BBC' are our stand-alone implementations of the compressed bitmap indexes. The column marked 'DBMS' shows the sizes of the bitmap indexes in a commercial DBMS. Since the particular DBMS implements a BBC compressed bitmap index, conceptually it is equivalent to our BBC compressed bitmap index.

In the first data set, there are a total of 312 distinct values, i.e., there are 312 bitmaps in all bitmap indexes. Without compression, 312 bitmaps use about 84MB. All three versions of the compressed bitmap indexes are less than 10% of this size and are less than 7% of the data size.

In the second data set, there are nearly 2.7 million distinct values. Without compression, the bitmap index size would be more than 720GB (more than 6000 times the data size). Both BBC and WAH are very effective in reducing the sizes of the bitmap indexes because the majority of the bitmaps are very sparse. The total size of each set of the compressed bitmap indexes is less than half of the size of the B-tree indexes. Using the DBMS, the total size of 12 B-tree indexes take about 400 MB, nearly four times the size of the data.

Figure 8 shows the average query processing time of three compressed bitmap indexes and the projection index on the high cardinality data set. The three bitmap indexes are the same as in Figure 7. The query processing time is measured from the client side, and therefore includes network communication time as well as the time to actually answer the query. The partial range queries are generated by randomly selecting some attributes and constructing a query with the specified query box size. The *query box* is defined to be the ratio of the volume of the hypercube formed by the ranges to the total volume of the attributes [18]. For example, let the values of `Energy` be in the range of 0 to 30 GeV and `NumParticles` in the range of 1 to 15, the query box size of "`Energy > 15 GeV and 7 <= NumParticles < 13`" is $15/31 \times 6/15 = 0.19$. 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.

We also show the time used by the projection index, marked as 'p scan', in Figure 8. The projection index only access the attributes involved in a query and is much faster than most indexing strategies [22]. For example, on our test machine, the DBMS takes about 6.5 seconds to scan a table with 12 attributes while the projection scan only need 0.56 ($\approx 6.5/12$) seconds. Had we actually stored all 500 attributes in the table, the DBMS would take nearly 5 minutes to per-

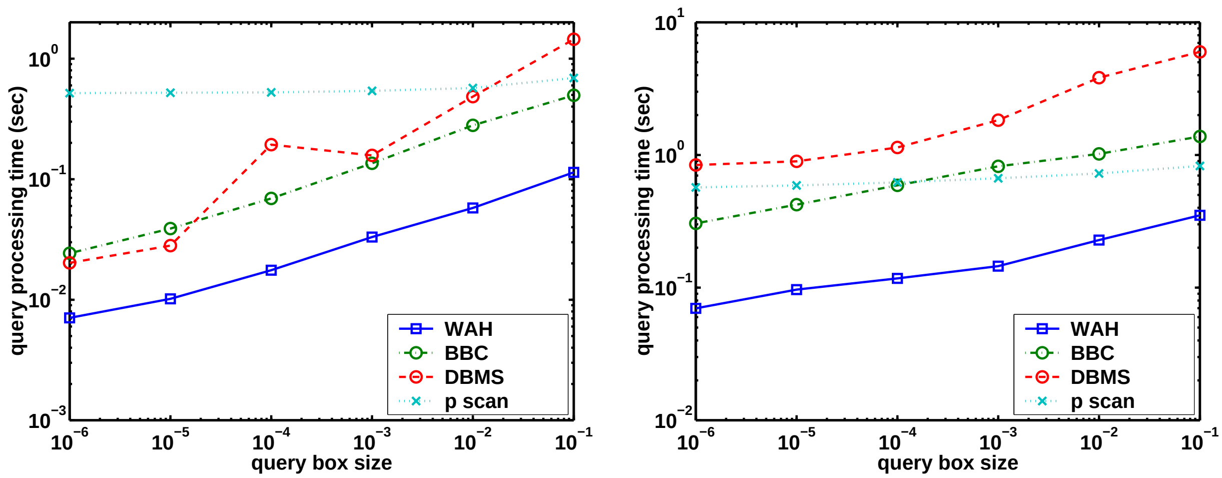
c	WAH	BBC	DBMS
4	10,196	8,733	335,037
4	305,296	164,665	421,074
18	1,510,740	924,035	1,077,269
19	1,437,892	842,359	1,001,476
24	1,703,456	975,465	1,127,116
25	1,729,380	988,060	1,140,852
33	33,568	9,516	334,420
35	151,808	39,254	349,970
35	151,708	39,222	349,771
35	151,808	39,257	349,797
40	1,964	1,534	330,128
40	1,972	1,599	329,785
total (DBMS B+-Tree: 370,631,794 bytes)			
312	7,189,788	4,033,699	7,146,695

(1) 12 low cardinality attributes

c	WAH	BBC	DBMS
40	1,964	1,534	340,946
40	1,972	1,599	340,573
116	10,339,232	3,393,224	3,473,910
367	10,585,524	3,164,756	3,572,127
371	23,436	16,622	350,916
1,688	11,855,904	3,858,185	4,271,522
1,807	16,182,848	4,922,029	5,414,222
3,786	10,973,128	3,827,861	4,122,542
76,920	19,849,220	8,874,753	8,642,620
514,516	20,807,036	18,059,791	15,606,417
818,300	33,036,432	28,014,187	25,763,032
1,255,695	52,427,916	43,689,012	39,122,608
total (DBMS B+-Tree: 408,149,316 bytes)			
2,673,646	186,084,612	117,823,553	111,021,435

(2) 12 most commonly queried attributes

Figure 7. Sizes (Bytes) of the bitmap indexes stored in various schemes.



(a) 2 attributes per query

(b) 5 attributes per query

Figure 8. The average query processing time of random range queries on the 12 most queried attributes of the STAR data.

form its scan operation. Clearly, the projection scan is fast. We also take full advantage of the fast bitmap data structure to store the intermediate results. When evaluating conjunctive queries, the result of the left side can be used as the mask to limit the amount work needed to evaluate the right side. A sophisticated execution planner could easily determine an evaluation order that minimizes the total amount of work. However, our stand-alone indexing software does not have such a planner. Nevertheless, simply using a mask has reduced the amount of work tremendously. This is reflected in the case where the projection scan time is always quite close to 0.56 seconds.

We see that WAH compressed bitmap indexes are significantly more efficient than the BBC compressed indexes. When there are two attributes per query, WAH compressed indexes are about four times faster than the stand-alone BBC compressed indexes and 10 times faster than the DBMS. When there are five attributes per query, WAH compressed indexes are nearly five times faster than the stand-alone BBC compressed indexes and 14 times faster than the DBMS. In all cases, our WAH compressed bitmap indexes are at least twice as fast as the projection index. When the query box sizes are small, it can be orders of magnitudes faster than the projection scan.

We saw in the previous section that on the average, WAH can perform logical operations 12 times faster than BBC, but in this section we observe that the query processing speed only differs by a factor of four to five. This is in part because much of the time is spent on performing logical operations on very sparse bitmaps where WAH was measured to be about four to five times faster than BBC. In addition, we have only improved the speed of logical operations which is only one part of the time spent in query processing. Other operations, such as network communication, query parsing, and locking overhead, used to be insignificant part of the total execution time now become more important after we have dramatically reduced the logical operation time.

Comparing the commercial implementation of BBC with our own, we found that the commercial implementation performs slower than ours. This is clearly evident when a large number of logical operations are needed, as in the cases of processing queries on high cardinality attributes, see Figure 8. Next, we examine whether the same behavior persists on low cardinality attributes.

Figure 9 shows the average query processing time on the 12 low cardinality attributes. From Figure 9 we see that it always takes less time to use the WAH compressed bitmap indexes. The two versions of BBC compressed bitmap indexes (the stand-alone version and the commercial version) take about the same amount of time when there are two attributes in a query. However, the DBMS takes less time than the stand-alone version when there are five attributes in a query. This is because the DBMS uses a better execution plan than the stand-alone version. For example, if `NumParticles` actually have only three values, 1, 3, and 15, even tough our sample query "`Energy > 15 GeV and 7 <= NumParticles < 13`" has a query box size of 0.19, it generates no hits. If the condition on `NumParticles` is evaluated first, there is no need to evaluation the condition on `Energy`. Since the stand-alone version has not implemented any query planning functionality, it evaluates the condition on `Energy` first and wastes time. The cost saving due to this query planning functionality is more significant when more attributes are involved.

Figure 9 also contains the timing information of the uncompressed bitmap indexes, marked as "LIT." The BBC compressed indexes often takes more time than the uncompressed indexes, but the WAH compressed indexes are always faster. In many cases, the WAH compressed indexes only needs about a third of the time used by the uncompressed indexes to process the same queries.

6. Summary

This research was motivated by the need to improve the query response time of a scientific data management project. Based on the characteristics of the dataset and queries, the bitmap indexing strategy is a good choice. However because most of the commonly queried attributes have a large number of distinct values, the basic bitmap index takes too much space and query response time is too long. This paper describes a compression scheme for addressing these performance issues. It is well accepted that I/O dominates the operational efficiency of out-of-core indexing methods. Thus, most compression schemes designed for bitmap indexes only attempt to minimize I/O, i.e., reduce the size of the bitmaps. Compressing bitmap indexes using these schemes doesn't lead to optimal query response time. Our tests show that the computation time dominates the total time. In addition, as main memories become cheaper, we expect that "popular" bitmaps will remain in memory once they are used. For these reasons, we pursued the course of improving the computational efficiency of operations over bitmaps. The best existing bitmap compression schemes are byte-aligned. In this paper, we presented a word-aligned scheme WAH, that is not only much simpler but is also very CPU-friendly. This ensures that the logical operations are performed efficiently. Tests on a set of real application data show that it is 12 times as fast as BBC while using only 60% more space.

We also demonstrated from tests that improving the compression scheme actually improves the query answering speed, not only logical operations. Tests show that WAH compressed indexes are not only smaller than the uncompressed indexes, they also take less time to answer queries. Compared to the indexes compressed with BBC, the WAH compressed indexes are faster by a factor of four or five. We did not see a factor of 12 improvement because the time spent in query processing are dominated by logical operations on very sparse bitmaps. On very sparse bitmaps, WAH scheme is faster than BBC usually by a factor of about four or five. During query processing there is also some amount of time spent in parsing the query, obtaining the locks and so on. The time for these operations have not been reduced by using a different compression scheme. In

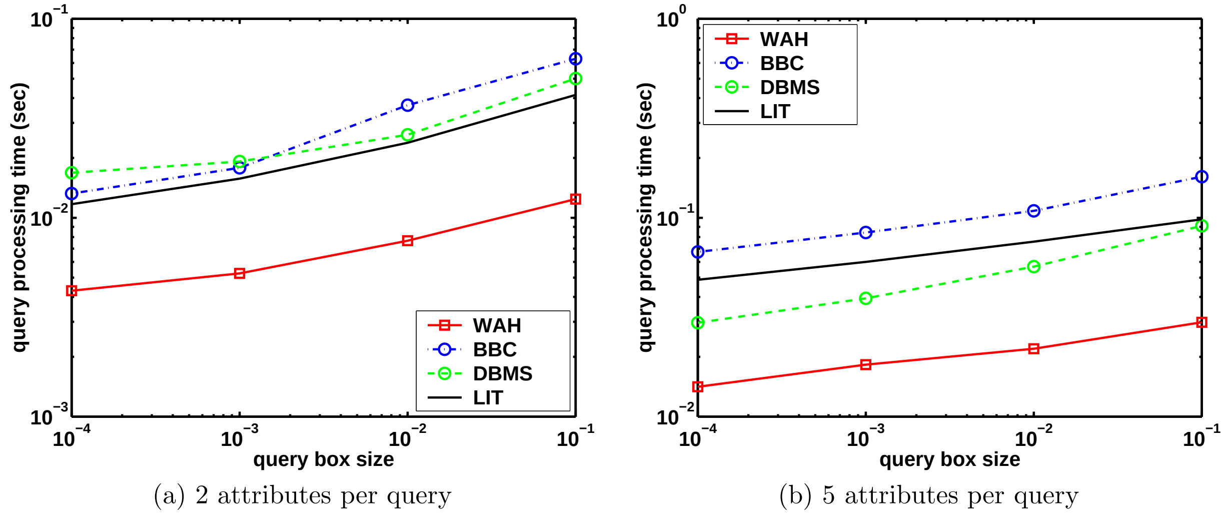


Figure 9. The average query processing time of random range queries on the 12 low cardinality attributes of the STAR data.

spite of all these, we believe it is worthwhile to use WAH instead of BBC to compress bitmap indexes.

The bitmap index is often thought to be effective only on low cardinality attributes. By using WAH, we demonstrated that it is effective even for attributes with thousands of distinct values.

7. Acknowledgments

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. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

References

[1] Sihem Amer-Yahia and Theodore Johnson. Optimizing queries on compressed bitmaps. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, 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] Rudolf Bayer. UB-trees and UB-cache – A new processing paradigm for database systems. Technical Report TUM-I9722, TU Mnchen, 1997.

[5] Luis M. Bernardo, Arie Shoshani, Alex Sim, and Henrik Nordberg. Access coordination of tertiary storage for high energy physics applications. In *IEEE Symposium on Mass Storage Systems*, pages 105–118, 2000.

[6] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD: International Conference on Management of Data*. ACM press, 1998.

[7] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. ACM Press, 1999.

[8] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, March 1997.

[9] Douglas Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[10] K. Furuse, K. Asada, and A. Iizawa. Implementation and performance evaluation of compressed bit-sliced signature files. In Subhash Bhalla, editor, *Information Systems and Data Management, 6th International Conference, CISMOD'95, Bombay, India, November 15-17, 1995, Proceedings*, volume 1006 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1995.

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

[12] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evalution of signature files as set access facilities in OODBs. In P. Buneman and S. Jajodia, editors, *Proceedings ACM*

SIGMOD International Conference on Managerment of Data, May 26-28, 1993, Washington, D.C., pages 247–256. ACM Press, 1993.

[13] T. Johnson. Performance measurements of compressed bitmap indices. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 278–289. Morgan Kaufmann, 1999. A longer version appeared as AT&T report number AMERICA112.

[14] M. Jürgens and H.-J. Lenz. Tree based indexes vs. bitmap indexes - a performance study. In S. Gatziu, M. A. Jeusfeld, M. Staudt, and Y. Vassiliou, editors, *Proceedings of the Intl. Workshop on Design and Management of Data Warehouses, DMDW'99, Heidelberg, Germany, June 14-15, 1999*, 1999.

[15] Nick Koudas. Space efficient bitmap indexing. In *Proceedings of the ninth international conference on Information knowledge management CIKM 2000 November 6 - 11, 2000, McLean, VA USA*, pages 194–201. ACM, 2000.

[16] D. L. Lee, Y. M. Kim, and G. Patel. Efficient signature file methods for text retrieval. *IEEE Transactions on Knowledge and Data Engineering*, 7(3), 1995.

[17] Jean loup Gailly and Mark Adler. *zlib 1.1.3 manual*, July 1998. Source code available at `http://www.info-zip.org/pub/infozip/zlib`.

[18] V. Markl and R. Bayer. Processing relational OLAP queries with UB-trees and multidimensional hierarchical clustering. In M. A. Jeusfeld, H. Shu, M. Staudt, and G. Vossen, editors, *Proceedings of the Second Intl. Workshop on Design and Management of Data Warehouses, DMDW 2000, Stockholm, Sweden, June 5-6, 2000*, 2000.

[19] Volker Markl. *MISTRAL: processing relational queries using a multidimensional access technique*. PhD thesis, Institut für Informatik der Technischen Universität München, 1999.

[20] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In N. Belkin, P. Ingwersen, and A. M. Pejtersen, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval, Copenhagen, June 1992*, pages 274–285. ACM Press, 1992.

[21] P. O'Neil. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems, Asilomar, CA*, volume 359 of *Springer-Verlag Lecture Notes in Computer Science*, pages 40–59, September 1987.

[22] P. O'Neil and D. Quass. Improved query performance with variant indices. In Joan Peckham, editor, *Proceedings ACM SIGMOD International Conference on Managerment of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 38–49. ACM Press, 1997.

[23] P. E. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[24] D. A. Patterson, J. L. Hennessy, and D. Goldberg. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[25] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional indexing and query coordination for tertiary storage management. In *11th International Conference on Scientific and Statistical Database Management, Proceedings, Cleveland, Ohio, USA, 28-30 July, 1999*, pages 214–225. IEEE Computer Society, 1999.

[26] K. Stockinger, D. Duellmann, W. Hoschek, and E. Schikuta. Improving the performance of high-energy physics analysis through bitmap indices. In *11th International Conference on Database and Expert Systems Applications DEXA 2000, London, Greenwich, UK*, September 2000.

[27] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *Proceedings of VLDB 85, Stockholm*, pages 448–457, 1985.

[28] K.-L. Wu and P. Yu. Range-based bitmap indexing for high cardinality attributes with skew. Technical Report RC 20449, IBM Watson Research Division, Yorktown Heights, New York, May 1996.

[29] Kesheng Wu, Ekow J. Otoo, Arie Shoshani, and Henrik Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.

[30] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 220–230. IEEE Computer Society, 1998.