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] [23, 24]. In this experiment, information about each potentially interesting collision event is recorded. Tens of millions of such events are collected each year, amounting to multiple terabytes of raw data. All raw data go through a preliminary analysis where hundreds of summary attributes are generated for each event. Further analyses are typically only performed on some of the events. One important way of selecting the events is to search for events satisfying some condition on the summary attributes such as "`Energy > 15 GeV and 7 <= NumParticles < 13`" [4, 23]. This type of queries are known as *partial range queries*. One important data management task is to answer these partial range queries efficiently. Since these summary attributes are usually read, not modified, the indexing schemes used for commercial data warehouses should be useful for our task. Based on experiences in data warehouse applications, we know the bitmap index is efficient for partial range queries on relations with many attributes [5, 7, 19, 28]. Since our datasets have hundreds of attributes, bitmap index is even more appropriate [23].

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,

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.

*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.

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

2 and 3. There are four bitmaps, each corresponding to one of the 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 bitwise logical operations are well supported by computer hardware, bitmap indexes are very efficient to use [19]. In many data warehouse applications, bitmap indexes perform better than tree based schemes [5, 19, 28], such as the variants of B-tree [8] or R-tree [10]. According to the performance model proposed by Jürgens and Lenz [13], bitmap indexes are likely to be even more competitive in the future as disk technology improves. In addition to supporting queries on one single table as shown in this paper, researchers have also demonstrated that bitmap indexes can accelerate complex queries involving multiple tables [21]. Realizing the value of the bitmap indexes, most major DBMS vendors have implemented them.

The example shown in Figure 1 is the simplest bitmap index which 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 multiple bitmaps when answering a range query involving the attribute. For attributes with low cardinality, a bitmap index is small compared to tree based indexes and processes range conditions faster as well. To process the example 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`. The final answer is the result of a bitwise logical AND operation on these two bitmaps. The whole process can be carried out efficiently if indexes for `Energy` and `NumParticles` involves only a small number of bitmaps. However, in real applications, especially scientific applications, there are many bitmaps in a bitmap index because the attribute cardinalities are high. In these cases, the bitmap indexes take a lot of space and processing range queries using these indexes take longer than without an index.

Compression is one way to reduce the size of the bitmap index and improve its effectiveness. To compress a bitmap, a simple option is to use one of the text compression algorithms, such as LZ77 (used in gzip) [16]. These algorithms are well-studied and effective in reducing file sizes. However, performing logical operations on the compressed bitmap are usually significantly slower than on the uncompressed bitmap. To address this performance issue, a number of special algorithms have been proposed. Johnson and colleagues have conducted extensive studies on their performances [12, 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, ORACLE [2, 3]. However, even with BBC, in many cases logical operations on the compressed bitmap still can be orders of magnitudes slower than on the uncompressed bitmap.

When processing range queries using BBC compressed bitmap indexes, we observed that more than 90% of the time is spent on performing logical operations. The I/O time is only a small part of the total time. To reduce the total query processing time, we propose a "CPU-friendly" compression scheme. It improves the speed of logical operations by an order of magnitude over BBC at a cost of small increase in space. We call the method the Word-aligned Hybrid (WAH) compression scheme. This scheme 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.

Compression reduces the total size of a bitmap index by reducing the size of each bitmap, another strategy is to reduce the number of bitmaps used, for example, by using binning or more complex encoding schemes. With binning, multiple values are grouped into a single bin and only the bins are indexed [14, 23, 26]. This strategy reduces the number of bitmaps used but it produces precise answers only if range conditions fall on bin boundaries. In order to accurately answer an arbitrary 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 [5, 6, 20, 25, 28]. 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 [20]. This is related to the binary encoding scheme discussed elsewhere [5, 25, 28]. 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 [5], interval encoding [6] and the K-of-N encoding [25]. In all these schemes, an efficient compression scheme may further improve their effectiveness. Additionally, a number of

other common indexing schemes such as the signature file [9, 11, 15] and the bit transposed files [25] may also benefit from efficient bitmap compression schemes.

The remainder of this paper is organized as follows. In Section 2 we review three commonly used compression schemes and identify their key features. These three were selected as representatives in our performance comparisons. Section 3 contains the description of the word-aligned hybrid code (WAH). We discuss the timing results of the bitwise logical operations in Section 4, and the overall query processing performance in Section 5. A short summary is given in Section 6.

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 [12, 27].

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 [16]. 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, 12]. 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 [27, Section 3.2]. This version performs as well as the improved version by Johnson [12]. In both Johnson's tests [12] and ours, the time curves for BBC and gzip (marked at LZ in [12]) 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, 12]. This strategy often uses more bits to represent a fill length than others such as ExpGol [18]. However it allows for faster operations [12].

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 [18] 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 [12].

3 Word-aligned hybrid scheme

Most of the known compression schemes are byte based, that is, they access computer memory one byte at a time. On modern computers, accessing one byte usually takes as much time as accessing one word [22]. To take advantage of this and to minimize the logical operation time, we devised a compression scheme called the *word-aligned hybrid* (WAH) code. The main idea is to simplify the coding scheme so there are only two types of words in the compressed data and to design an alignment requirement so there is no need to extract individual bits or bytes during any logical operation. We have previously considered a number of word-based schemes and this is the most efficient one in our tests [27].

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. The two types of words in WAH are *literal* words and *fill* words,

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

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.

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.

where a literal word represents bitmap literally and a fill word represents a fill. Each word in WAH can be interpreted independently from others. 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. The word-alignment requirement is that fill lengths must be integer multiples of the number of bits in a literal word. If a computer word is 32-bit long, a literal word would represent 31 bits and the fill lengths must be multiple of 31 bits. If a computer word has 64-bit long, each literal word would store 63 bits from the bitmap and each fill would have a multiple of 63 bits.

Figure 2 shows a WAH bit vector representing 128 bits. In this example, we assume each computer word contains 32 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 [27]. 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.

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 [27].

Compared against BBC, the logical operations on WAH compressed bitmaps should be more efficient mainly due to three reasons.

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 implementation of BBC has four different types of runs, other implementations have even more [12]. 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.

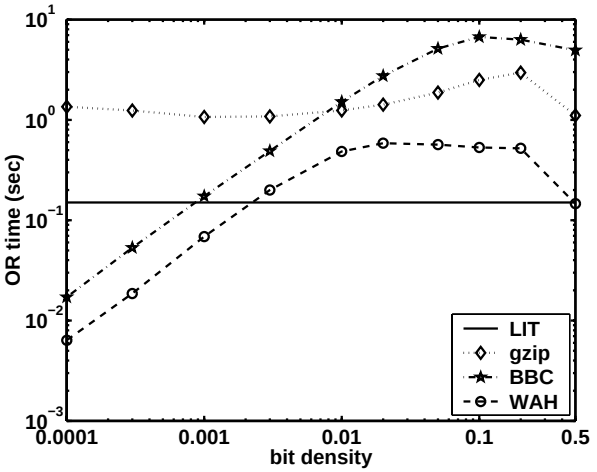
3. BBC can encode shorter fills more compactly than WAH, however, this comes at a cost. Each time BBC encounters a short fill, say a fill with less than 8 bytes, it starts a new run. WAH typically represent such a short fill literally. It is much faster to operate on a WAH literal word than on a BBC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

4 Performance of the logical operations

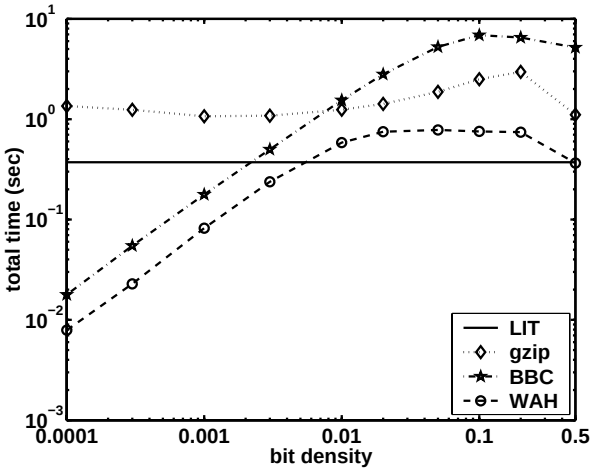
In this section, we discuss the performance of the logical operations. Ultimately we are interested in enhancing the speed of query processing. However, because logical operations are the main operations on the bitmaps and their performances are directly affected by the compression schemes, we discuss 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 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 [23, 24]. 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 [12]. 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.



(a) logical OR time



(b) total time (including IO)

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

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

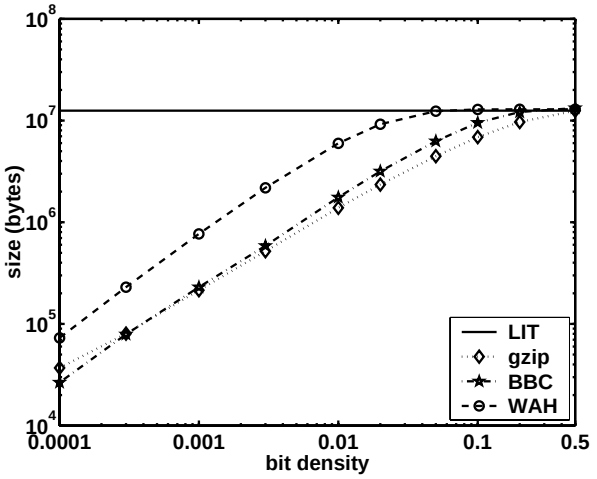
[3] Information about the E450 is available at `http://www.sun.com/servers/workgroup/450`.

cases, the bit vectors simply need to be read into memory and stored in the corresponding in-memory data structures. Only the gzip scheme 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 [27]. 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 [12]. We have chosen the direct method because it requires less memory and is often faster than the alternative methods.

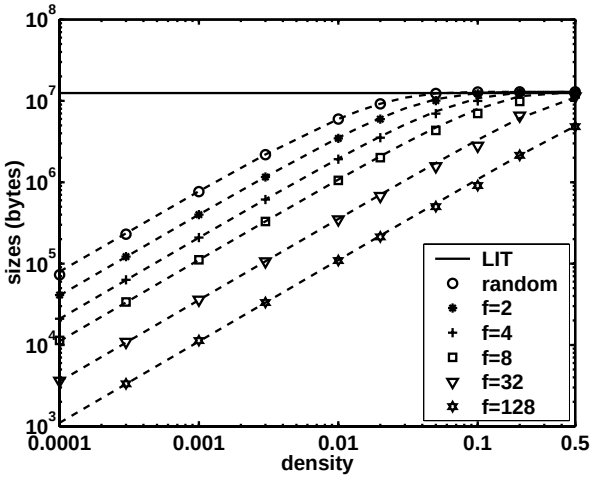
Figure 4 shows the time it takes to perform the bitwise logical OR operations on the random bitmaps. Each data point shows the time to perform a logical operation on two bitmaps with similar bit densities. Figure 4(a) shows the logical operation time and Figure 4(b) shows the total time including the 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.



(a) random



(b) Markov (WAH only)

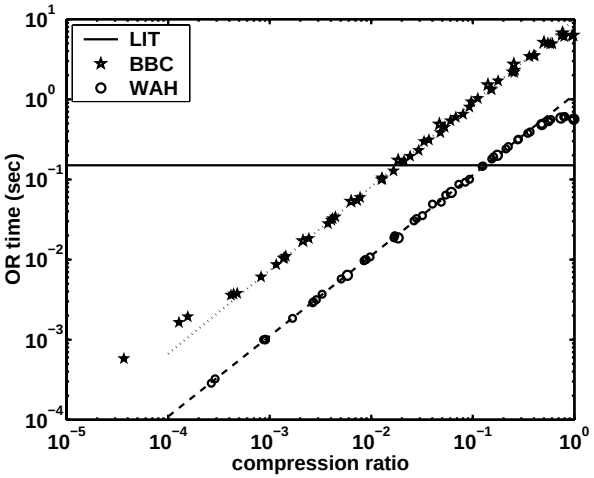
Figure 5. The sizes of the compressed bit vectors. The symbols for the Markov bitmaps are marked with their clustering factors.

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.

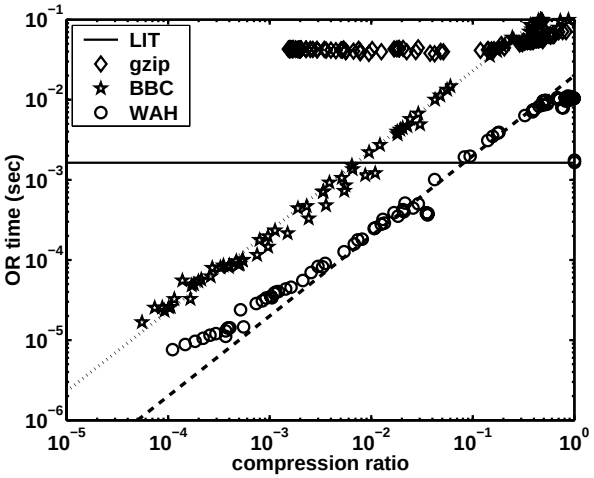
Figure 5 shows the sizes of the four types of bit vectors. Each data point in this figure represents the average size of a number of bitmaps with the same bit density and clustering factor. As the bit density increases from 0.0001 to 0.5, the bit sequences become less compressible and it takes more space 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 symbol 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.

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 reason 3 discussed in the previous section. 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.



(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.

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

Our goal was to develop a compression scheme to reduce the overall query processing time. To see whether we have actually achieved this goal, we measure the query processing time of partial range queries on a set of real application data from STAR. In the previous section, we demonstrated that WAH can perform logical operations much faster than BBC, but WAH also uses more space than BBC. Since query processing involves many operations other than bitwise logical operation, e.g., I/O operation, query parsing and locking, we need to make sure the decrease in logical operation time is not offset by the increase in I/O time caused by the increase index size. In addition to testing the performance of our own implementations of compressed and uncompressed bitmap indexes, we also tested ORACLE's BBC compressed bitmap index and the projection scan. The *projection scan* is a scheme of performing comparisons on the attribute values where each attribute is stored separately. It is also know as the projection index [20].

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 physicists. 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 total sizes of the bitmap indexes. Four columns are displayed in each table. Columns marked 'ours' are our own implementation of the compressed bitmap indexes based on WAH and BBC compression schemes. The columns marked 'ORACLE' show the sizes of the two kinds of indexes available, the bitmap index and the B-tree index. The bitmap index is marked with the compression scheme 'BBC'. Conceptually, ORACLE's BBC compressed index is equivalent to our BBC compressed bitmap index.

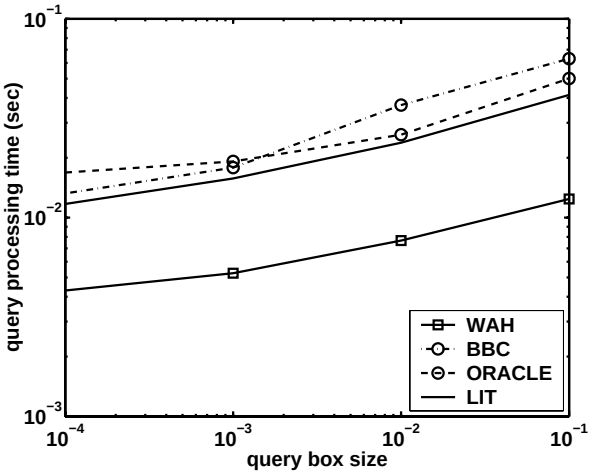
	ours		ORACLE	
N	WAH	BBC	BBC	B-tree
12 low cardinality attributes				
312	7	4	7	370
12 most frequently queried attributes				
2,673,646	186	117	111	408

Figure 7. Sizes (MB) of the bitmap indexes. "N" indicates the number of bitmaps in the bitmap indexes.

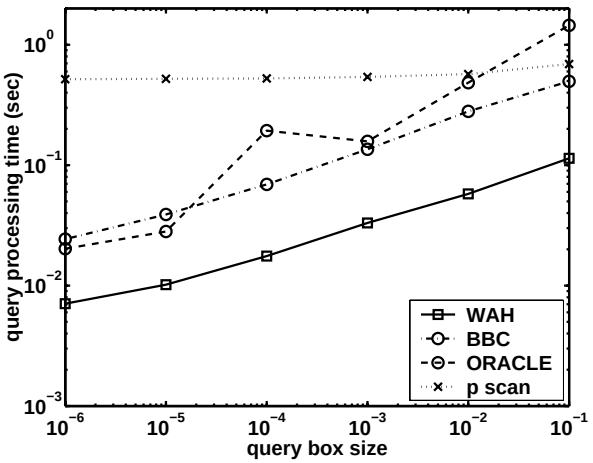
In the first data set, there are a total of 312 distinct values, i.e., there are 312 bitmaps in all bitmap indexes. Without compression, 312 bitmaps use about 84 MB. All three versions of the compressed bitmap indexes are less than 10% of this size and are less than 7% of the data size. In the second data set, there are nearly 2.7 million distinct values. Without compression, the bitmap index size would be more than 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. For both datasets, the compressed bitmap indexes are significantly smaller than the B-tree indexes.

Figure 8 shows the average query processing time of three compressed bitmap indexes. In this figure, the performance of ORACLE's BBC compressed indexes is marked as ORACLE. The partial range queries are generated by randomly selecting two 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 and the total volume of the attributes [17]. 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.

Since the projection scan, "p scan" in Figure 8, only access the attributes involved in a query, it is quite faster [20]. For example, on our test machine, ORACLE 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 ac-



(a) on 12 low cardinality attributes



(b) on 12 most frequently queried attributes

Figure 8. The average query processing time of random range queries on the STAR data. Each query contains range conditions on two attributes.

tually stored all 500 attributes in the table, ORACLE would take nearly 5 minutes to perform its scan operation. 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. This is an efficient strategy since the projection scan time is always close to 0.56 seconds in Figure 8.

Our tests indicate that the bitmap indexes are efficient for both low cardinality attributes and high cardinality attributes. On the low cardinality attributes, WAH compressed indexes are not only smaller but are also faster. In contrast, BBC compressed indexes are slower than the uncompressed ones. On high cardinality attributes, BBC compressed indexes require about as much time as the projection indexes, but WAH compressed indexes are always faster. The relative differences between WAH compressed indexes and BBC compressed indexes are larger on high cardinality attributes than on lower cardinality attributes. On high cardinality attributes, the average query processing time using the ORACLE bitmap indexes is nearly 10 times longer than using the WAH compressed bitmap indexes.

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. 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 operations on compressed bitmaps can be 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.

Since the total query processing time includes both I/O time and logical operation time, we need to verify that the decrease in logical operation time is not offset by the increase in I/O time due to the 60% increase in index size. Our tests on a set of real application data show that WAH compressed indexes can indeed reduce the overall query processing time. Compared to uncompressed indexes, WAH compressed indexes are not only smaller but also take less time to answer partial range queries. Compared to the indexes compressed with BBC, the WAH compressed indexes can be 10 times faster.

References

[1] Sihem Amer-Yahia and Theodore 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] 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.

[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 wharehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, March 1997.

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

[9] K. Furuse, K. Asada, and A. Iizawa. Implementation and performance evaluation of compressed bit-sliced signature files. In *Proceedings of CISMOD'95*, pages 164–177. Springer, 1995.

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

[11] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evalution of signature files as set access facilities in OODBs. In *Proceedings of SIGMOD 1993*, pages 247–256. ACM Press, 1993.

[12] T. Johnson. Performance measurements of compressed bitmap indices. In *Proceedings of VLDB'99*, pages 278–289. Morgan Kaufmann, 1999. A longer version appeared as AT&T report number AMERICA112.

[13] M. Jürgens and H.-J. Lenz. Tree based indexes vs. bitmap indexes - a performance study. In *Proceedings of DMDW'99*, 1999.

[14] Nick Koudas. Space efficient bitmap indexing. In *Proceedings of CIKM 2000*, pages 194–201. ACM, 2000.

[15] 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.

[16] 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`.

[17] V. Markl and R. Bayer. Processing relational OLAP queries with UB-trees and multidimensional hierarchical clustering. In *Proceedings of DMDW 2000*, 2000.

[18] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *Proceedings of ACM-SIGIR 1992*, pages 274–285. ACM Press, 1992.

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

[20] P. O'Neil and D. Quass. Improved query performance with variant indices. In *Proceedings of SIGMOD 1997*, pages 38–49. ACM Press, 1997.

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

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

[23] 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 1999*, pages 214–225. IEEE Computer Society, 1999.

[24] K. Stockinger, D. Duellmann, W. Hoschek, and E. Schikuta. Improving the performance of high-energy physics analysis through bitmap indices. In *Proceedings of DEXA 2000*, September 2000.

[25] 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.

[26] 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.

[27] 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.

[28] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *Proceedings of ICDE 1998*, pages 220–230. IEEE Computer Society, 1998.