Compressing Bitmap Indexes for Faster Search Operations

Kesheng Wu Ekow J. Otoo Arie Shoshani

January 18, 2002

Abstract

In this paper, we study the effectiveness 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

Bitmap indexes have been discussed extensively in the literature because of their usefulness in various database applications such as data warehouses[4, 6, 17, 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 **R** of a tiny table (**T**) consisting of only eight tuples (rows). The attribute **R** can have one of four values, A, B, H and W, representing Races: Asian, Black, Hispanic and White. There are four bitmaps each representing whether the value of **R** is one of the four choices. For convenience, we have labeled the four bit sequences $b_1, \ldots, b_4$. To process a query such as "`select * from T where R=A or R=B`", 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 [17]. In many data warehouse applications [4, 17, 25], bitmap indexes are better than the tree based schemes, such as the B-tree. According to the performance model proposed by Jürgens and Lenz [11], 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 they can accelerate complex queries involving multiple tables [19]. Realizing the value of the bitmap indexes, most major DBMS vendors have implemented them in their products.

The example shown in Figure 1 is the simplest form of the bitmap index which we will call the traditional bitmap index. It may use more space than what is practical, especially for attributes with high cardinalities, i.e., attributes with a large number of distinct values. One solution proposed is to use a more complex encoding scheme for generating the bitmaps. One well-known scheme is the bit-sliced index [18], that encodes n distinct values using $log_2 n$ bits and creates a bitmap for each binary digit of the value indexed. This is related to the binary encoding scheme discussed

OID	**R**	bitmap index			
		=A	=B	=H	=W
1	A	1	0	0	0
2	B	0	1	0	0
3	W	0	0	0	1
4	H	0	0	1	0
5	W	0	0	0	1
6	W	0	0	0	1
7	B	0	1	0	0
8	W	0	0	0	1
		b_1	b_2	b_3	b_4

Figure 1: A sample bitmap index for attribute **R**.

elsewhere [4, 12, 23, 25]. 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 during query processing. Examples of these are the attribute value decomposition [4], interval encoding [5] and the K-of-N encoding [23]. In this paper, we explore a strategy for reducing the index size by compression. Since compression can be applied on any bitmap, an efficient compression scheme should benefit all bitmap indexes no matter what encoding scheme is used. In this paper, we study the effect of compression on the traditional bitmap index. Since we no longer discuss other schemes, we simply refer to the traditional bitmap index as the bitmap index.

One simple option to compress the bitmaps is to use one of the text compression algorithms, such as LZ77 (used in gzip) [14]. 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 performance [10, 1]. From their studies, we know that the logical operations using these specialized schemes are in general faster than those using gzip. One of such specialized algorithm, named 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, logical operations on the compressed data still can be orders of magnitudes slower than on the uncompressed data in many cases.

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. We demonstrate that this algorithm not only supports faster logical operations but also enables the bitmap index to be applied to attributes with high cardinalities. In general, the bitmap indexes are said to be effective for attributes with low cardinalities, say, < 100. On scientific data where most of the attributes are of high cardinality, the bitmap index may not be effective. Our tests show that by using WAH compression, we can achieve good performance even on scientific data sets.

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 longer the need to switch compression schemes in a bitmap indexing software. In short, the new compression scheme not only improves the performance of the bitmap indexes but also simplifies the indexing software. Additionally, a number of other common indexing schemes such as the signature file [7, 9, 13] and the bit transposed files [23] may also benefit from this efficient compression algorithm.

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). Section 4 contains some timing results of the bitwise logical operations. Some timing information on processing range queries are presented 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 [10, 24].

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*[1]. 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 schemes such as gzip [14]. 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, 10]. 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 [24, Section 3.2]. This version performs as well as the improved version by Johnson [10]. In both Johnson's tests [10] and ours, the time curves for BBC and gzip (marked at LZ in [10]) 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*)

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

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 any fill length. 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 contain 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, 10]. This strategy often uses more bits to represent a fill length than others such as ExpGol [16]. However it allows for faster operations [10].

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, is 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 requirement may lead to better compression. For example, the ExpGol scheme [16] 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 [10].

3 Word based schemes

Most of the known compression schemes are byte based, that is, they access computer memory one byte at a time. On most modern computers, accessing one byte takes as much time as accessing one word [20]. 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 [24].

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 multiple of 31 bits. If the machine has 64-bit words, each

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.

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 multiple 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 [24]. 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 to 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 [24].

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 in this section and leave the performance of query processing for the next section.

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 specified bit density and

clustering factor. In our tests, 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 for high bit density cases should be the same as low bit densities ones. 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[2] [21, 22]. 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 [10]. 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 not on most other parameters such as cache size, memory size, memory bandwidth, the number of instructions issued per clock cycle, and so on. 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 case in memory. The 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.

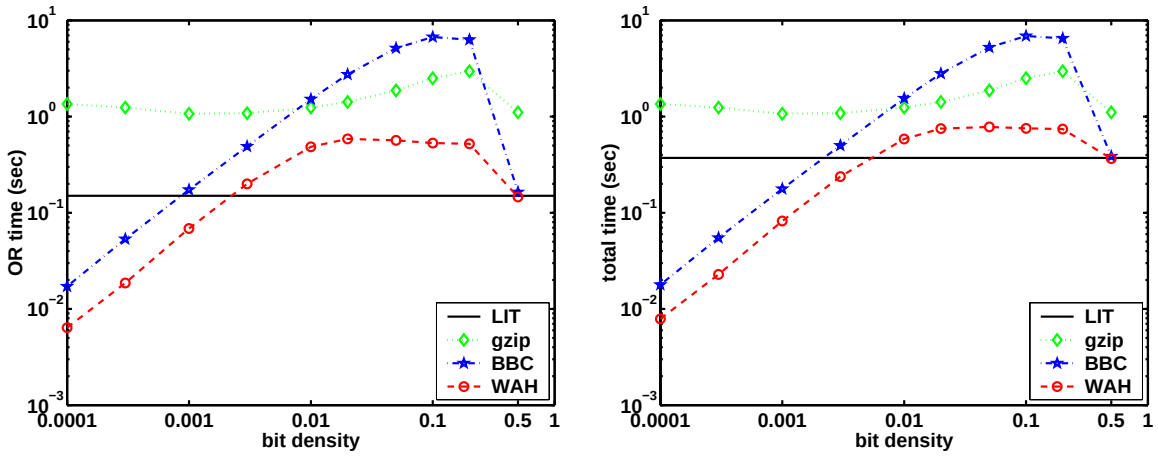
Because of space limitations, we only show performance of the logical OR operations in the

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

following graphs. 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 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 [24]. 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 [10]. 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



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

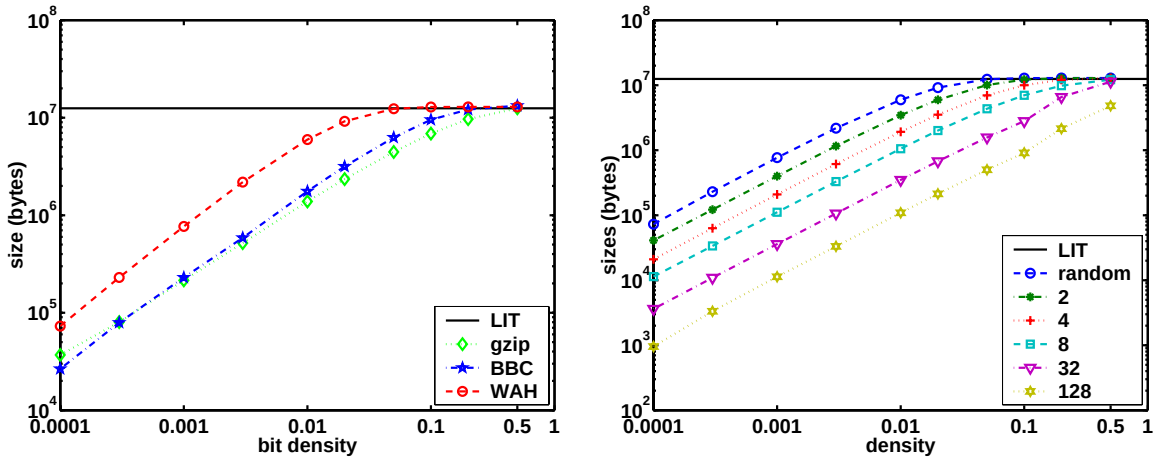
Figure 4: CPU seconds needed to perform a bitwise OR operation on two random bitmaps.

time, we will not include the IO time.

Among the schemes shown, it is clear that WAH uses much less time than either the BBC or the gzip schemes. 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 any compression scheme. In these cases, our implementation of BBC and WAH are able to perform nearly as fast as the literal scheme by not compressing the results. In Figure 4, the lines for BBC and WAH fall on top of the one for the literal scheme at bit density of 1/2.

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

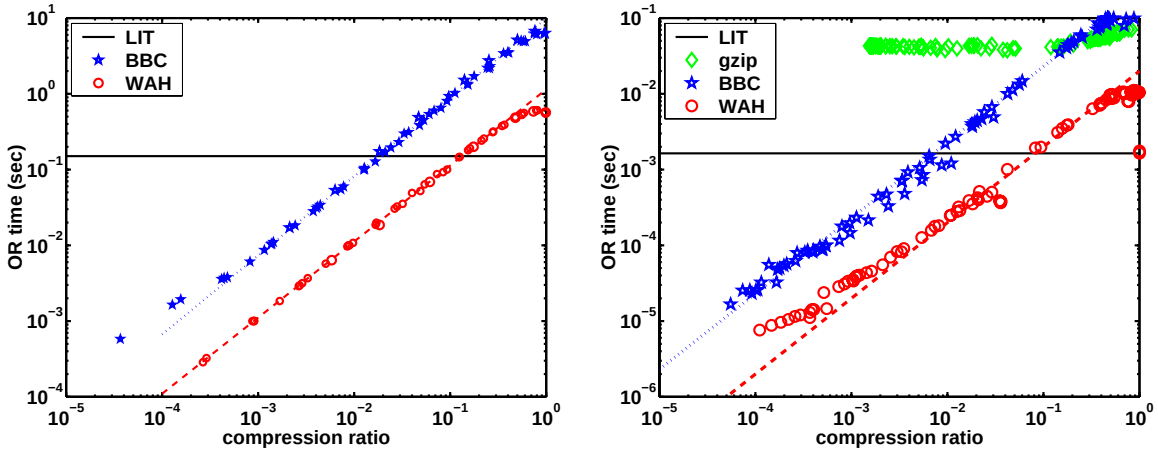


(a) random (b) Markov (WAH only)

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

determine the logical operation speed.

Figure 5 shows the sizes of the four types of bit vectors. Each data points in this figure represent 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 WBC 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 compressible ones, the differences in total



(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.

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.

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 implementation of BBC has four different types of runs, other implementations have even more [10]. It may take up to three tests in order to decide which run type a header byte is. After deciding the run type, many clock cycles may also be needed to fully decode a run.

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 the 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, the BBC scheme 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

This research was originally motivated by the need to manage the volume of data produce by the STAR experiment. The frequently queried attributes, the tags, 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 *ad hoc range queries*, are particularly difficult for most database systems. For example, if a B-tree index is created for each attribute, ORACLE 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 [8] 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 data 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 ad hoc range queries [11, 17, 18]. The projection index is simply another name for vertical partitioning a relational table where one stores the values of an

attribute consecutively rather than storing the values of a tuple consecutively. In this case, queries are processed by performing comparisons on the values. In later discussions, we will refer to this as the *projection scan*.

Our goal is to demonstrate that WAH compression scheme 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 and the frequently queried attributes are mostly four-byte integers and floating-point values along, but one has eight-byte floating-point values. 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 'ORACLE' shows the sizes of the bitmap indexes in ORACLE. Since ORACLE implements a BBC compressed bitmap index, conceptually it is equivalent to our BBC compressed bitmap index.

c	WAH	BBC	ORACLE
4	10196	8733	335037
4	305296	164665	421074
18	1510740	924035	1077269
19	1437892	842359	1001476
24	1703456	975465	1127116
25	1729380	988060	1140852
33	33568	9516	334420
35	151808	39254	349970
35	151708	39222	349771
35	151808	39257	349797
40	1964	1534	330128
40	1972	1599	329785
total			
312	7189788	4033699	7146695

(1) 12 low cardinality attributes

c	WAH	BBC	ORACLE
40	1964	1534	340946
40	1972	1599	340573
116	10339232	3393224	3473910
367	10585524	3164756	3572127
371	23436	16622	350916
1688	11855904	3858185	4271522
1807	16182848	4922029	5414222
3786	10973128	3827861	4122542
76920	19849220	8874753	8642620
514516	20807036	18059791	15606417
818300	33036432	28014187	25763032
1255695	52427916	43689012	39122608
total			
2673646	186084612	117823553	111021435

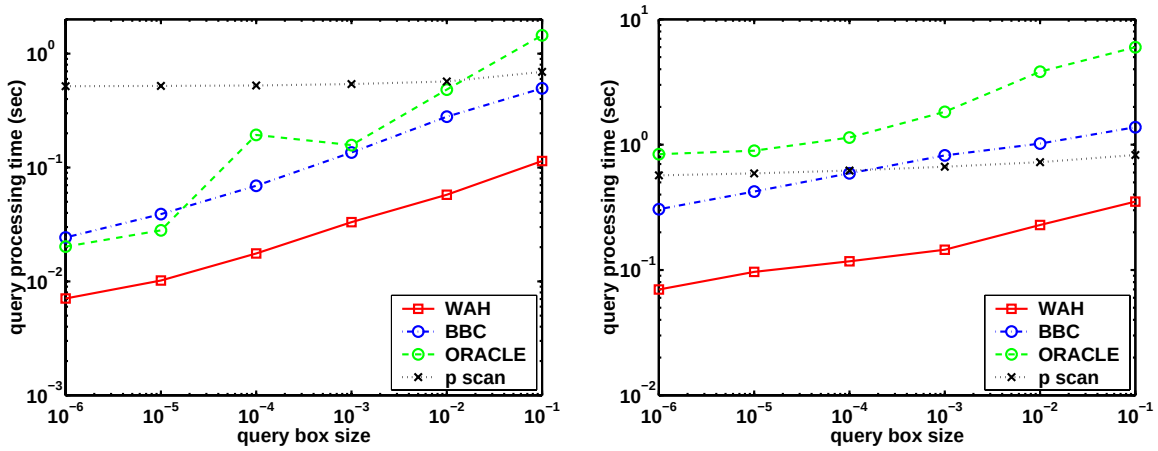
(2) 12 most commonly queried attributes

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

In the first data set, there are a total of 312 distinct values, i.e., there are 312 bitmaps in the 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 more than 2.6 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 because the majority of the bitmaps are very sparse. The total size of each set of the compressed bitmap indexes is about the same size as the data. These sizes are comparable to those used B-tree and others.

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 ad hoc range queries are generated by randomly selecting some attributes and constructing a query with the specified



(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.

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 [15]. 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.

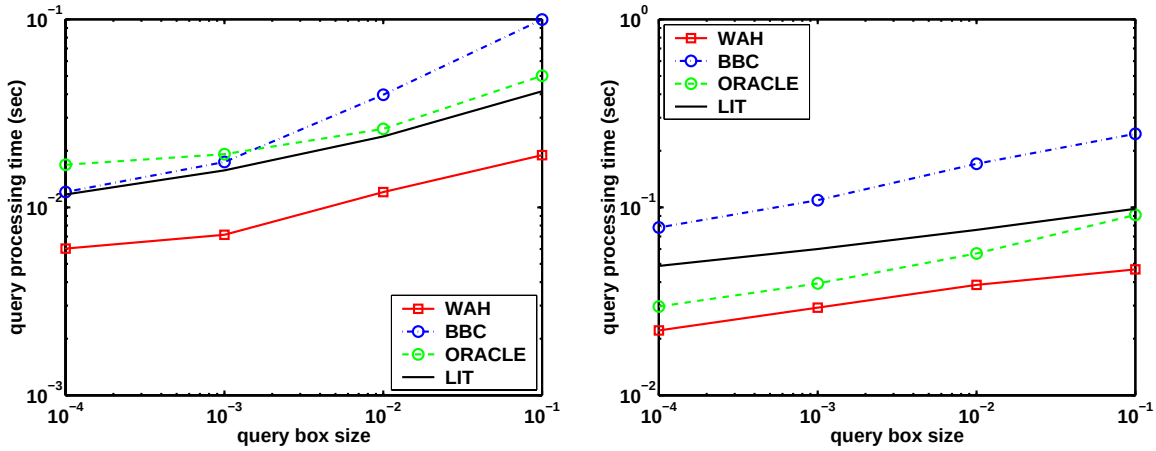
We also show the time used by the projection index, marked as 'p scan', short for the projection scan, in Figure 8. The projection index only access the attributes involved in a query and is much faster than most indexing strategies [18]. For example, on our test machine, ORACLE takes about 6.5 seconds to scan a table with 12 attributes while the projection scan only need 0.56 ($\approx 6.5/12$) seconds. Had we actually stored all 500 attributes in the table, ORACLE would take nearly 5 minutes to perform its scan operation. Clearly, the projection scan is fast. We also take full

advantage of the fast bitmap data structure implemented to store the intermediate result. When evaluating conjunctive queries, the result of the left side can be used as the mask to limit the amount work during the evaluation of the right side. A sophisticated execution planner like the one in ORACLE can 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 ORACLE. 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 ORACLE. 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, which can amount to a few milliseconds, become more important after we have significantly reduced the logical operation time.

Comparing ORACLE's implementation of BBC with our own, we found that ORACLE's implementation performs slower than ours. This is clearly evident when a large number of logical



(a) 2 attributes per query (b) 5 attributes per query

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

operations are needed as in the case of processing range 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 Oracle version) take about the same amount of time when there are two attributes in a query. However, ORACLE takes less time than the stand-alone version when there are five attributes in a query. This is because ORACLE 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 query planning functionality clearly is important to have. However, by the fact that the two versions of BBC compressed indexes use about the

same amount of time when there are two attributes per query, we can infer that the BBC in the stand-alone version is at least as efficient as the one in ORACLE.

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 need half the time used by the uncompressed indexes to process the same queries.

6 Summary

It is well accepted that I/O dominates the operational efficiency of out-of-core indexing methods. Thus, most indexing techniques focus on minimizing I/O. For bitmap indexing, this assumption is incorrect. Our test results show that the computation time dominates the total time. In addition, as main memories become cheaper, we expect that "popular" bitmaps remain in memory once there are used. For these reasons, we pursued the course of improving the computational efficiency of operations over bitmaps. Specifically, we were interested in compression schemes that are able to support fast bitwise logical operations. 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 show that our WAH scheme performs exceptionally well for skewed data, i.e. when the value distribution on each attribute is skewed. In this case bitmaps are either very sparse or very full and they compress well. For low skewed data (i.e. close to random data) compression is ineffective, and the uncompressed bitmap scheme performs better, but not much better than WAH.

Since most interesting data is skewed, using WAH in all cases is a good choice. However, if one chooses to leave the data uncompressed when compression is ineffective, then we recommend using the uncompressed version of WAH (effectively wasting one bit of a 32 bit word) since it will work smoothly and efficiently with other WAH compressed bitmaps. This approach permits the use of WAH with mixed skewness of the attribute value distributions.

We also demonstrated from our tests that improving the compression scheme actually improve the query answering speed, not only logical operations. Tests show that WAH compressed indexes are not only smaller than the uncompressed indexes they are also more efficient in answering queries. Due to various overheads involved in query processing (parsing, locking, network communication) the improvement in query processing is about a factor of four or five rather than 12 for logical operations only. Still, we believe that even for relational database implementations which use bitmap indexes, it is worthwhile to use WAH instead of BBC.

The bitmap index is often thought to be effective only on low cardinality attributes. By using WAH, we also 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] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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`.

[15] 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.

[16] 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.

[17] 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*, September 1987.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.