A Performance Comparison of bitmap indexes

Kesheng Wu   Ekow J. Otoo   Arie Shoshani

Abstract

Many data warehousing systems and database management systems make extensive use of bitmap indexing schemes. To ensure optimal performance, these bitmaps should be efficient in both memory usage and computation time. In this paper, we perform experimental comparisons of various compression schemes for bitmap indexes to identify which ones are most efficient in a large scientific data management system. Because bitwise logical operations are the most common operations on these bitmaps, our experiment measures the performance of the bitwise logical operations between the compressed bitmaps. Generic compression schemes, such as gzip, that are efficient in reducing the space requirement cannot be subjected to logical operations without explicit decompression. This is costly in computational time and memory usage. It is generally more efficient to use schemes that can perform logical operations directly on the compressed bitmaps. To further ensure good performance, some compression schemes have been designed to be byte-aligned. A well-known example of such a scheme is the Byte-aligned Bitmap Code (BBC). Since most current computers operate on words just as fast as on bytes, we also explored the performance of two word-aligned schemes in comparison with BBC. Such word-aligned schemes have not been previously studied. From the extensive experiments we carried out, we found that the word-aligned schemes significantly outperform BBC with only a modest increase in storage. On both synthetic data and real data from scientific experiments, the word-aligned schemes use an average of 50% more space, but are about 12 times faster in performing logical operations on the compressed bitmaps.

1 Introduction

Bitmap based indexing schemes of various kinds have attracted considerable research and commercial interests recently [3, 8, 15]. They are found to be effective in reducing processing time of complex queries on slowly changing datasets, such as those in data warehousing and decision support systems. Commercial database systems, e.g., Oracle 8, IBM DB2, and Sybase IQ, now implement some forms of a bitmap index scheme. The bitmap index has also been effectively used for indexing very large scientific datasets [11, 12]. Efficient representation of these bitmap indexes are important to ensure the theoretical advantages are realized in practice. This paper summarizes our experiences of search for an efficient data structure for representing these bitmap indexes.

To show how bitmap indexes are generated and used, we give one example [3, 4, 15]. Figure 1 shows two sets of bitmap indexes each for an attribute of a tiny database consisting

i	$\mathbf{R}_i$	bitmap index			
		A	B	H	W
1	W	0	0	0	1
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

i	$\mathbf{X}_i$	bitmap index			
		< 1	[1,3]	[4,6]	> 6
1	1	0	1	0	0
2	4	0	0	1	0
3	7	0	0	0	1
4	6	0	0	1	0
5	0	1	0	0	0
6	6	0	0	1	0
7	0	1	0	0	0
8	4	0	0	1	0
		b_5	b_6	b_7	b_8

Figure 1: Two sample bitmap indexes for two attributes **R**, on the left, and **X**, on the right.

of only two attributes and eight tuples (rows). On the left, the attribute **R** has categorical values, and each bit sequence of the bitmap index denotes whether $\mathbf{R}_i$ is one of the four categories. For example, the 4th value of **R** is H, the 4th bit of the bit sequence representing $\mathbf{R} = H$ is 1, the 4th bit of the other three sequences are 0. On the right, attribute **X** contains integer values that are divided into four ranges (also called bins). In this case, each bit sequence is associated with a bin and represents whether a value is in the bin. To answer a range query, some bit sequences are combined together using bitwise logical operations. For convenience, we have labeled the eight bit sequences $b_1, \ldots, b_8$. To answer query "($\mathbf{R} = B$) AND ($\mathbf{X} < 4$)", one performs the logical operation (b_2 AND (b_5 OR b_6)). As demonstrated in this example, the bitmap index is well suited for categorical values that are common in data warehousing environment [3, 8, 15].

For large datasets with hundreds of attributes and millions of rows in the dataset [11, 12], it is easy to generate thousands of bit sequences each with millions of bits. Typically, these bit sequences are stored on disk and are compressed to reduce storage requirement. We call data structure representing these compressed bit sequence *bit vectors*. The logical operations are the primary operations on the bit sequences from the bitmap indexes. The speed of logical operations on these bit vectors is crucial to the overall efficiency of query processing. The goal of this paper is to search for a compressed bit vector scheme that supports fast bitwise logical operation.

Bit sequences are so common in computer programs that the Standard Template Library (STL) of C++ language supports two schemes to represent them, namely `vector<bool>` and `bitset` [13]. The container type `vector<bool>` is more suitable for storing long bit sequences. The STL standard does not require the data to be compressed and does not provide any bitwise logical operations. For these reasons, specialized data structures are needed. One possibility is to use a general purpose compression tool such as gzip [6] or bzip2 [10]. These schemes are effective in reducing the file sizes. Bit vectors compressed this way have to be explicitly decompressed before the logical operations. The decompression process may be done one segment at a time to limit the memory required for the operands of a logical operation. The difficult choice is in dealing with the result of the operation. One

either stores the result without compression at a significant cost in computer memory or compresses it at a significant cost in time.

To limit the memory usage, the ideal scheme should perform *direct* logical operations on the compressed bit vectors and produce compressed result. In an earlier paper [5], the Byte-aligned Bitmap Code (BBC) was identified to be close to this ideal case. It is generally faster than other compressed schemes and it compresses quite well too. However, BBC only outperforms other schemes in some cases. In a subsequent study, the author proposed a complex dynamic scheme for switching among a number of schemes [1]. The difficulty of using such a strategy is that one either has to duplicate all bit sequences in different formats or has to convert to and from different forms on the fly. In both above mentioned studies [1, 5], the authors have only considered byte based schemes, i.e., these schemes access memory one byte at a time. Since on most computers accessing one word takes about the same amount of time as accessing a byte [9], a word based scheme may potentially allow faster logical operations. The disadvantage is that a word based scheme might take more space than a byte based scheme. Our hope is to find a word based scheme that compresses reasonably well but performs logical operation at clearly faster speed. Through extensive testing, we have identified two word based schemes that greatly outperform BBC yet use only slightly more storage.

The remainder of this paper is organized as follows. In Section 2 we review three commonly used bit vectors schemes and identify the key features of BBC. Section 3 contains the description of the two word based schemes. Section 4 contains performance tests on synthetic data and analyses of the performance characteristics. We confirm the relative performance differences using real application data 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 bit sequences. We also use this chance to introduce the terminology used to described the later schemes.

A straightforward way of representing a bit sequence is to use one bit of computer memory to represent one bit of the sequence. We call this the *literal* (LIT) representation. It does not compress bit sequences, and logical operations on literal bit vectors are extremely simple and fast. The ideal bit vector scheme should be at least as fast as the literal scheme but use much less memory.

The second type of schemes in our comparisons is the general purpose compression scheme such as gzip and bzip2. They are highly effective in compressing data files. We use gzip to represent these schemes because it is usually faster than bzip2 in decompressing the data. In our tests involving gzip, only the operands of a logical operation are compressed; the results are not. This is to save time. Had we compressed the result as well, the measured times would be significantly longer.

There are a number of compression schemes that offer good compression and allow fast bitwise logical operations as mentioned earlier. One of the best known schemes is the Byte-aligned Bitmap Code (BBC) [2, 5]. The BBC scheme can perform bitwise logical operations very efficiently. In addition, it compresses almost as well as gzip. We use BBC as the representative of the byte based compression schemes.

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 *fills*) by its bit value and its 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. A fill can be represented by its length and the fill bit. General purpose compression schemes try to store repeating bit patterns in a compact way. The run-length encoding is among the simplest of these compression schemes. For this reason, these specialized bitmap compression schemes can support faster logical operations.

There are many ways of using the run-length encoding idea to compress a bit sequence. The naive implementation of using a word to represent all fill lengths is ineffective because it uses more space to represent short fills than in the literal scheme. One common improvement is to represent 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, the BBC scheme represents fill length as number of bytes rather than number of bits. All these characteristics contribute to its effectiveness.

A property that is crucial to the efficiency of the BBC scheme is the byte alignment. The essence of this property is that a fill size is limited to be an integer multiple of a literal byte. This ensures that during any bitwise logical operation it never breaks up a literal byte to extract individual bits. Removing the alignment requirement can lead to better compression. For example, the ExpGol scheme [7] can compress better than BBC partly because it does not have byte alignment. However, bitwise logical operations on ExpGol bit vectors are usually much slower than that of BBC [5]. Bitwise operations between two words or two bytes are generally supported by the computing hardware and each operation takes effectively one clock cycle. The operation of breaking up a byte to extract some bits involves shifting and masking that usually takes a number of clock cycles. Additional time is needed to perform bitwise logical operations and to assemble the bits into bytes or words. Altogether, a bitwise logical operation that needs to access individual literal bits takes considerably longer than one that only accesses whole bytes or words. For this reason, our word based schemes should be word-aligned.

3 Word based schemes

Most of the known compressed 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 [9]. 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 BBC bit vector 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 two special schemes that access only whole words. The two new schemes are named the word-aligned hybrid run-length code (WAH) and the word-aligned bitmap code (WBC).

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 00000004

Figure 2: A WAH bit vector. Each code word (bottom) represents a multiple of 31 bits from the bit sequence shown on the top except the last two words that represent the four leftover bits.

3.1 Word-aligned hybrid run-length code (WAH)

This is based on the *hybrid run-length code* (HRL) that represents long fills using run-length encoding and represents short fills literally. There are two types of code words in HRL: *literal* and *fill*. In our current 32-bit implementation, we use the Leftmost Bit (LMB) of a word to distinguish between a literal word (0) and a fill word (1). The lower 31 bits of a literal word contains the literal bit values of the bit sequence. The second leftmost bit of a fill word is the fill bit and the 30 lower bits store the fill length. The *word-aligned hybrid run-length code* (WAH) imposes the word-alignment requirement on the fills, which requires all fill lengths to be integer multiples of 31 bits (i.e., literal word size). We also represent fill lengths in multiples of literal word size. For example, if a fill contains 62 bits, the fill length will be recorded as two (2).

Figure 2 shows a WAH bit vector representing a bit sequence that is an extension of b_6 in Figure 1. The second line shows how the bit sequence is divided into 31-bit groups and the third line shows the hexadecimal representation of the grouping. The last line shows the values of the words used in WAH coding. The first three words are normal code words, two literal words and one fill word. The fill word 80000002 indicates a fill of two-word long containing 62 consecutive zero bits. The forth word is the *active word* and the last one counts the number of useful bits in the active word. Among the three code words, two are literal words, only the second one is a fill word. In this case, the three normal code words represent 124 bits and the active word contains the last 4 bits.

3.2 Word-aligned bitmap code (WBC)

This scheme is designed to mimic the behavior of the BBC scheme. In this case, we first group bits of a bit sequence into words, then group words into runs. A run contains a fill followed by a number of literal words called a tail. On a 32-bit machine, a literal WBC word contains 32 bits from the sequence it represents. All fill lengths must be multiples of 32 bits. A header word is used for each run. It contains three pieces of information, the fill bit, the fill length and the tail length. Both the fill length and the tail length are measured in the number of words. In our current 32-bit implementation, we use the rightmost 16 bits to store the tail length, the LMB to store the fill bit and the remaining 15 bits to store the fill length.

Figure 3 shows how this scheme represents the same bit sequence shown in Figure 2. The first line of this figure shows a hexadecimal representation of the same 128 bits grouped into

128 bits	80000700	00000000 00000000 01FFFFFF	
WBC (hex)	00000001 80000700	00020001 01FFFFFF	00000000 00000000

Figure 3: A WBC bit vector representing the same bit sequence as in Figure 2.

32 bits each corresponding a literal representation in 32-bit words. This sequence is divided into two runs. Run 1 includes only the first word and run 2 includes the last three words. The header word (00000001) of run 1 indicates that it contains no fill words and one literal word; the header word (00020001) of run 2 indicates that it contains a 0-fill of two words long followed by one literal word. The last two words are again the active word and its associated counter. In this case, both of them are zero indicating that there is no useful bits in the active word.

4 Performance on synthetic data

In this section, we present some timing results on synthetic data. In theory, we know what to expect, the goal here is to how well the word-aligned schemes meet our expectation. This section is organized into three subsections. The first subsection contains how the synthetic test data is generated and how the programs are timed. The second subsection contains some representative timing results. The last subsection contains our analyses of the performance characteristics.

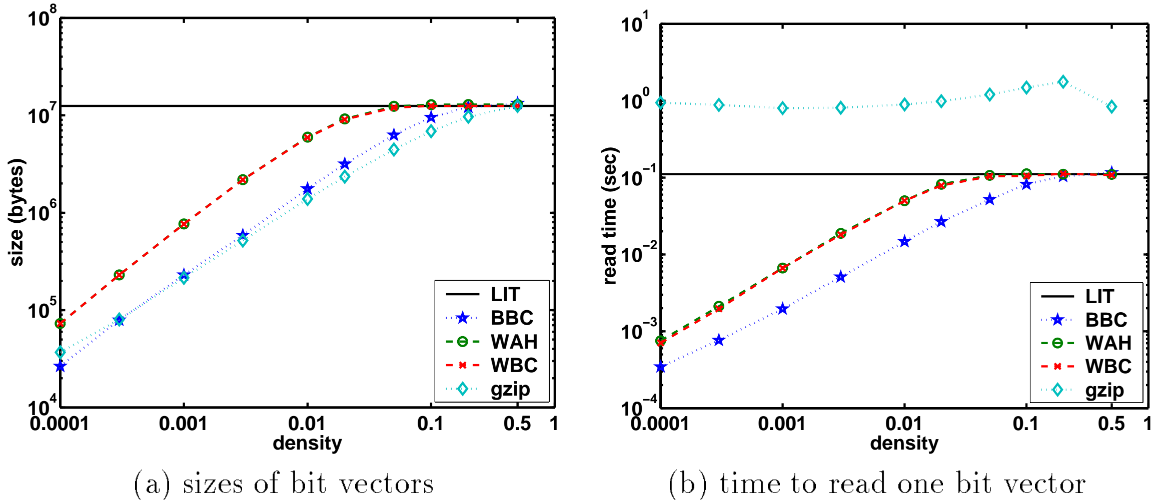
4.1 Experiment setup

The design goal of synthetic data is to reveal typical performance characteristics of the different bit vector schemes. For simplicity we will use time as the measure of logical operation performance. We have conducted a number of tests on different machines. To our surprise, the relative performance among the different schemes is independent of the specific machine architecture. In this paper, we only report the timing results from a Sun Enterprise 450 that is based 400 MHz UltraSPARC II CPUs [1]. The test data were stored in files residing on a file system consisting of five disks connected to the internal UltraSCSI controller and managed by a VERITAS Volume Manager[2]. The VERITAS software distribute files onto the five disks to maximize the SCSI controller. The machine has four gigabytes (GB) of RAM which is large enough to store each of our test cases 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. All test bit sequences used in this section contain 100 million bits. To limit space, we will only show performance of the logical OR operations. Other logical operations have similar relative performance characteristics.

Our synthetic data were generated based on two characteristics of bit sequences, namely the *bit density* and the *clustering factor*. We define the *bit density* as the fraction of bits in

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

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



(a) sizes of bit vectors (b) time to read one bit vector

Figure 4: Sizes and time needed to read bit sequences of the random dataset.

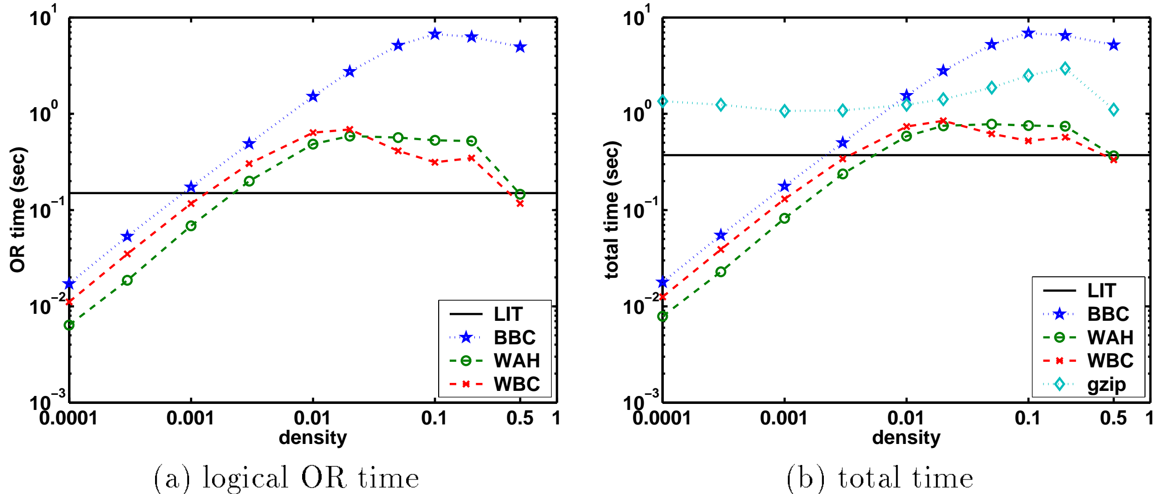
a sequence that have the value one. The *clustering factor* is defined as the average number of bits in 1-fills. The first group of test data are generated to have specified bit density. This group consists of 20 bit sequences with ten different bit densities ranging from 0.0001 to 0.5. If the bit density is d, each bit of these bit sequence has a probability d of being one. We don't test bit sequences with higher density than 0.5 because all the algorithms tested should behave the same if every bit of a bit sequence is turned into its complement. This set of test data is referred to as the *random dataset*. The performance results shown in the next subsection are produced with this dataset.

The second test dataset is produced by a simple Markov process. We use this process to generate bit sequences of specified bit density and clustering factor (denoted by c). Our Markov dataset consists of 100 bit sequences with ten different bit densities and five different clustering factors (2, 4, 8, 32, 128). The clustering factors of the bit sequences from the random dataset are close to one. With increased clustering factors, most schemes can compress better.

4.2 Some performance data

In this subsection, we show the performance data on the random dataset. The goal is to show some representative performance plots before we attempt to characterize the performance.

The results reported in Figure 4 is on the sizes of the five schemes and the times it takes to read one bit vector from a file. As the bit density increases from 0.0001 to 0.5, the bit sequences become less compressible and it takes more computer memory to represent them. When the bit density is 0.0001, all four compressed schemes use less than 1% of the disk space required by the literal scheme. At a bit density of 0.5, the test sequences become incompressible and the compressed schemes all use slightly more space than the literal scheme. As expected, the two word based schemes, WAH and WBC, take up more space than the two byte based schemes, BBC and gzip. In many cases, files storing the



(a) logical OR time (b) total time

Figure 5: CPU seconds needed to perform a bitwise OR operation. The total time includes the time to read the two operands from files.

WAH and WBC bit vectors are 2.5 times as large as those for the byte based schemes. When the bit density is 0.0001, each BBC run represents about 10,000 bits and can be coded in three bytes. The same 10,000 bits are represented by two words in both WAH and WBC schemes. At a bit density of 0.05, the word based schemes take almost as much space as the literal scheme. The probability that two consecutive words contain only zero bits is $(1 - 0.05)^{64} = 0.04$. Since the word-aligned schemes can only compress fills that are more than one word in length, the two word-aligned scheme can save about 4% space compared to the literal scheme. The probability that two consecutive bytes contain only zero bits is $(1 - 0.05)^{16} = 0.44$. In other words, there are many short fills that can be compressed by the two byte based schemes.

On the random dataset, the two word based schemes, WAH and WBC use nearly the same amount of space and can be read in about the same amount of time. The two byte based schemes, BBC scheme and gzip use about the same amount of disk space.

The most likely scenario of using these bit vectors in a real database systems is to read a number of them from files and then perform bitwise logical operations. Thus the time to read bit vectors from files is an important aspect of the overall performance. 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 significant amount of CPU cycles to decompress the data files into the literal representation. This is clearly the case as shown in Figure 4.

Figure 5 show the times it takes to perform bitwise logical operations. It shows timing results of logical operation OR. Each figure has two plots, one showing only the logical operation time and the other showing the total time including the times to read the two operands from files. In an actually application, once the bit vectors are read into memory, they are likely to be used more than once. The average cost of a logical operation would be somewhere between what is shown in the left plot and the right plot.

Among the schemes shown in this set of figures, it is clear that the two word-aligned schemes, WAH and WBC, use about the same amount of time and they use much less time than both the BBC and the gzip schemes. In all test cases, the gzip scheme uses at least three times more time than the literal scheme. In half of the test cases, the WAH and WBC scheme are more than an order of magnitude faster the BBC scheme.

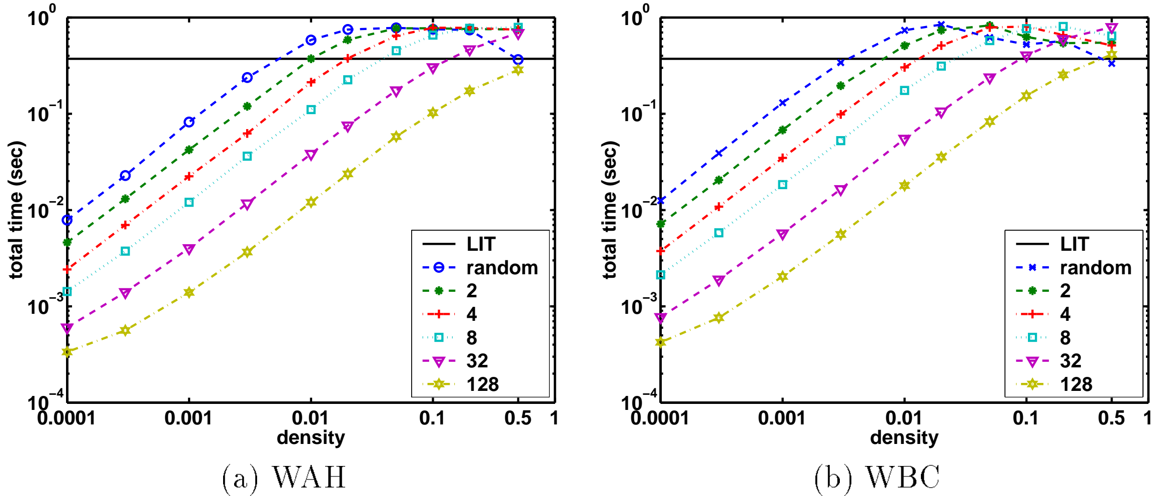
The logical operations on the uncompressed literal bit vectors are faster than those on the compressed ones in some cases. For the random test data, when bit density is between 0.01 and 0.5, the logical operations on literal bit vectors can be up to eight times faster than all other methods. However, the differences among the total times are smaller because it takes longer to read the literal bit vectors. In Figure 5, we see that when bit density is 0.5, the lines for both WAH and WBC fall right on top of the line for the literal scheme. In these cases, both WAH and WBC bit vectors contain only literal words and they can perform logical operations as fast as the literal scheme. In general, it is possible to force any bit sequence to be stored in such a manner. However, so far we are not able determine exact when to only keep the decompressed form without also keep the compressed form as well.

4.3 Analyses

We have seen that WAH and WBC outperforms BBC on random data. Here we attempt to reveal the reasons behind the performance differences.

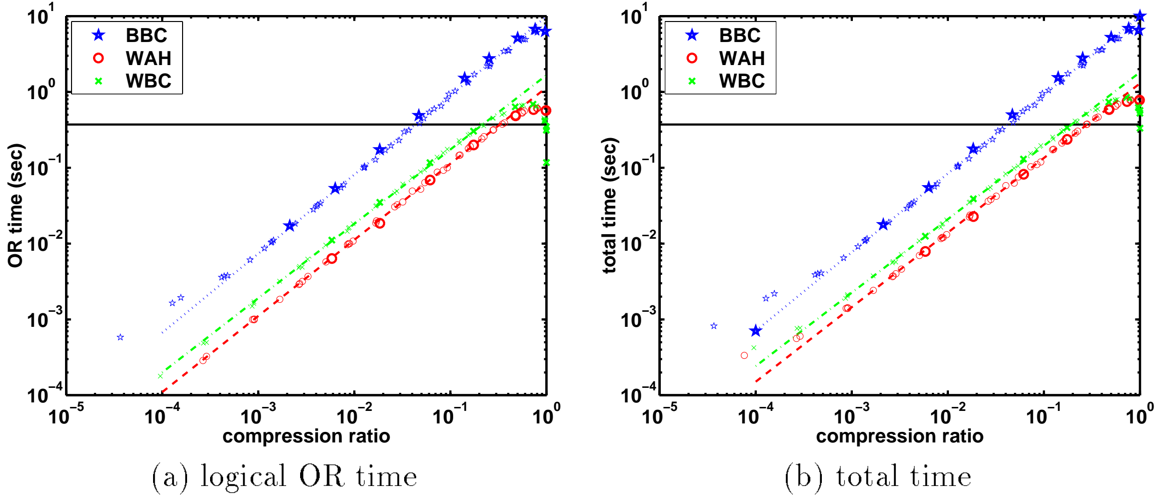
From Figure 5, we might conclude that bit density determines the performance. However, the clustering factor of the bit sequences also strongly affect the logical operation speed as shown in Figure 6. From these figures it is not clear what it is the relation among the bit density, the clustering factor and the logical operation time. In addition, computing the bit density and the clustering factor for a bit sequence is time-consuming because one has to examine every bit. It turns out that the logical operation time is directly proportional to the number of code words used to represent the bit sequences [14]. Since the compression algorithms used in the two word-aligned bit vectors are very inexpensive, it is efficient to first compress a bit sequence than determine if it should be decompressed or not.

The time-consuming part of a logical operation includes decoding each code word of the two operands of the logical operation and encode the result bit vector. The decode operation is proportional to the total number code words in the two operands. The logical operations of BBC, WAH and WBC generates code words(bytes) directly. For each such code words(bytes), the encoding function is invoked to determine whether it can be combined with the preceding code word. In most cases, the total time of this encoding process is also directly related to the total number of code words in the two operands [14]. Logical operations on BBC bit vectors are slower than the word based schemes because its encode and decode algorithms are much more complex and it needs to invoke them more times. The WBC scheme is modeled after the BBC scheme. Because BBC has four different run types and WBC only has one, it is easy to see that the encoding and decoding algorithm for BBC would be much more complex than that of WBC. Since BBC use bytes to represent bit sequences, the fills and literal tails typically contain less bits than the corresponding WBC code words. Because of this, there are more BBC files and tails than WBC. This causes the encoding and decoding algorithm to be invoked more time for BBC than for WBC.



(a) WAH (b) WBC

Figure 6: Clustering factor improves the logical operation speed.



(a) logical OR time (b) total time

Figure 7: Logical operation time is almost proportional to compression ratio.

To verify that the logical operation time is proportional to the bit vector sizes, we plotted the time against the sizes in Figure 7. The data points in this plot include tests from both the random dataset (larger points) and the Markov dataset (smaller points). In this figure we choose to use the compression ratios to represent the number of code words in the bit vectors. The compression ratio is the ratio of space required to store the compressed bit vector to that of the uncompressed one. Given that the bit sequences have exactly the same number of bits, the average compression ratio is proportional to the total number of code words. In the log-log plots, it is easy to see that there is a linear relation between the logical operation time and the compression ratio when the compression ratio is less than 0.5. The same is true for the total time except the lower left corner. This is because the time to read the small files (those with very low compression ratios) are dominated by the IO system

LIT	gzip	BBC	WAH	WBC
12.4	2.01	2.43	3.60	3.50

Figure 8: Total sizes (MB) of the bitmap indexes stored in various schemes.

overhead that is not related to file sizes.

Let t denote either the logical operation time or the total time, and let r denote the compression ratio, assuming t and r are related by $\log(t) = \alpha \log(r) + \beta$, we can use a linear regression to determine the coefficient α. If α is one, then the relation between t and r is linear. For both the logical operation time and the total time of all three compressed bit vector schemes, the values of α are in the range of 0.96 to 1.04. In fact the α value for the logical operations on WAH bit vectors is 1.004. This confirms that the relation between the logical operation time and the compression ratio is indeed linear. The dashed lines in Figure 7 are from the result of these linear regressions.

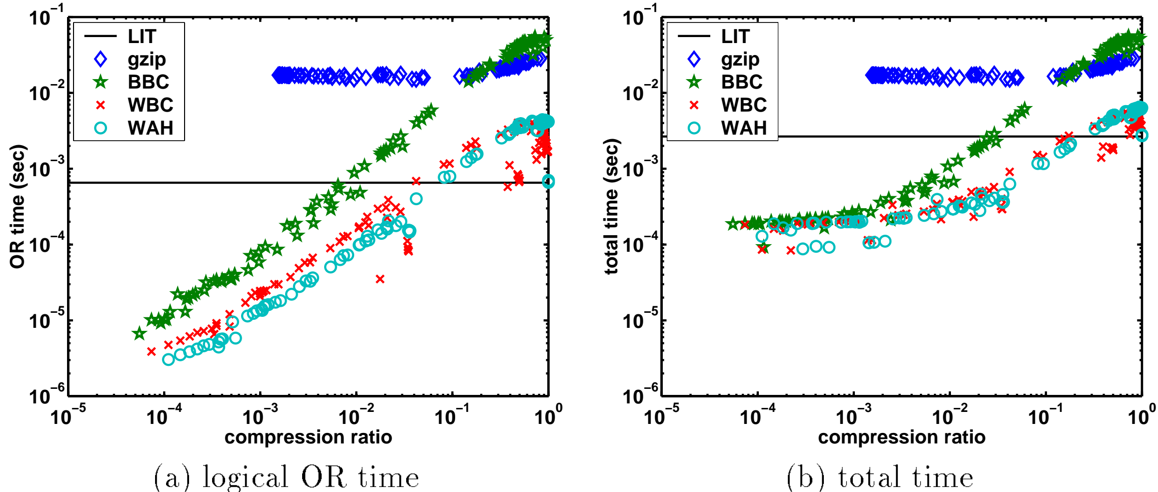
5 Performance on a real dataset

This project was initially motivated by the needs of a high energy Physics project called STAR[3] [11, 12]. In this section, we present some timing results on bitmap indexes for a set of actual data generated from STAR. There are 868,437 tuples and 502 attributes in this dataset. To limit the scope of this test, we selected 10 attributes that are most likely to be used in an actual user query. In this test, each attribute is indexed like attribute **X** in Table 1, i.e., the bit sequences represent a binning of the attribute values. The test index uses 12 bins for each attribute. The expected range of each attribute is divided into 10 bins. Two additional bins are used to store values fall blow and above the expected range.

After generating all the bit sequences, we store them in four different bit vector schemes, LIT, BBC, WAH, and WBC. Figure 8 shows the total sizes of the files that hold the bitmap indexes. On this bitmap index, the compressed schemes use less than one third of the space required by the literal scheme. Comparing the word-aligned schemes against BBC, WAH and WBC use about 50% more space. From the previous tests, we know that it is possible for WAH to use as much as 2.6 times the space as BBC. On this set of application data, WAH uses only about 50% more space than BBC because most of the space are taken up by the incompressible bit vectors. Both BBC and WAH use about the same amount of space to represent these bit vectors.

Figures 9 shows the logical operation performance on the two sets of indexes. As before the left plot shows the logical operation time and the right shows the total time including the time to read the two bit vectors from files. In both plots, the horizontal axes represent compression ratios. The test cases used for generating the data in Figures 9 involve all the bit sequences of the test dataset. Because the bit sequences are relative small, in many cases the IO overhead dominates the total logical operation time and the total time is around 2×10^{-4}seconds. In these anomalous cases, the total times are about same for both

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



(a) logical OR time (b) total time

Figure 9: Time (CPU seconds) needed to perform bitwise OR on the STAR dataset.

word-aligned schemes and byte-aligned scheme. However, in most cases, the word-aligned schemes are still significantly faster. When the compression ratios are close to one, the logical operations on WAH bit vectors are close to 80 times faster than the same operations on BBC bit vectors. In these cases, even include the time to read two bit vectors, the WAH scheme is still about 20 times faster than BBC. If we sum up all the total times from all test cases, the sum for BBC is about 12 times of those for WAH and WBC. In other words, on average WAH and WBC are about 12 times as fast as BBC. If the IO time is not included, the average differences are even larger.

Compared to the literal scheme, the BBC scheme is faster in less than half of the test cases, the WAH and WBC are faster in about 60% of the test cases. Even in the worst cases, the two word-aligned schemes take no more than twice as long as the literal scheme.

6 Summary

To enhance the effectiveness of bitmap indexes for large datasets, we set out to search for a compressed bit vector that is able to support fast bitwise logical operations. The essence of our approach is to trade space for speed. More specifically, we noticed that the best existing schemes are byte-aligned. Naturally, we wanted to check the effectiveness of the word-aligned schemes. In this paper, we have developed two such schemes, namely, WAH and WBC. They are expected to be faster at a cost of more space. What is unexpected it that they use only 50% more space than BBC, but are 12 times faster. For large indexes, this performance enhancement should greatly improve the response time to a query. Because the cost in space is modest, we believe the word-aligned schemes are superior data structure for storing bitmap indexes.

7 Acknowledgments

The authors wish to express our sincere gratitude to Professor Ding-Zhu Du for his help in simplifying the analyses of the complexity of the logical operations.

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] 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.

[4] 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.

[5] 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*. Morgan Kaufmann, 1999. A longer version appeared as AT&T report number AMERICA112.

[6] 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`.

[7] 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, New York, 1992.

[8] 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.

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

[10] Julian Seward. *bzip2 and libbzip2*, March 2000. Source code available at `http://-sourceware.cygnus.com/bzip2`.

[11] 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*. IEEE Computer Society, 1999.

[12] 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.

[13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing, 3nd edition, 1997.

[14] Kesheng Wu, Ekow J. Otoo, Arie Shoshani, and Henrik Nordberg. *Notes on Design and Implementation of Compressed Bit Vectors*. Berkeley, CA, 2001. In preparation.

[15] 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.