LBNL-49626

An Efficient Compression Scheme For Bitmap Indices*

Kesheng Wu, Ekow J. Otoo and Arie Shoshani

April 13, 2004

Abstract

When using an out-of-core indexing method to answer a query, it is generally assumed that the I/O cost dominates the overall query response time. Because of this, most research on indexing methods concentrate on reduceing the sizes of indices. For bitmap indices, compression has been used for this purpose. However, in most cases, operations on these compressed bitmaps, mostly bitwise logical operations such as AND, OR, and NOT, spend more time in CPU than in I/O. To speedup these operations, a number of specialized bitmap compression schemes have been developed; the best known of which is the byte-aligned bitmap code (BBC). They are usually faster in performing logical operations than the general purpose compression schemes, but, the time spent in CPU still dominates the total query response time. To reduce the query response time, we designed a *CPU-friendly* scheme named the word-aligned hybrid (WAH) code. In this paper, we prove that the sizes of WAH compressed bitmap indices are about two words per row for large range of attributes. This size is smaller than typical sizes of commonly used indices, such as a B-tree. Therefore, WAH compressed indices are not only appropriate for low cardinality attributes but also for high cardinality attributes.

In the worst case, the time to operate on compressed bitmaps is proportional to the total size of the bitmaps involved. The total size of the bitmaps required to answer a query on one attribute is proportional to the number of hits. These indicate that WAH compressed bitmap indices are optimal. To verify their effectiveness, we generated bitmap indices for four different datasets and measured the response time of many range queries. Tests confirm that sizes of compressed bitmap indices are indeed smaller than B-tree indices, and query processing with WAH compressed indices is much faster than with BBC compressed indices, projection indices and B-tree indices. In addition, we also verified that the average query response time is proportional to the index size. This indicates that the compressed bitmap indices are efficient for very large datasets.

1 Introduction

This research was originally motivated by the need to manage the volume of data produced by a high-energy physics experiment called STAR[1] [22]. In this experiment, information about each potentially interesting collision event is recorded. 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 performed on some selected events. One important way of selecting events is to apply some conditions on the summary attributes such as "`Energy > 15 GeV and 700 < NumParticles < 1300`" [4, 22].

*The authors wish to express our sincere gratitude to Professor Ding-Zhu Du for his helpful suggestion in simplifying the analyses of the time complexity of logical operations, and to Drs. Doron Rotem and Kurt Stockinger for their help in reviewing the drafts of this paper. 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`.

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.

These queries are known as *partial range queries*. One of the data management tasks is to answer these queries efficiently. Since these summary attributes are only read, not modified, the indexing schemes used in commercial data warehouse applications should be useful for our task. Based on data warehouse literature, we know that the bitmap index is particularly efficient on datasets with many attributes [5, 7, 17, 31]. Since our datasets have hundreds of attributes, the bitmap index is even more appropriate [22].

A bitmap index consists of a set of bit sequences that represent some information about the indexed attribute. These bit sequences are generally known as bitmaps. Figure 1 shows a set of such bitmaps for an attribute **X** of a tiny table (**T**), consisting of eight tuples (rows). The attribute **X** can have one of four values, 0, 1, 2 and 3. There are four bitmaps, appearing as four columns in Figure 1, each representing whether the value of **X** is one of the four choices. For convenience, we label the four bitmaps as $b_1, \ldots, b_4$. When processing the query "`select * from T where X < 2`," the main operations on the bitmaps is the bitwise logical operation "b_1 OR b_2." Since bitwise logical operations are well supported by computer hardware, bitmap indices are very efficient to use [17]. In many data warehouse applications, bitmap indices perform better than tree-based schemes, such as the variants of B-tree or R-tree [12, 5, 17, 31]. According to the performance model proposed by Jürgens and Lenz [12], bitmap indices are likely to be even more competitive in the future as disk technology improves. In addition to supporting queries on a single table as shown in this paper, researchers have also demonstrated that bitmap indices can accelerate complex queries involving multiple tables [19]. Realizing the value of the bitmap indices, most major DBMS vendors have implemented them in their products.

The example shown in Figure 1 is the simplest bitmap index which we call the *basic bitmap index*. There is only one attribute in the example. With more than one attribute, typically a bitmap index is generated for each attribute. It is straightforward to process queries involving multiple attributes. For example, to process a query with the condition "`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 "`Energy > 15 GeV`" and "`7 < NumParticles < 13`", and the final answer is then generated with a bitwise logical AND operation on these two partial solutions. The whole process can be carried out efficiently if indices for `Energy` and `NumParticles` involve only a small number of bitmaps. However, in real applications, especially scientific applications, a basic bitmap index will contain many bitmaps because the attribute cardinalities are high. In these cases, the bitmap indices take a large amount of space, and processing of range queries using these indices may take longer than without an index.

Compression is one way to reduce the size of the bitmap index and improve its efficiency. To compress a bitmap, a simple option is to use one of the text compression schemes, such as LZ77 (used in gzip) [9, 32]. These schemes are well-studied and efficient in reducing file sizes.

However, performing logical operations on the compressed bitmaps is usually much slower than on the uncompressed bitmaps since the compressed bitmaps have to be explicitly uncompressed. To address this performance issue, a number of specialized schemes have been proposed. Johnson and colleagues have studied the performance of many of them thoroughly [11, 1]. From their studies we know that the logical operations using these specialized schemes are usually faster than those using gzip. One such specialized scheme, called the *Byte-aligned Bitmap Code* (BBC), is very efficient [2, 3]. However, even with BBC in some cases, logical operations on the compressed bitmaps still can be orders of magnitudes slower than on the uncompressed bitmaps.

When processing range queries using BBC compressed bitmap indices, it was reported that more than 90% of the time was spent in CPU when performing logical operations [24]. The I/O time is only a small part of the total time. As main memory becomes cheaper, most computers will have even more main memory. Therefore, the commonly used bitmap indices will more likely reside in memory. This will further reduce the average I/O time during query processing and make CPU time more dominant. To reduce the total query processing time, we propose a *CPU-friendly* compression scheme that 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 ensures that the sizes of compressed bitmap indices are relatively small for attributes with very high cardinalities. More specifically, for a table with N rows, as the cardinality of the attribute increases, the WAH compressed bitmap index approaches a stable plateau of $2N$ words. A B-tree index on the same attribute is observed to use $3N \sim 4N$ words in a commercial DBMS, which is nearly twice the size of a WAH compressed bitmap index.

Analyses show that the time to answer a query using a WAH compressed bitmap index is linear in the size of the bitmaps involved. Tests on a number of datasets confirm this as well. A search algorithm is considered optimal if the time complexity is linear in the number of hits. Using the WAH compressed bitmap indices, in the worst case, the total size of the bitmaps required to answer a query is proportional to the number of hits. Since the time complexity using a WAH compressed index is linear in the total size of bitmaps, this indicates that the whole searching process is optimal.

To evaluate a query using a compressed bitmap index, at most half of the index is required. If more than half of the bitmaps are required, we can evaluate the complement of the condition, and then take the complement of the answer. This indicates that at most N words may be accessed. Another indexing scheme that accesses N words is the projection index. For high-dimensional datasets, the projection index is often the best for processing a variety of queries [18, 15, 25]. Our tests show that the WAH compressed index usually uses less time to answer the same query than the projection index. In fact, on the average, it is about three times more efficient than the projection index.

Based on their performance studies, Amer-Yahia and Johnson [1] came to the conclusion that one has to dynamically switch among different compression schemes in order to achieve the best performance. Since WAH is significantly faster than those compression schemes studied, there is no need to switch compression schemes in a bitmap indexing software. Therefore, the new compression scheme not only improves the performance of the bitmap indices 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 [13, 22, 27]. Many researchers have studied the strategy of using different encoding schemes [5, 6, 18, 26, 31]. One well-known scheme is the bit-sliced index that encodes c distinct values using $\lceil log_2 c \rceil$ bits and creates a bitmap for each binary digit [18]. This is related to the binary encoding scheme discussed elsewhere [26, 5, 31]. A drawback of this scheme is that to answer a 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 [26]. In all these schemes, an efficient compression scheme should improve their effectiveness. Additionally, a number of common indexing schemes such as the signature file [8, 10, 14] and the bit transposed files [26] may also benefit from using an efficient bitmap compression scheme.

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 for later performance comparisons. Section 3 contains the description of the word-aligned hybrid code (WAH). Section 4 contains analyses of bitmap index sizes, and Section 5 contains the analyses of time complexity to use the indices. We verify the theoretical analyses with two sets of experimental measurements. The first set is presented in Section 6, which measures the sizes of individual bitmaps and the time required to perform bitwise logical operations on pairs of bitmaps. The second set of measurements presented in Section 7 measures bitmap index sizes and time required to answer random range queries. A short summary is given in Section 8.

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 of a number of different schemes [11, 30].

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 schemes such as gzip [9]. They are highly efficient in compressing text files. We use gzip as a representative for this type of schemes because it is usually faster than others in decompressing files.

There are a number of compression schemes that offer good compression and also allow fast bitwise logical operations. The Byte-aligned Bitmap Code (BBC) is one of the well-known schemes [2, 3, 11]. BBC performs bitwise logical operations efficiently and it compresses almost as well as gzip. Our implementation of the BBC scheme is a version of the two-sided BBC code [30, Section 3.2]. This version performs as well as the improved version by Johnson [11, 24]. In both Johnson's tests [11] and ours, the time curves for BBC and gzip (marked as LZ in [11]) intersect at about the same position.

Many of the specialized bitmap compression schemes, including BBC, are based on the basic idea of run-length encoding. They represent a group of consecutive identical bits (also called a *fill* or a *gap*) by its bit value and 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 *short fill* is a fill that cannot be efficiently represented in the encoded form. For example, a fill of one bit usually cannot be efficiently represented using the run-length encoding. Many variants of the run-length encoding scheme represent these short fills literally, which make the overall compression scheme a hybrid of the run-length scheme and the literal scheme. The second improvement is to use as few bits as possible to represent the fill length. Given a bitmap, 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

[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 bitmap, except the last word that represents the four leftover bits.

whole bytes, it represents the fill length as number of bytes rather than number of bits. In addition, it uses a multi-byte scheme to represent the fill lengths [2, 11]. This strategy often uses more bits to represent a fill length than others such as ExpGol [16]. However it allows for faster operations [11].

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, 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, the byte-alignment property is crucial to the operational efficiency of BBC. Removing the alignment requirement may lead to better compression. For example, ExpGol [16] can compress better than BBC partly because it does not obey the byte alignment. However, bitwise logical operations on ExpGol bit vectors are usually much slower than on BBC bit vectors [11].

3 Word-aligned bitmap compression scheme

When we started testing the BBC compressed indices, we noticed that the time spent on bitwise logical operations is dominated by time spent in CPU rather than in reading bitmaps from disk. If we can reduce the time spent in CPU, it might be possible to reduce the overall query response time. Most of the known compression schemes are byte based, that is, they access computer memory one byte at a time during most operations. On a modern computer, accessing one byte takes at least as much time as accessing one word [21]. One obvious strategy is to develop a word-based compression scheme[3]. Among the schemes tried, we found that the *word-aligned hybrid code* (WAH) was the best [30]. It is very simple since there are only two ways to interpret a word in the compressed data. It is also efficient since all operations can be performed on words without extracting individual bits or bytes.

The word-aligned hybrid (WAH) code is similar to BBC in that it is also 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 bytes. There are two types of regular 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 us to distinguish a literal word from a fill word without explicitly extracting the bit. Let w denote the number of bits in a word, the lower $(w-1)$ 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 remaining bits store the fill length. WAH imposes the word-alignment requirement on the fills, i.e., all fill lengths must be integer multiples of the number of bits in a literal word $(w-1)$. This requirement is key in ensuring that logical operations only access words, and not bytes or bits.

Figure 2 shows a WAH bit vector representing 128 bits. In this example, we assume each computer word contains 32 bits. Under this assumption, each literal word stores 31 bits from the bitmap, and each fill word represents a multiple of 31 bits. The second line in Figure 2 shows how

[3]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 bitmap typically have complex dependencies, the MMX technology is unlikely to be effective in enhancing their performance.

uncompressed (in 31-bit groups)					
A	40000380	00000000	00000000	001FFFFF	0000000F
B	7FFFFFFF	7FFFFFFF	7C0001E0	3FE00000	00000003
C	40000380	00000000	00000000	00000000	00000003
compressed					
A	40000380	80000002		001FFFFF	0000000F
B	C0000002		7C0001E0	3FE00000	00000003
C	40000380	80000003			00000003

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

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 WAH words. The first three words are regular 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 a multiple of the literal word size. The fourth word is the *active word*, it stores the last few bits that could not be stored in a regular word. Another word with the value four, not shown, is needed to stores the number of useful bits in the active word.

The detailed algorithms for performing logical operations are given in the appendix. Here we briefly describe one example, see Figure 3. To perform a logical AND operation, we essentially need to match each group of 31 bits from the two operands, and generate the corresponding groups for the result. 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 separately.

4 Compressed sizes of bitmap indices

In this section, we examine the space complexity of the WAH compression scheme. More specifically, we develop analytical expressions for the compressed sizes of some random bitmaps, and the compressed index sizes of random attributes. These expressions are useful because they are the upper bounds of sizes of similar bitmaps and indices. In the next section, we discuss the complexity of using the compressed bitmaps to answer user queries.

4.1 The counting groups

Before attempting to derive the expression for sizes of random bitmaps, let us first give some basic definitions. Let w be the number of bits in a computer word. On a 32-bit CPU, the value of w is 32, and on a 64-bit CPU, $w = 64$. In the following analyses, we assume the number of bits in a bitmap can be represented by a computer word[4], i.e., $N < 2^w$, where N is the number of bits in a bitmap.

Given a bitmap, the WAH compression scheme divides the bits into $(w-1)$-bit groups. For convenience of discussions, we call a $(w-1)$-bit group a *literal group*. For a bitmap with N bits, there are $\lceil N/(w-1) \rceil$ such groups. The first $M \equiv \lfloor N/(w-1) \rfloor$ groups each contains $(w-1)$ bits. They can be represented by regular words in WAH. The remaining bits are stored in the active

[4]If the database contains more record than 2^w rows, multiple bitmaps can be used each to represent a subset of the rows. In a robust implementation, most likely the bitmap index will be segmented, and each bitmap will only have a few thousands or millions of bits.

31-bit groups	40000380	00000000	00000000	001FFFFF
counting groups	40000380	00000000		
		00000000	00000000	
			00000000	001FFFFF

Figure 4: Breaking the 31-bit groups from Figure 2 into three counting groups.

word. The regular words could all be literal words, each representing one literal group. In later discussions, we refer to it as the *decompressed form* of WAH. This form requires M words plus the active word and the word to record the number of bits in the active word, a total of $M + 2$ words.

A bitmap can be represented by WAH compression scheme in less than $M + 2$ words if some of the literal groups are fills. A WAH fill is a set of neighboring literal groups that have identical bits. In this context, the word-alignment properties refers to the fact that all fills must span an integer number of literal groups. In the example bitmap shown in Figure 2, the longest fill without the word-alignment restriction contains 79 bits of zero, but with the restriction, the longest fill only contains 62 bits.

Lemma 1 *If a word contains more than five bits, $w > 5$, then any fill containing no more than 2^w bits can be represented in one WAH fill word.*

Proof. A WAH fill word dedicates $w-2$ bits to represent the fill length, therefore the maximum fill length can be $2^{w-2} - 1$. The fill length is measured in number of literal groups. This means the maximum number of bits that can be represented is $(w-1)(2^{w-2}-1)$. It is easy to verify that when $w > 5$, $(w-1)(2^{w-2}-1) > 2^w$. ∎

To minimize the space required, this lemma indicates that we should use one fill word to represent all neighboring literal groups that have identical bits. Naturally, we define a *WAH fill* to contain all neighboring literal groups with identical bits. The number of regular words in a WAH compressed bitmap is equal to the number of fills plus the number of literal groups. However, since a fill can be of any length, counting the number of fills is messy. Instead we define an auxiliary concept called the *counting group* to simplify the counting process. A counting group always consists of two consecutive literal groups, and the counting groups are allowed to overlap. For a bitmap with M literal groups, there are exactly $M - 1$ counting groups, as illustrated in Figure 4. The next lemma states how the counting groups are related to fills.

Lemma 2 *A WAH fill that spans l literal groups generates exactly $(l-1)$ counting groups that are fills.*

Proof. To prove this lemma we first observe that any l literal groups can be broken into $(l-1)$ counting groups. If the l literal groups are a fill, then the $(l-1)$ counting groups are also fills. The two literal groups at the ends of the fill may be combined with their adjacent groups outside the fill to form up to two counting groups. According to our definition of a WAH fill, the groups adjacent to the fill must be one of the following: (1) a fill of different type, (2) a literal group with mixed zeros and ones, or (3) null, i.e., there is no more literal groups before or after the fill. In the last case, no more counting groups can be constructed. In the first two cases, the counting groups constructed with one literal group within the fill and one literal group outside are not fills. Therefore the fill generates exactly $(l-1)$ counting groups that are fills. ∎

Combining the above two lemmas gives an easy way to compute the number of words needed to store a compressed bitmap.

Theorem 3 *Let G be the number of counting groups that are fills, the number of regular words in a compressed bitmap is $M - G$.*

Proof. The number of regular words in a compressed bitmap is the sum of the number of fills and the number of literal groups that are not fills. Let L be the number of fills of a bitmap and let l_i be the size of ith fill in the bitmap, according to the previous lemma it must contribute $(l_i - 1)$ counting groups that are fills. By definition, $G \equiv \sum_{i=1}^{L}(l_i - 1)$. The number of fills is $\sum_{i=1}^{L} l_i - \sum_{i=1}^{L}(l_i - 1)$, and the number of groups that are not in any fills is $M - \sum_{i=1}^{L} l_i$. Altogether, the number of regular words is

$$\left(\sum_{i=1}^{L} l_i - \sum_{i=1}^{L}(l_i - 1)\right) + \left(M - \sum_{i=1}^{L} l_i\right) = M - \sum_{i=1}^{L}(l_i - 1) = M - G.$$

■

One important consequence of this theorem is that to compute the size of the compressed bitmap we only need to compute the number of counting groups that are fills, which is relatively easy to do for a number of random bitmaps.

4.2 Sizes of random bitmaps

Now that we have the basic tool for counting the size of a compressed bitmap, we can examine the size of some random bitmaps. The type of bitmaps that are hardest to compress are the random bitmaps where each bit is generated independently following an identical probability distribution. We refer to this type of random bitmaps as *uniform random bitmaps* or simply random bitmaps. Random bitmaps can be characterized with one parameter, the *bit density* d. The bit density is the fraction of bits that are one ($1 \geq d \geq 0$).

The efficiency of a compression scheme is often measured by the *compression ratio*, which is the ratio of its compressed size and its uncompressed size. For a bitmap with N bits, the uncompressed scheme (LIT) needs $\lceil N/w \rceil$ words, and the decompressed form of WAH requires $M + 2$ words. The compression ratio of the decompressed WAH is $(\lfloor N/(w-1) \rfloor + 2)/\lceil N/w \rceil \approx w/(w-1)$. All compression schemes pay an overhead when representing incompressible bitmaps. For WAH, this overhead is one bit per word. When a word is 32 bits, this overhead is about three percent. This overhead for BBC is about one byte per 15 bytes, roughly six percent.

Let d be the bit density of a uniform random bitmap, the probability of finding a counting groups that is a 1-fill, i.e., $2w-2$ consecutive bits that are one, is d^{2w-2}. Similarly, the probability of finding a counting group that is a 0-fill is $(1-d)^{2w-2}$. With WAH compression, the expected size of a bitmap with N bits is

$$\begin{aligned} m_R(d) &\equiv M + 2 - G = \left\lfloor \frac{N}{w-1} \right\rfloor + 2 - \left(\left\lfloor \frac{N}{w-1} \right\rfloor - 1 \right) \left((1-d)^{2w-2} + d^{2w-2} \right) \\ &\approx \frac{N}{w-1} \left(1 - (1-d)^{2w-2} - d^{2w-2} \right). \end{aligned} \quad (1)$$

The above approximation neglects the constant 2, which corresponds to the two words comprising of the active word and the counter for the active word. This constant may become important when G is close to M, i.e., m_R is close to 2. For application where compression is useful, N is typically much larger than w. In these cases, dropping the floor operator ($\lfloor\rfloor$) does not introduce significant amount of error.

The compression ratio is approximately $w(1 - (1-d)^{2w-2} - d^{2w-2})/(w-1)$. For d between 0.05 and 0.95, the compression ratios are nearly one. In other words, these random bitmaps can't be compressed with WAH. For sparse bitmaps, say $2wd \ll 1$, $m_R(d) \approx 2dN$, since $d^{2w-2} \to 0$ and $(1-d)^{2w-2} \approx 1 - (2w-2)d$. In this case, the compression ratios are approximately $2wd$.

Let h denote the number of bits that are one. Using the definition of bit density, $h = dN$. The compressed size of a sparse bitmap is related to h according to the following equation.

$$m_R(d) \approx 2dN = 2h. \tag{2}$$

In such a sparse bitmap, all literal words contain only a single bit that is one, and each literal word is separated from the next by a 0-fill. On the average, two words are used for each bit that is one [29].

The second type of bitmap that we can compute its size analytically is the bitmaps generated from a two-state Markov process. These bitmaps require a second parameter to characterize it. We call this second parameter the *clustering factor* f. The clustering factor of a bitmap is the average number of bits in all 1-fills. The bit density d and the cluster factor f can fully specify this simple Markov process. Let the two states of the Markov process be named 0 and 1. A bitmap generated from this Markov process is a recording of the states. The transition probabilities of the Markov process are p for going from state 0 to state 1, and q for going from state 1 to state 0. Starting with a bit 1, the probability that there are i more bits of the same value is proportional to $(1-q)^i$. The expected number of bits in a 1-fill is as follows[5].

$$\begin{aligned} f &= \frac{\sum_{i=0}^{\infty}(i+1)(1-q)^i}{\sum_{i=0}^{\infty}(1-q)^i} = -q\sum_{i=0}^{\infty}\frac{d(1-q)^{i+1}}{dq} = -q\frac{d}{dq}\sum_{i=0}^{\infty}(1-q)^{i+1} \\ &= -q\frac{d}{dq}\left(\sum_{j=0}^{\infty}(1-q)^j - 1\right) = -q\frac{d}{dq}\left(\frac{1}{q} - 1\right) = \frac{1}{q}. \end{aligned}$$

Similarly, the average size of a 0-fill is $1/p$. The bit density d is $(1/q)/(1/p+1/q) = p/(p+q)$. The transition probabilities can be expressed in terms of d and f as follows (Note: $f \geq 1$, $f \geq d/(1-d)$, and $1 > d > 0$):

$$q = 1/f, \qquad \text{and} \qquad p = \frac{d}{(1-d)f}.$$

The probability of finding a counting group that is a 0-fill, i.e., $(2w-2)$ consecutive bits of zero, is $(1-d)(1-p)^{2w-3}$. Similarly, the probability of finding a counting group that is a 1-fill is $d(1-q)^{2w-3}$. Based on these probabilities, we expect the compressed size to be

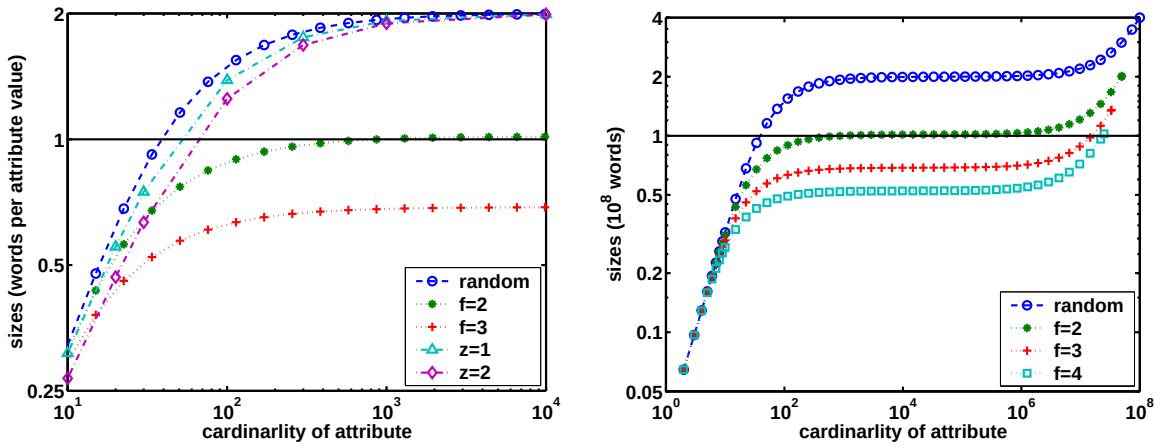
$$\begin{aligned} m_M(d,f) &\equiv \left\lfloor\frac{N}{w-1}\right\rfloor + 2 - \left(\left\lfloor\frac{N}{w-1}\right\rfloor - 1\right)\left((1-d)(1-p)^{2w-3} + d(1-q)^{2w-3}\right) \\ &\approx \frac{N}{w-1}\left(1 - (1-d)\left(1 - \frac{d}{(1-d)f}\right)^{2w-3} - d\left(\frac{f-1}{f}\right)^{2w-3}\right). \end{aligned} \tag{3}$$

When $2wd \ll 1$ and $f < 10$, almost all of the fills are 0-fills. In this case, the compression ratio is approximately $wd(1 + (2w-3)/f)/(w-1)$. In other word, the size of a compressed bitmap is nearly inversely proportional to the clustering factor.

4.3 Sizes of bitmap indices

With the formulas for two types of random bitmaps, we can compute the bitmap index sizes of some discrete random attributes. In the first type of random attribute, the probability of one value does not depend on another value. The bitmaps in an index for this type of attribute are uniform random bitmaps, though the bitmaps may have different bit densities. Let us first examine the simple case where the discrete attribute has a uniform distribution. In this case, the bit densities of the bitmaps are the same, and inversely proportional to the attribute cardinality. Let the cardinality be c, the

[5]Note $\sum_{i=0}^{\infty}(1-q)^i = 1/q$. The computation here is similar to that for rehash chain length in [20, Section 8.6].



(a) $n = \infty$ (b) $n = 10^8$

Figure 5: The expected sizes of bitmap indices on random data and Markov data with various clustering factors.

bit density is $d = 1/c$. Since a bitmap index consists of c bitmaps, the total size (in number of words) is

$$s_U \equiv cm_R(1/c) \approx \frac{Nc}{w-1}\left(1 - (1-\frac{1}{c})^{2w-2} - (\frac{1}{c})^{2w-2}\right). \tag{4}$$

When c is large, $(1/c)^{2w-2} \to 0$ and $(1-1/c)^{2w-2} \approx 1-(2w-2)/c$. The total size of the compressed bitmap index for a uniform random attribute has the following asymptotic formula.

$$cm_R(1/c) \approx \frac{Nc}{w-1}\left(1 - \left(1 - \frac{2w-2}{c}\right)\right) = 2N, \qquad (c \gg 1). \tag{5}$$

This formula is based on an approximation of Equation 1 which neglected about two words per bitmap. Since there are c bitmaps, for very large c, a better approximation is

$$s_U \approx 2N + 2c.$$

When the probability distribution is not uniform, let the ith value have a probability of p_i, then the total size of the bitmap index compressed with WAH is

$$s_N \equiv \sum_{i=1}^{c} m_R(p_i) \approx \frac{N}{w-1}\left(c - \sum_{i=1}^{c}(1-p_i)^{2w-2} - \sum_{i=1}^{c} p_i^{2w-2}\right). \tag{6}$$

Under the constraint that all the probabilities must sum to one, i.e., $\sum p_i = 1$, the above equation achieves its maximum when all p_i are equal. In other words, the bitmap indices for uniform random attributes are the largest.

In the second type of random attribute, the probabilities are allowed to depend on one other value, one such example is a simple uniform c-state Markov process. From any state, this Markov process has the same transition probability q to other states, and it selects one of the $c-1$ states with equal probability. In this case, the bitmap corresponding to each of the c values has the same size as the Markov bitmaps given in Equation 3, where $d = 1/c$. The total size is

$$s_M \equiv cm_M(\frac{1}{c}, f) \approx \frac{Nc}{w-1}\left(1 - \left(1-\frac{1}{c}\right)\left(1 - \frac{1}{(c-1)f}\right)^{2w-3} - \frac{1}{c}\left(1-\frac{1}{f}\right)^{2w-3}\right). \tag{7}$$

Figure 5 shows the sizes of the a number of different bitmaps. The line marked "random" is for the uniform random attributes. The lines marked with "f=2" and "f=3" are attributes generated

from the Markov process with specified clustering factor f. The two lines marked with "z=1" and "z=2" are attributes with Zipf distribution where $p_i \propto i^{-z}$.

Figure 5 displays the total sizes of the bitmaps under two conditions, the infinite N and a finite N. In the first case, the cardinality is always much less than N. As the cardinality increases, the size of bitmap index of a uniform random attribute approaches the stable plateau of two words per attribute value. When N is finite, it is possible for c to be as large as N. In this case, we can not simply drop the small constants in Equations 1 and 3. If $c = N$, the total size of bitmap index takes about $4N$ words, and most of the bitmaps have three regular words plus the active word[6]. Since a B-tree index takes as much as $4N$ words, the worst-case size of a compressed bitmap index is about the same as that of a B-tree index. For a large range of high cardinality attributes, $c < 0.01n$, the maximum size of WAH compressed bitmap indices is about $2N$ words.

For attributes with clustering factor f greater than one, the stable plateau is reduced by a factor close to $1/f$. Another factor that limits the total size of the compressed bitmap index is that the cardinality of an attribute is usually much smaller than N. For attributes with Zipf distribution, the stable plateau is the same as the uniform random attribute. However, because the actual cardinality is much less than N, it is very likely that the size of the compressed bitmap index would be about $2N$ words. For example, for an attribute with Zipf distribution with $z = 1$ and $i < 10^9$, among 100 million values, we see about 27 million distinct values, and the index size is about $2.3N$ words. Clearly, for Zipf distributions with larger z, we expect to see less distinct values and the index size would be smaller. For example, for $z = 2$, we see about 14,000 distinct values for nearly any limit on i that is larger than 14,000. In these cases, the index size is about $2N$. The following theorem summarizes these observations.

Theorem 4 *Let N be the number of rows in a table, and let c be the cardinality of the attribute to be indexed, then the size s of the bitmap index compressed with WAH is such that,*

1. *it never takes more than $4N$ words,*
2. *if $c < 0.01N$, the maximum size of the compressed bitmap index of the attribute is about $2N$ words,*
3. *and if the attribute has a clustering factor $f > 1$ and $c < 0.01N$, the maximum size of its compressed bitmap index is*
$$s \sim \frac{N}{w-1}\left(1 + \frac{2w-3}{f}\right),$$
which is nearly inversely proportional to the clustering factor f.

Proof. In the above theorem, item (2) is a restatement of Equation 5, and item (3) is an approximation of Equation 7. Item (1) can be proven by verifying that the maximum space required for an uniform random attributes is achieved when its cardinality c is equal to N. This can be done by examining the expression for $cm_R(1/c)$ without dropping the small constant or removing the floor operator. In this extreme case, each bitmap contains only a single bit that is one. This bitmap requires at most three regular WAH words plus one active word[6]. Since there are N such bitmaps, the space required in this extreme case is $4N$ words. ∎

When the probability distribution is not known, we can estimate the sizes in a number of ways. If we only know the cardinality, we can use Equation 4 to give an upper bound on the index size. If the histogram is computed, we can use the frequency of each value as the probability p_i to compute the size of each bitmap using Equation 6. We can further refine the estimate if the clustering factors are computed. For a particular value in a dataset, computing the clustering factor requires one to

[6]Since all active words have the same number of bits, one word is sufficient to store this number.

scan the data to find out how many times it appears in consecutive groups, including groups of size one. The clustering factor is the total number of appearances divided by the number of consecutive groups. With this additional parameter, we can compute the size the compressed bitmaps using Equation 3. However, since the total size of the compressed bitmap index is relatively small in all cases, one can safely generate a index without first estimating its size.

5 Time complexity of query processing

In using the WAH compressed bitmap index to answer queries, bitwise logical operations are the most important operations. For this reason, we next examine the complexity of the bitwise logical operation procedures. Two different algorithms are given in the appendix, see Listings 3 and 4. Algorithm `generic_op` in Listing 3 performs an arbitrary bitwise logical operation on two compressed bitmaps, and algorithm `inplace_or` in Listing 4 performs a bitwise OR between a decompressed bit vector and a compressed one. The first one is for general use and the second one is mainly used to sum together a large number of sparse bitmaps. Before we give detailed analyses, we first summarize the main points.

The time to perform an arbitrary logical operation between two compressed bitmaps is proportional to the total size of the two bitmaps. The exception is when the two operands are nearly decompressed, in which case the time needed is constant. The time to perform a logical OR operation between a decompressed bit vector and a compressed one is linear in the size of the compressed one. When ORing a large number of sparse bitmaps using `inplace_or`, the total time is linear in the total size of all input bitmaps. In this case, using `generic_op` takes more time because it allocates memory for intermediate results and compresses them too. In contrast, using `inplace_or` avoids all of these operations.

Operations on WAH compressed bitmaps are faster than the same operations on BBC compressed bitmaps for three main 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 [11]. It may take up to three tests in order to decide the run type of a header byte and 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. For this reason, BBC needs more time to transfer its data between the main memory and CPU registers than WAH.

3. BBC can encode short fills, say those with less than 60 bits, more compactly than WAH. However, this comes at a cost. Each time BBC encounters a short fill it starts a new run. WAH typically represents such a short fill in literal words. It takes much less time to operate on a literal word in WAH than on a run in BBC. This situation is common when bit density is greater than 0.01 in random bitmaps.

5.1 Complexity of algorithm generic_op

In the appendix, we show four pseudocode segments to illustrate two basic algorithms used to perform bitwise logical operations. The code segments follow the syntax of C++ and make use of the standard template library functions. The first two listings contain data structures and supporting functions for the two main functions, `generic_op` and `inplace_or`. In these listings, compressed bitmaps are represented as objects of type `bitVector`. The three key functions used for performing bitwise logical operations are `bitVector::appendLiteral`, `bitVector::appendFill`, and `run::decode`, where the first two are shown in Listing 1 and the last one is shown in Listing 2.

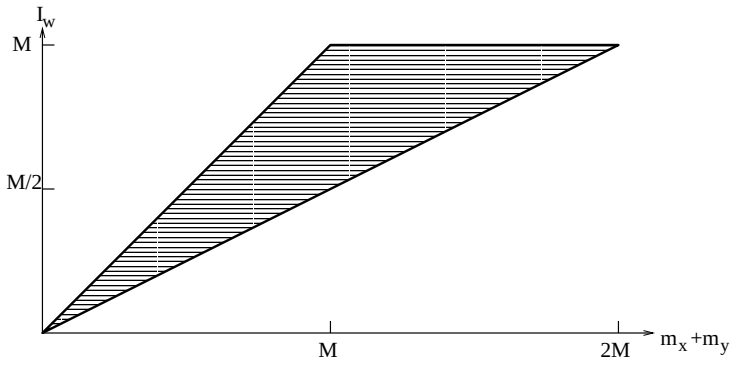


Figure 6: The range of possible values for the number of iterations through the main WHILE loop in function `generic_op`, see Listing 3.

The total execution time of algorithm `generic_op` is dominated by the number of iterations through the main while-loop. Each iteration through the loop either produces a literal word or a fill. This leads us to the following lemma about the number of iterations.

Lemma 5 *Let m_x denote the number of words in* `x.vec`*, let m_y denote the number of words in* `y.vec`*, and let M denote the number of words in a decompressed bit vector, then the number of iterations through the while-loop I_w satisfy the following conditions,*

$$\max(m_x, m_y) \leq I_w \leq \min(m_x + m_y - 1, M).$$

Proof. To prove this lemma, we observe that each iteration of the while-loop consumes one word from either operand `x` or `y`, or both of them. If each iteration consumes a word from both operands, it takes at least $\max(m_x, m_y)$ iterations. If each iteration consumes only one word from either operand, it may take $m_x + m_y$ iterations. Because the two operands contain the same number of bits, the last iteration must consume one word from each operand, the maximum number of iterations is actually $m_x + m_y - 1$. The additional upper bound comes from the fact that each iteration produces at least one word for the result and the result contains at most M words, therefore, the main loop requires at most M iterations. ∎

Figure 6 depicts the range of values for I_w pictorially. Both the upper bound and the lower bound of the ranges are achievable, however, it is less likely to reach lower bound than to reach upper bound. If for every i, `x.vec[i]` and `y.vec[i]` always represent the same number of bits, then every iteration consumes a word from each operand and $I_w = m_x = m_y$. Clearly, this could happen only in very special cases. For example if `x` and `y` are complements of each other. In more general cases, the two operands are not related like this, we expected I_w to be close to its maximum.

Theorem 6 *Using the algorithm* `generic_op` *to perform a bitwise logical operation between* `x` *and* `y`*,*

(a) the number of iterations $I_w < m_x + m_y$,

(b) and the result `z.vec` *contains at most I_w words, i.e., the number of words in* `z.vec`*, $m_z < m_x + m_y$.*

The proof of this theorem follows from the previous lemma. Using big-$\mathcal{O}$ notation, we say that both I_w and m_z have complexity of $\mathcal{O}(m_x + m_y)$.

Knowing the maximum size of the result is important. In functions `bitVector::appendLiteral` and `bitVector::appendFill`, we see that the function `vector::push_back` is used a number of

times. If we don't know the size of `z.vec`, `push_back` will have to allocate space dynamically which can be expensive. However, since we know the maximum size to expect, we can avoid dynamic memory allocation in `push_back` by reserving enough space before entering into the while-loop. This allows us to avoid dynamic memory allocation and make each invocation of `push_back` take only a constant number of clock cycles.

To compute the time complexity of `generic_op`, we need to count the number of times it invokes the various functions. It is easy to see that it invokes `run::decode` $m_x + m_y$ times because it needs to decode each word in the operands. In an iteration, it invokes either `bitVector::appendLiteral` or `bitVector::appendFill`. On most computers, the time to allocate some space is at worst linear in the number of words. Let us assume the time to allocate $m_x + m_y$ words is $C_a(m_x + m_y)$ seconds. Let C_1 denote the time required to invoke `run::decode` once, C_l denote the time required to invoke `bitVector::appendLiteral` once, C_f denote the time required to invoke `bitVector::appendFill` once. The time required by algorithm `generic_op` consists of four main terms: (1) the time to allocate space for the result z, (2) the time to invoke `run::decode`, $C_1(m_x + m_y)$, (3) the time to invoke `bitVector::appendLiteral`, $C_l\alpha I_w$, and (4) the time to invoke `bitVector::appendFill`, $C_f(1-\alpha)I_w$, where α is the fraction of iterations that generates literal words. Iterations through the main loop also involve some loop overhead which can be expressed as $C_o I_w$. Altogether, the time required by algorithm `generic_op` is

$$\begin{aligned} t_G &= C_a(m_x + m_y) + C_1(m_x + m_y) + C_l\alpha I_w + C_f(1-\alpha)I_w + C_o I_w \\ &\leq (C_a + C_1)(m_x + m_y) + (C_o + \max(C_l, C_f))I_w \\ &\leq C_2(m_x + m_y), \end{aligned} \tag{8}$$

where $C_2 = \max(C_l, C_f) + C_1 + C_o + C_a$. This proves the following theorem.

Theorem 7 *The time complexity of function* `generic_op` *is* $\mathcal{O}(m_x + m_y)$.

Both functions `bitVector::appendFill` and `bitVector::appendLiteral` use a number of conditional branching instructions (cbi) followed by a small number of other operations. On most CPUs, these conditional branching instructions will dominate the total execution time of these two functions. In Listing 1 we show the number of conditional branching instructions needed to reach the final decisions. On the average, function `bitVector::appendLiteral` uses about 4 conditional branching instructions and `bitVector::appendFill` uses about 3. This indicates that the difference between C_f and C_l is relatively small. The equality in the above expression may be achieved in many cases. Experimental results confirming this assertion are presented later.

Algorithm `generic_op` is a generic bitwise logical operation. When implementing a specific logical operation, such as AND and OR, it is easy to improve upon this algorithm. For example, in AND operations, if one of the operand is a 0-fill, we know for sure that the result is a 0-fill, if one of the operand is a 1-fill, the corresponding words in the result are the same as those in the other operand. Similar special cases can be devised for other logical operations. These special cases reduce the value of I_w by producing more than one word in each iteration of the while-loop. In addition, if we assume that both operands are properly compressed, when appending a number of consecutive literal words, we may invoke the function `bitVector::appendLiteral` only on the first word, and use `vector::push_back` on the remaining words. Since the chance of discovering fills on a set of literal words is small, this heuristic reduces the CPU time without significantly increasing the size of the result.

5.2 Complexity of algorithm inplace_or

While using a bitmap index to process a range query, we may need to OR a large number of very sparse bitmaps. If each OR operation generates a new compressed bit vector, the overhead of memory management will dominate the total execution time. To reduce the memory management

cost, we copy one of the bitmaps to a decompressed bit vector and use it to store the result of the OR operation, i.e., we perform $x \mid= y$, instead of $z = x \,|\, y$, where $|$ denotes the bitwise OR operation. In addition to avoiding repeated allocation of new memory for the intermediate results, it also removes the need to generate the words to represent 0-fills in the intermediate results.

Following the derivation of Equation 8, we can easily compute the total time required by algorithm `inplace_or`. Let m_y denote the number of words in `y.vec`, it needs to call function `run::decode` m_y times at a cost of $C_1 m_y$. If F_y denotes the length of all 1-fills in `y`, the total number of iterations through the inner loop marked "assign 1-fill" is F_y. Assume each inner iteration takes C_4 seconds, the total cost of this inner loop is $C_4 F_y$. The main loop is executed m_y times, the time spent in the main loop, excluding that spent in the inner loops, should be linear in number of iterations. This cost and the cost of decoding can be combined as $C_3 m_y$. The total time is

$$t_I = C_3 m_y + C_4 F_y. \tag{9}$$

For sparse bitmaps where $2wd_y \ll 1$, $F_y = N d_y^{2w-2}/(w-1) \to 0$. In this case, the above formula can be stated in big-$\mathcal{O}$ notation as follows.

Theorem 8 *On a sparse bitmap y, the time complexity of algorithm* `inplace_or` *is $\mathcal{O}(m_y)$.*

5.3 Dealing with a large number of bitmaps

When answering a query involving a high cardinality attribute, it is likely that we have to perform bitwise logical OR on a large number of sparse bitmaps. Such operations can be accomplished either by using `generic_op` or `inplace_or`. Here we examine two options. The first option simply takes a set of input bitmaps, OR the first two bitmaps using `generic_op`, then OR the results with the third one and so on. The second option generates a decompressed bitmap with all zero bits and use `inplace_or` to operate on the input bitmaps. There are many other possibilities, analyzing these two gives us a good indication of what can be achieved with WAH compressed bitmap indices [29].

When a query involves a large number of bitmaps, each individual bitmap must be sparse, i.e., the bit density d must be much smaller than one. When operating on sparse bitmaps, the time to invoke `generic_op` is very close to $C_2(m_x + m_y)$ and the result vector z has nearly $m_x + m_y$ regular words. Let $m_1, m_2, \ldots, m_k$ denote the sizes of the k compressed bitmaps to be ORed together, the time required to call function `generic_op` $(k-1)$ times is

$$\begin{aligned} T_G &\leq C_2(m_1+m_2) + C_2(m_1+m_2+m_3) + \ldots + C_2(m_1+m_2+\ldots+m_k) \\ &= C_2\left[\left(\sum_{i=1}^{k}(k+1-i)m_i\right) - m_1\right]. \end{aligned} \tag{10}$$

Assume all bitmaps have the same size m, the above formula simplifies to $T_G \leq C_2 m(k+2)(k-1)/2$. In other words, the total time grows super-linearly in the total size of input bitmaps (km).

To use `inplace_or`, we first need to produce a decompressed bitmap. Let C_d denote the amount of time to produce this bitmap. To complete the operations, we need to call `inplace_or` k times on k input bitmaps. The total time required is

$$T_I = C_d + C_3 \sum_{i=1}^{k} m_i + C_4 \sum_{i=1}^{k} F_i, \tag{11}$$

where F_i denotes the total length of the 1-fills in the ith bitmap. If the set of bitmaps are sparse, the term $C_4 \sum_{i=1}^{k} F_i$ in Equation 11 would be much smaller than others, which leads to $T_I \approx C_d + C_3 \sum_{i=1}^{k} m_i$. This leads to the following proposition.

Proposition 9 *When ORing a set of sparse bitmaps using* `inplace_or`*, the total time is linear in the total size of the input bitmaps.*

If all input bitmaps have the same size m, Equation 11 is linear in k, while Equation 10 is quadratic in k. This indicates that when k is large, using `inplace_or` is preferred. When operating on only two bitmaps, `generic_op` is preferred because the time required to produce a decompressed bitmap is often longer than the time to carry out the logical operation using `generic_op`. We have studied exactly when to use which algorithm and reported the results in a technical report [29]. Those interested in more details should refer to the report. Here we only repeat one key point stated in the following proposition.

Proposition 10 *Searching on one attribute using the WAH compressed bitmap index is optimal.*

A searching algorithm is considered optimal if the time complexity is linear in the number of hits H. When using the basic bitmap index to answer a query on one attribute, the result bitmap is a bitwise OR of some bitmaps from the index. The number of hits is the total number of 1s in the bitmaps involved. In cases where long query response time is expected, there must be a large number of sparse bitmaps involved. In these cases, the total size of all bitmaps is proportional to the number of 1s, see Equation 2. Using the previous proposition, the total search time using a compressed bitmap index is linear in the number of hits. Therefore, the compressed bitmap index is optimal.

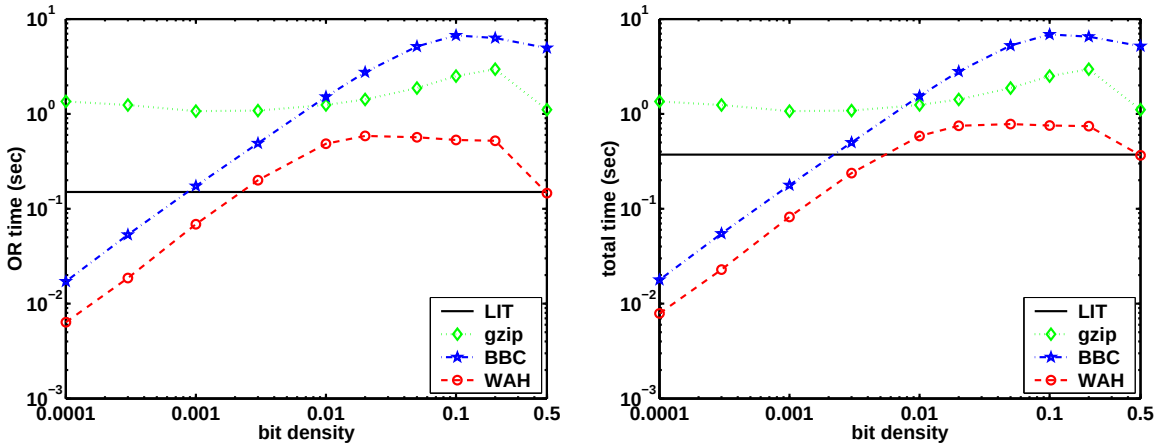
When the algorithms are implemented on computers, there are two factors that may cause the time complexity to deviate from the expected linear relation. The first one is that the "constant" C_d actually is a linear function of M. To generate a decompressed bitmap, one has to to allocate $M + 2$ words and fill them with (zero) values. Because the decompressed bitmap is generated in memory, the procedure is very fast. The observed value of C_d is typically much smaller than the total time T_I.

The total time is not strictly a linear function of H because of the memory hierarchy in most computers. Given two sets of sparse bitmaps with the same number of 1s, the procedure of using `inplace_or` is basically taking one bitmap at a time to modify the decompressed bitmap, and the content of the decompressed bitmap is modified one word at a time. In the worst case, the total number of words to be modified is H, which is the same for both sets of bitmaps. However, because the words are loaded from main memory into the caches one cache line at a time, this causes some of the words to be loaded into various levels of caches more than once. In addition, some words are loaded into caches unnecessarily. We loosely refer to these as extra work. The lower the bit density, the more extra work is required. This makes the observed value of C_3 increases as the the attribute cardinality increases. In the extreme case, H cache lines are loaded, one for each hit. In other word, the value of C_3 approaches an asymptotic maximum as the attribute cardinality increases.

6 Performance of the logical operations

In this section, we discuss the performance of the logical operations. Since logical operations are the main operations during query processing, their performance determine the overall system performance.

In our tests, we measure the performance of the WAH compression scheme along with the three schemes reviewed in Section 2. The tests are conducted on four sets of data, a set of random bitmaps, a set of bitmaps generated from a Markov process and two sets of bitmap indices from real application data. Each synthetic bitmap has 100 million bits. In our tests, the bit densities of all synthetic bitmaps are no more than 1/2. Since all compression schemes tested can compress 0-fills and 1-fills equally well, the performance for high bit density cases should be the same as their complements. One of the application dataset is from a high-energy physics experiment called



(a) logical OR time (b) total time (including I/O)

Figure 7: CPU seconds needed to perform a bitwise OR operation on two random bitmaps.

STAR [22, 23]. The data used in our tests can be viewed as one relational table consisting of about 2.2 million rows with 500 columns. The bitmaps used in this test are bitmap indices on a set of 12 most frequently queried attributes. The second application dataset is from a direct numerical simulation of the combustion process [28]. This dataset contains about 25 million rows with 16 columns.

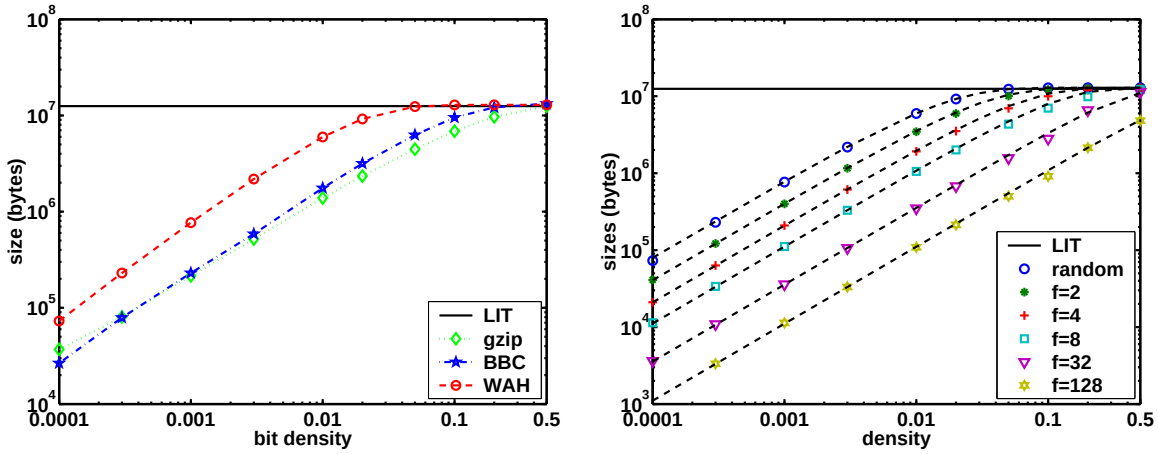
We conducted a number of tests on different machines and found that the relative performances among the different compression schemes are independent of the specific machine architecture. This was also observed in a different performance study [11]. The main reason for this is that most of the clock cycles are consumed by conditional branching instructions such as "if" tests, "switch" statements 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[7] with 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[8].

To reduce clutter, we only show the performance of logical OR operations. On the same machine, the logical AND operation and the logical XOR operation typically take about the same amount of time where XOR operation usually takes slightly more time.

The most likely scenario of using these bitmaps 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 bitmaps 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 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 have taken several times longer than those reported in this paper, because the compression process is more time-consuming [30]. The timing results for WAH and BBC are for logical operations on two compressed bitmaps that produce one compressed result, i.e., the *direct* method used by Johnson [11].

Figure 7 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 7(a) shows the logical operation time and Figure 7(b) shows the total time including the time to read the two bitmaps from files. Because the OS has cached some information about the files, the I/O time is a relatively small portion of the total time. In a database system,

[7]Information about the E450 is available at `http://www.sun.com/servers/workgroup/450`.
[8]Information about VERITAS Volume Manager is available at `http://www.veritas.com/us/products`.



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

Figure 8: The sizes of the compressed bit vectors. The symbols for the Markov bitmaps are marked with their clustering factors. The dashed lines are predictions based on Equations 1 and 3.

commonly used bitmaps would also be cached in memory and I/O cost would be invisible to most users. To simplify the discussions, we will not include the I/O time in other figures of this section. A more careful study of I/O performance is reported elsewhere [24].

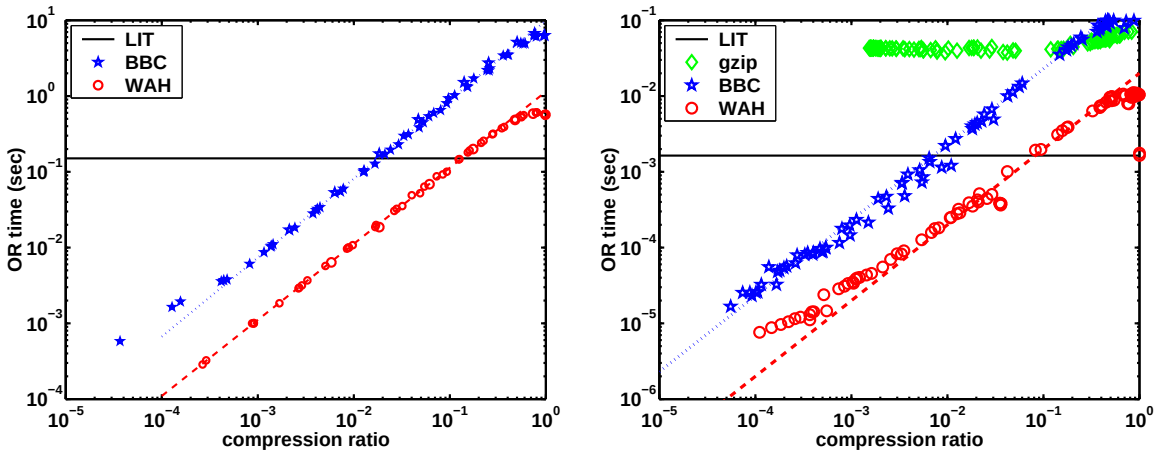
Among the schemes shown, it is clear that WAH uses much less time than either BBC or gzip. In all cases, gzip uses at least three times the time used by the literal scheme. In four out of ten test cases, BBC takes more than ten times longer than WAH.

When the bit density is 1/2, the random bitmaps are not compressible by WAH. In this case, our implementation of WAH performs logical operations as fast as the literal scheme. In Figure 7, the line for WAH falls on top of the one for the literal scheme at bit density of 1/2.

Figure 8 shows the sizes of the four types of bit vectors. Each data point in this figure represent the average size of a number of bitmaps with the same bit density and clustering factor. The dashed lines in Figure 8 are the expected sizes according to Equations 1 and 3. It is clear that the actual sizes agree with the predictions.

As the bit density increases from 0.0001 to 0.5, the bitmaps become less compressible and it takes more space to represent them. When the bit density is 0.0001, all three 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 indices compressed with WAH are about 60% bigger than those compressed with BBC, see Figure 10.

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 9(a) and the results on the STAR bitmaps in Figure 9(b). In both cases, the compression ratio is shown as the horizontal axes. Since in each plot, the bitmaps are of the same length, the average compression ratio is proportional to the total size of the two operands of a logical operation. In each plot, a symbol represents the average time of logical operations on bitmaps with the same sizes. The dashed and dotted lines are



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

Figure 9: Logical operation time is almost proportional to compression ratio. The STAR bitmap indices are on the 12 most queried attributes.

produced from linear regressions. Most of the data points near the center of the graphs are close to the regression lines, which verifies that the time is proportional to the sizes of the operands. As expected, logical operations on bitmaps with compression ratios near one approach constant. For very small bit vectors, where the logical operation time is measured to be a few microseconds, the measured time deviates from the linear relation because of factors such as the timing overhead and function call overhead. 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 factor 3 discussed in the previous section, see page 12. More specifically, 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 can be nearly 100 times faster than BBC. On very sparse bit vectors, WAH is about four to five times faster than BBC when bitmaps are in memory, WAH is about two to three times faster than BBC when both operands have to be read from disk.

Compared with 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 that the decompressed form of WAH be used.

7 Performance on range queries

This research was originally motivated by the need to manage the volume of data produce by the STAR experiment. In this case, the queries on the data are range queries of the form $x_1 \leq X < x_2$, where X is an attribute name, x_1 and x_2 are two constants. In this section, we measure the time used to answer random range queries, where the values of x_1 and x_2 are randomly chosen[9] within the domain of X. Bitmap indices is known to support multidimensional queries well, in this paper, we concentrate on one dimensional queries. In the following tests, the measured query response time which includes the time to read the necessary bitmaps into memory, perform bitwise logical operations, parse the queries, manage the locks and so on.

[9]If $x_1 > x_2$, we swap the two. If $x_1 = x_2$, the range is taken to be $x1 \leq X$.

		commercial DBMS		our bitmap indices		
	projection	B-tree	bitmap	LIT	BBC	WAH
size (MB)	113	408	111	726,484	118	186
time (sec)	0.57	0.95	0.66	-	0.32	0.052

Figure 10: Total sizes of the indices on the 12 commonly queried attributes of STAR data and the average time needed to process a random range query.

The table in Figure 10 shows the summary about the various indices on the STAR data. We consider the projection index as the reference method since it is simple and efficient [18]. On this set of data, our implementation of the projection index takes about 0.57 seconds to answer a range query, which translates to a reading speed of about 16 MB/s. We also stored the same data in a commercial DBMS. It takes about 113 MB of space to store the relation. With a B-tree index for each attribute, the total size is nearly four times as large as the data. The average time for answering the same set of range queries using B-tree indices is about 0.95 seconds. We also tested the bitmap index available in the commercial system. Compared to our own implementation of the BBC compressed index, this commercial implementation takes slightly less space but uses about twice the time to process the same queries. With the literal (LIT) scheme, the bitmap index would use more than 726 GB of space and is clearly impractical in this case.

On this set of STAR data, the WAH compressed bitmap index is about 60% larger than the BBC compressed index. In terms of query processing time, the WAH compressed indices are about 13 times faster than the commercial implementation of the compressed index, and about six times faster than our own version of the BBC compressed index. In the previous section, we observed that WAH performs bitwise logical operations about 12 times as fast as BBC. We only see a factor of six improvement in query response time because of two main reasons. WAH compressed bitmaps are typically larger than those compressed with BBC, therefore it takes longer to read WAH compressed bitmaps. While answering queries, a large number of test cases involve sparse bitmaps, where WAH may only be twice as fast as BBC.

The bitmap index is often considered efficient for low cardinality attributes. For scientific data like the one from STAR, where the cardinalities of some attributes are in the millions, there are doubts as to whether it is still efficient. Our analyses and this example demonstrate that the compressed bitmap index is efficient. The main reason for this is that the size of the bitmap index is near its stable plateau when the cardinality is large. Since the logical operation time is proportional to the total size of the compressed bitmaps involved, this limit in size also implies a limit on the query processing time. To verify that the query processing time using the WAH compressed bitmap indices indeed scales linearly as the index sizes, we plotted the timing results from all four datasets in Figure 11.

All four datasets are used in Figure 11. Each dot or a symbol represents the average query processing time on queries involving one attribute from the datasets. The dataset marked STAR is the smallest dataset with 2.2 million rows, the combustion dataset has about 25 million rows, and the two synthetic datasets consist of 100 million rows of random data with various distributions. To illustrate how WAH compressed indices might behave on even larger datasets, we drew two lines to represent what can be expected in the average case and in the worst case. In the worst cases, about half of the bitmaps in a bitmap index is read into memory and operated on. In these cases, the total query response time should be more closely related to the total sizes of the bitmap indices. The points in Figure 11(b) are indeed close to the trend line. The average query response time in Figure 11(a) has a large scatter because it depends more on the distribution of the data.

Next we make some quantitative observations based on Figure 11. Let S denote the index size in bytes, the average query response time is expected to be $10^{-8}S$ seconds on the particular test machine, and the maximum query response time is $3.5 \times 10^{-8}S$ seconds. From earlier tests, see

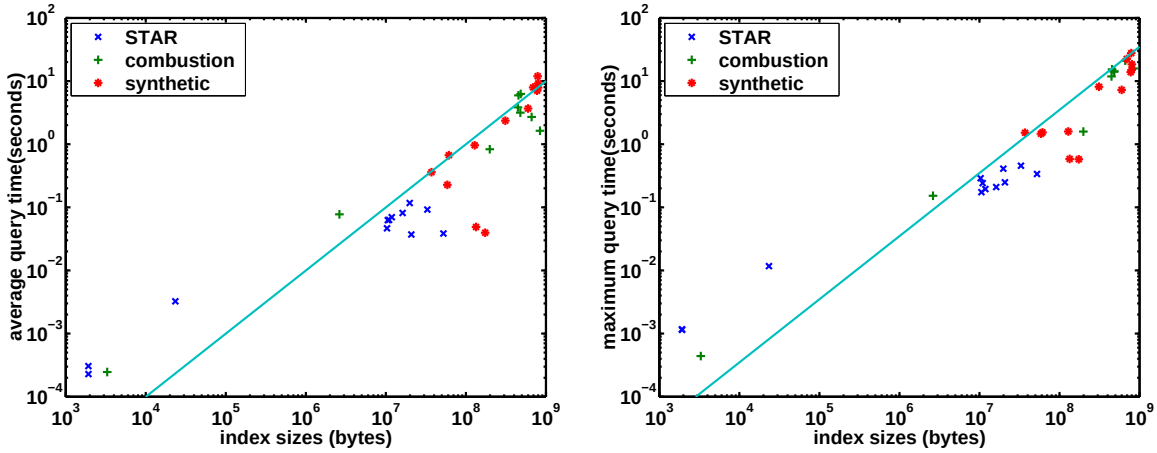
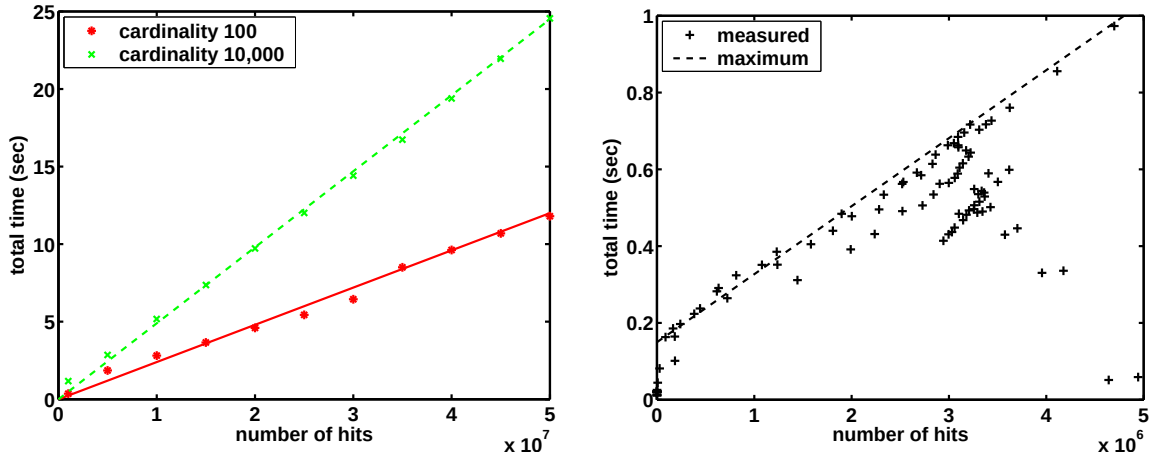


Figure 11: The average time and the maximum time needed to process a random range query using a WAH compressed bitmap index.



(a) Uniform random data (b) combustion data

Figure 12: Query processing time using compressed bitmap indices against the number of hits.

Figure 10, we know that the average reading speed is about 16 MB/s. If the query response time is spent to only read bitmap indices, the measured worst case time can be used to read about half of the compressed bitmaps[10]. For high cardinality attributes, the WAH compressed bitmap index is about twice as large as the projection index. This suggests that the maximum query response time is close to the time needed by the projection index. The average query response time is about a third of the maximum query response time and also about a third of the time required by the projection index.

We have observed that the query response time using BBC compressed indices is six time as long as that using WAH compressed indices. The above comparison suggests that the query response time using BBC compressed indices can be twice as long as that using the projection indices. Overall, our tests indicate that WAH compressed bitmap indices are indeed efficient for high cardinality attributes.

Next we verify the proposed linearity between the search time and number of hits, see Proposition 10. Figure 12 shows the time required to answer some random queries on two sets of data,

[10]More precisely 56%, which can be computed as follows. Let β be the fraction of bitmaps read, the time to read it at 16 MB/s is $\frac{\beta S}{16\times 10^6}$ seconds. The measured time is $3.5\times 10^{-8}S$ seconds. The two time values must be the same, $\frac{\beta S}{16\times 10^6} = 3.5\times 10^{-8}S$, which leads to $\beta = 16\times 10^6\times 3.5\times 10^{-8} = 0.56$.

Figure 12(a) for uniform random data and Figure 12(b) for combustion data. The bitmaps for uniform random data are very close the worst case, therefore, the points follow the expected straight lines. In general, the straight lines are expect for the maximum time, we see that this is the case for the combustion data. The slopes of the lines, C_3, show some dependencies on the characteristics of data. For uniform random data, an attribute with a large cardinality also has a larger C_3. In Figure 12(a), C_3 for the attribute with $c = 10,000$ is about twice as large as that for the attribute with $c = 100$. The slope for the maximum time in Figure 12(b) is slightly smaller than that for the random attribute with $c = 100$. This verifies that the worst case query processing time is indeed linear in the number of hits. Therefore, the WAH compressed bitmap index is optimal for searching on one attribute.

8 Summary

It is well accepted that I/O cost dominates the query response time when using out-of-core indexing methods. Thus, most indexing techniques focus on minimizing I/O cost. For bitmap indices, most research efforts concentrate on reducing the sizes of indices. However, tests show that the computation time can dominate the total time when using compressed bitmap indices [24]. In addition, as main memories become cheaper, we expect that "popular" bitmaps would remain in memory. This would further reduce the average I/O cost and make the time spent in CPU more prominent part of the total query processing time. For these reasons, we seek to improve the computational efficiency of operations on compressed bitmaps. The best existing bitmap compression schemes are byte-aligned. In this paper, we have presented a word-aligned scheme (WAH) that is not only much simpler but is also more CPU-friendly. This ensures that the logical operations are performed efficiently. Tests on a STAR dataset show that it is 12 times faster than BBC while using only 60% more space.

The bitmap index is often considered efficient for low cardinality attributes. Our analyses show that WAH compression is effective in reducing the bitmap index size. For an attribute with vary high cardinality, say millions, its bitmap index takes roughly two words per attribute value. This is about half the size of a typical B-tree index. We have also shown through both analyses and tests that the query processing time grows linearly as the index size increases. In addition, we demonstrated that the query processing time is linear in the number of hits when using a WAH compressed bitmap index. This proves that WAH compressed bitmap indices are optimal.

Given the same attribute cardinality, bitmap indices of uniform random attributes are larger than bitmap indices of others, and it also takes longer to search the random attributes. On these random attributes, WAH compressed indices are measured to be three times as fast as the projection indices. On real world data, such as those from the STAR experiment, the compressed bitmap indices are even more efficient. On the STAR dataset, the WAH compressed bitmap index is about six times faster than our own version of the BBC compressed bitmap index, and about 13 times faster than a commercial implementation of the compressed bitmap index. It is about 11 times faster than the projection index, and nearly 20 times faster than the B-tree index. WAH is indeed a very efficient compression method for bitmap indices.

References

[1] Sihem Amer-Yahia and Theodore Johnson. Optimizing queries on compressed bitmaps. In A. El Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. 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. *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. Document available at `http://esdis-it.gsfc.nasa.gov/MSST/conf2000/PAPERS/B05PA.PDF`.

[5] 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*, pages 355–366. ACM press, 1998.

[6] 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*, pages 215–226. 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] 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.

[9] 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`.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Volker Markl, Martin Zirkel, and Rudolf Bayer. Processing operations with restrictions in RDBMS without external sorting: The tetris algorithm. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 562–571. IEEE Computer Society, 1999.

[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 *Lecture Notes in Computer Science*, pages 40–59, 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] Patrick O'Neil and Elizabeth O'Neil. *Database: pronciples, programming, and performance.* Morgan Kaugmann, 2nd edition, 2000.

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

[22] 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.

[23] 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.

[24] Kurt Stockinger, Kesheng Wu, and Arie Shoshani. Strategies for processing ad hoc queries on large data warehouses. In *Proceedings of DOLAP'02*, 2002.

[25] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In J. Widom, A. Gupta, and O. Shmueli, editors, *VLDB'98, Proceedings of 24th International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 194–205. Morgan Kaufmann, 1998.

[26] 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.

[27] 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.

[28] Kesheng Wu, Wendy Koegler, Jacqueline Chen, and Arie Shoshani. Using bitmap index for interactive exploration of large datasets. In *Proceedings of SSDBM 2003*, pages 65–74, 2003. A draft appeared as tech report LBNL-52535.

[29] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. On the performance of bitmap indices for high cardinality attributes. Technical Report LBNL-54673, Lawrence Berkeley National Laboratory, Berkeley, CA, 2004.

[30] 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.

[31] 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.

[32] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Appendix: Algorithms to perform logical operations

LISTING **1** Data structure (classes) to store the WAH compressed bitmaps.

```
class bitVector {
      std::vector<unsigned> vec; // list of regular code words
      activeWord active; // the active word
      class activeWord {
            unsigned value; // the literal value of the active word
            unsigned nbits; // number of bits in the active word
      };
};
```

bitVector::appendLiteral() {
Input: 31 literal bits stored in `active.value`.
Output: `vec` extended by 31 bits.

```
      IF (vec.empty())
            vec.push_back(active.value);                    // cbi = 1
      ELSEIF (active.value == 0)
            IF (vec.back() == 0)
                  vec.back() = 0x80000002;                  // cbi = 3
            ELSEIF (vec.back() ≥ 0x80000000 AND vec.back() < 0xC0000000)
                  ++vec.back();                             // cbi = 4
            ELSE
                  vec.push_back(active.value)               // cbi = 4
      ELSEIF (active.value == 0x7FFFFFFF)
            IF (vec.back() == active.value)
                  vec.back() = 0xC0000002;                  // cbi = 4
            ELSEIF (vec.back() ≥ 0xC0000000)
                  ++vec.back();                             // cbi = 5
            ELSE
                  vec.push_back(active.value);              // cbi = 5
      ELSE
            vec.push_back(active.value);                    // cbi = 3
}
```

bitVector::appendFill(`n, fillBit`) {
Input: `n` and `fillBit`, describing a fill with `31n` bits of `fillBit`
Output: `vec` extended by `31n` bits of value `fillBit`.
COMMENT: Assuming `active.nbits = 0` and `n` > 0.

```
      IF (n > 1 AND ! vec.empty())
```

```
        IF (fillBit == 0)
                IF (vec.back() ≥ 0x80000000 AND vec.back() < 0xC0000000)
                        vec.back() += n;                 // cbi = 3
                ELSE
                        vec.push_back(0x80000000 + n);   // cbi = 3
        ELSEIF (vec.back() ≥ 0xC0000000)
                vec.back() += n;                         // cbi = 3
        ELSE
                vec.push_back(0xC0000000 + n);           // cbi = 3
    ELSEIF (vec.empty())
        IF (fillBit == 0)
                vec.push_back(0x80000000 + n);           // cbi = 3
        ELSE
                vec.push_back(0xC0000000 + n);           // cbi = 3
    ELSE
        active.value = (fillBit?0x7FFFFFFF:0),           // cbi = 3
        appendLiteral();
    }
```

LISTING **2** An auxiliary data structure used in Listings 3 and 4.

```
class run { // used to hold basic information about a run
    std::vector<unsigned>::const_iterator it;
    unsigned fill; // one word-long version of the fill
    unsigned nWords; // number of words in the run
    bool isFill; // is it a fill run
    run() :  it(0), fill(0), nWords(0), isFill(0) {};
    decode() { // Decode the word pointed by iterator it
        IF (*it > 0x7FFFFFFF)
            fill = (*it≥0xC0000000?0x7FFFFFFF:0), // cbi = 2
            nWords = *it & 0x3FFFFFFF,
            isFill = 1;
        ELSE
            nWords = 1,                           // cbi = 1
            isFill = 0;
    }
};
```

LISTING **3** Given two bitVector x and y, perform an arbitrary bitwise logical operation (denoted by $\circ$, can be any binary logical operator, such as, AND, OR, and XOR) to produce a bit vector z.

```
z = generic_op(x, y) {
```
Input: Two bitVector x and y containing the same number of bits.
Output: The result of a bitwise logical operation as z.
```
    run xrun, yrun;
    xrun.it = x.vec.begin(); xrun.decode();
    yrun.it = y.vec.begin(); yrun.decode();
    WHILE (x.vec and y.vec are not exhausted) {
        IF (xrun.nWords == 0) ++xrun.it, xrun.decode();
        IF (yrun.nWords == 0) ++yrun.it, yrun.decode();
        IF (xrun.isFill)
            IF (yrun.isFill)
                nWords = min(xrun.nWords, yrun.nWords),
                z.appendFill(nWords, (*(xrun.it) ∘ *(yrun.it))),
                xrun.nWords -= nWords, yrun.nWords -= nWords;
            ELSE
                z.active.value = xrun.fill ∘ *yrun.it,
```

```
                        z.appendLiteral(),
                        -- xrun.nWords;
            ELSEIF (yrun.isFill)
                  z.active.value = yrun.fill ∘ *xrun.it,
                  z.appendLiteral(),
                  -- yrun.nWords;
            ELSE
                  z.active.value = *xrun.it ∘ *yrun.it,
                  z.appendLiteral();
      }
      z.active.value = x.active.value ∘ y.active.value;
      z.active.nbits = x.active.nbits;
}
```

LISTING **4** Given two bitVector x and y, perform a bitwise logical OR operation. The bit vector x is assumed to be decompressed and the result is written back to x.

inplace_or(x, y) {
Input: Two bitVector x and y containing the same number of bits and x is decompressed.
Output: The result of a bitwise logical OR operation stored in x.

```
      run yrun;
      yrun.it = y.vec.begin();
      std::vector<unsigned>::iterator xit = x.vec.begin();
      WHILE (y.vec is not exhausted) {
            yrun.decode();
            IF (yrun.isFill)
                  IF (yrun.fill == 0)
                        *xit += yrun.nWords;
                  ELSE {
                        std::vector<unsigned>::iterator stop = xit;
                        stop += yrun.nWords;
                        for (; xit < stop; ++ xit) // assign 1-fill
                              *xit = 0x7FFFFFFF;
                  }
            ELSE
                  *xit |= *yrun.it,
                  ++ xit;
      ++ yrun.it;
      }
      x.active.value |= y.active.value;
};
```