doi>10.1145/1132863.1132864

Optimizing Bitmap Indices With Efficient Compression

KESHENG WU, EKOW J. OTOO and ARIE SHOSHANI
Lawrence Berkeley National Laboratory

Bitmap indices are efficient for answering queries on low cardinality attributes. In this paper, we present a new compression scheme called Word-Aligned Hybrid (WAH) code that makes compressed bitmap indices efficient even for high cardinality attributes. We further prove that the new compressed bitmap index, like the best variants of the B-tree index, is optimal for one-dimensional range queries. More specifically, the time required to answer a one-dimensional range query is a linear function of the number of hits. This strongly supports the well-known observation that compressed bitmap indices are efficient for multi-dimensional range queries because results of one-dimensional range queries computed with bitmap indices can be easily combined to answer multi-dimensional range queries. Our timing measurements on range queries not only confirm the linear relationship between the query response time and the number of hits, but also demonstrate that WAH compressed indices answer queries faster than the commonly used indices including projection indices, B-tree indices, and other compressed bitmap indices.

Categories and Subject Descriptors: E.4 [**Data**]: Coding and Information Theory—*Data compaction and compression*; H.3.1 [**Information Systems**]: Content Analysis and Indexing—*Indexing methods*

General Terms: Performance, Algorithms

Additional Key Words and Phrases: Compression, bitmap index, query processing

1. INTRODUCTION

Bitmap indices are known to be efficient, especially for read-mostly or append-only data. Many researchers have demonstrated this [O'Neil 1987; Chaudhuri and Dayal 1997; Jürgens and Lenz 1999]. Major DBMS vendors including ORACLE, Sybase and IBM have implemented them in their respective DBMS products. However, users are usually cautioned not to use them for high cardinality attributes. In this paper, we present an efficient compression scheme, called Word-Aligned Hybrid (WAH) code that, not only reduces the index sizes but also guarantees a theoretically optimal query response time for one-dimensional range queries. A number of

This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.
Copyright 2005 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to Post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

row ID	**X**	bitmap index b_0 =0	b_1 =1	b_2 =2	b_3 =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

Fig. 1. A sample bitmap index. Each column $b_0 \ldots b_3$ is called a bitmap in this paper.

empirical studies have shown that WAH compressed bitmap indices answer queries faster than uncompressed bitmap indices, projection indices, and B-tree indices, on both high and low cardinality attributes [Wu et al. 2001; 2002; Stockinger et al. 2002; Wu et al. 2004]. This paper complements the observations with rigorous analyses. The main conclusion of the paper is that the WAH compressed bitmap index is in fact optimal. Some of the most efficient indexing schemes such as B$^+$-tree indices and B*-tree indices have a similar optimality property [Comer 1979; Knuth 1998]. However, a unique advantage of compressed bitmap indices is that the results of one-dimensional queries can be efficiently combined to answer multi-dimensional queries. This makes WAH compressed bitmap indices well-suited for ad hoc analyses of large high-dimensional datasets.

1.1 The basic bitmap index

In this paper, we define a bitmap index to be an indexing scheme that stores the bulk of its data as bit sequences and answers queries primarily with bitwise logical operations. We refer to these bit sequences as *bitmaps*. Figure 1 shows a set of such bitmaps for an attribute **X** of a hypothetical table (**T**), consisting of eight tuples (rows). The cardinality of **X** is four. Without loss of generality, we label the four values as 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 $b_0, \ldots, b_3$. When processing the query "`select * from T where X < 2`," the main operation on the bitmaps is the bitwise logical operation "b_0 OR b_1." Since bitwise logical operations are well supported by the computer hardware, bitmap indices are very efficient [O'Neil 1987].

The earlier forms of bitmap indices were commonly used to implemented inverted files [Knuth 1998; Wong et al. 1985]. The first commercial product to make extensive use of the bitmap index was Model 204 [O'Neil 1987]. In many data warehouse applications, bitmap indices perform better than tree-based schemes, such as the variants of B-tree or R-tree [Jürgens and Lenz 1999; Chan and Ioannidis 1998; O'Neil 1987; Wu and Buchmann 1998]. According to the performance model proposed by Jürgens and Lenz [1999], 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 [O'Neil

and Graefe 1995].

The simple bitmap index shown in Figure 1 is known as the *basic bitmap index*. The basic bitmap index can be built for integer attributes as well as floating-point values and strings. The main practical difference is that floating-point attributes typically have more distinct values, and therefore their bitmap indices require more bitmaps.

There is only one attribute in the above 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 rows 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 intermediate bitmaps.

1.2 Bitmap compression

The above procedure is efficient if indices for both `Energy` and `NumParticles` have only a small number of bitmaps. However, in many applications, especially scientific applications, attributes like `NumParticles` and `Energy` often have thousands or even millions of distinct values. A basic bitmap index for such an attribute would contain thousands or millions of bitmaps. The index sizes could be very large. The author of Model 204 pointed out this problem and suggested a compact row identifier (RID) list as an alternative [O'Neil 1987]. Since using a RID list makes it harder to answer multi-dimensional queries, we choose to compress the bitmaps to reduce the index size instead.

To compress a bitmap, a simple option is to use a text compression scheme, such as LZ77 (used in gzip) [Gailly and Adler 1998; Ziv and Lempel 1977]. These schemes are 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 before any operation. To illustrate the importance of efficient logical operations, assume that the attribute `NumParticles` can have integer values from 1 to 10,000. Its bitmap index would have 10,000 bitmaps. To answer a query involving "`NumParticles > 5000`," 5000 bitmaps have to be OR'ed together. To efficiently answer this query, it is not sufficient that the bitmaps are small, the operations on them must be fast as well.

To improve the performance of bitwise logical operations, a number of specialized schemes have been proposed. Johnson and colleagues have thoroughly studied many of these schemes [Johnson 1999; Amer-Yahia and Johnson 2000]. From their studies we know that the logical operations with these specialized schemes are usually faster than those with LZ77. One such specialized scheme, called the *Byte-aligned Bitmap Code* (BBC), is especially efficient [Antoshenkov 1994; Antoshenkov and Ziauddin 1996]. However, in the worst case, the total time required to perform a logical operation on two BBC compressed bitmaps can still be 100 times longer than on two uncompressed bitmaps as shown later in Figure 9.

We have developed a number of compression schemes that improve the overall query response time by improving their worst case performance [Wu et al. 2001; Wu et al. 2001]. In this paper, we concentrate on the Word-Aligned Hybrid (WAH)

code for two main reasons: (1) it is the easiest to analyze, which leads us to prove an important optimality about the compressed bitmap indices, and (2) it is the fastest in our tests. In earlier tests, we observed that bitwise logical operations on WAH compressed bitmaps are 2 to 100 times faster than the same operations on BBC compressed bitmaps because WAH is a much simpler compression method than BBC [Wu et al. 2001; Stockinger et al. 2002].

There is a space-time trade-off among these compression schemes. Comparing BBC with LZ77, BBC trades some space for more efficient operations. Similarly, WAH trades even more space for even faster operations.

Compressing individual bitmaps is only one way to reduce the bitmap index size. An alternative strategy is to reduce the number of bitmaps, 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 [Koudas 2000; Shoshani et al. 1999; Wu and Yu 1996]. Many researchers have studied the strategy of using different encoding schemes to reduce the index sizes [Chan and Ioannidis 1998; 1999; O'Neil and Quass 1997; Wong et al. 1985; Wu and Buchmann 1998]. 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 [O'Neil and Quass 1997]. This is referred to as the binary encoding scheme elsewhere [Wong et al. 1985; Chan and Ioannidis 1998; Wu and Buchmann 1998]. A drawback of this scheme is that most of the bitmaps have to be accessed when answering a query. To answer a 2-sided range query such as "`120 < Energy < 140`", most bitmaps have to be accessed twice. 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 [Chan and Ioannidis 1998], interval encoding [Chan and Ioannidis 1999] and the K-of-N encoding [Wong et al. 1985]. In all these schemes, an efficient compression scheme should improve their effectiveness. Additionally, a number of common indexing schemes such as the signature file [Furuse et al. 1995; Ishikawa et al. 1993; Lee et al. 1995] may also benefit from an efficient bitmap compression scheme. Compressed bitmaps can also be effective for purposes other than indexing. In one case, we demonstrated that using compressed bitmaps significantly speeds up the tracking of spatial features as they evolve in the simulation of a combustion process [Wu et al. 2003].

1.3 Main results of this paper

The main objective of using an index in a database system is to reduce the query response time. To achieve this objective, it is important to reduce both the I/O time and the CPU time required to answer a query. Since the I/O time is known to be a linear function of the index size, in our analyses we concentrate on the index size instead of the I/O time. In addition to analyzing the index size, we also analyze the CPU time required to answer a query. To verify the analyses, we measure the index sizes and query response time using an implementation for read-only data. An implementation that accommodates frequent changes such as updates is likely to introduce extra overhead compared with that for read-only data. In this paper, we focus on the performance of bitmap indices without considering the update overhead and leave the topic for the future.

We prove that the index size of an attribute with uniform random data distri-

bution is the largest among all attributes with the same cardinality. In the worst case, each bit that is 1 requires two words in the compressed bitmaps, one for storing the bit that is 1, and one for storing the 0s separating two nearby 1's. For a table with N rows, the basic bitmap index illustrated in Figure 1 for any attribute contains exactly N bits that are 1. Therefore, the maximum size of a WAH compressed bitmap index is $2N$ words plus some overhead that is proportional to the number of bitmaps used. In most cases, the overhead is much less than $2N$ words. Compared with the commonly used B^+-tree index, this maximum size is relatively small. For example, in a commercial DBMS that we use regularly, a B^+-tree index is observed to use $3N \sim 4N$ words.

For answering ad hoc queries, projection indices are known to be the most efficient [O'Neil and Quass 1997]. A projection index projects out a particular attribute of a relation, so that all its values can be stored together to provide more efficient data accesses. If each value fits in a word, then a projection index uses exactly N words. When answering a one-dimensional range query, we never need to access more than half of the bitmaps in a bitmap index, and therefore the maximum number of words accessed using a WAH compressed bitmap index is also N words. Just considering the I/O time, using a WAH compressed bitmap index never needs more time than using the corresponding projection index. Indeed, considering I/O alone is sufficient since WAH is highly compute efficient and the total query response time using a WAH compressed bitmap index is I/O dominated [Stockinger et al. 2002]. This suggests that using the WAH compressed bitmap index to answer a user query should be faster than using the projection index. In this paper, we further confirm this advantage of the WAH compressed index in Section 7.2.

In addition to analyzing uniform random attributes, we also analyze attributes generated by a Markov process. Such attributes typically reflect application data better than uniform random attributes. For example, their compressed index sizes, like those from application data, can be much smaller than those of uniform random attributes. Our analysis utilizes a novel technique that makes it possible to compute the expected size of WAH compressed bitmaps generated from both uniform random attributes and attributes generated by a Markov process. The new analysis not only accurately predicts the maximum index size but also indicates what type of attributes leads to larger indices. In contrast, our previous attempt to analyze the index size only yielded an upper bound with no indication whether the upper bound was achievable, or how to achieve it if it is achievable [Wu et al. 2004].

In earlier performance tests, we observed that the time to perform a logical operation on two compressed bitmaps is proportional to the total size of the two input bitmaps [Wu et al. 2001; 2002; Stockinger et al. 2002]. In this paper, we confirm this observation by a formal analysis of the time spent in any bitwise logical operation (including AND, OR, XOR) on two compressed bitmaps. However, to answer a range query, it is often necessary to OR a large number of bitmaps. We also prove that the time required by a special OR algorithm for multiple bitmaps from a bitmap index is a linear function of the total size of all bitmaps involved.

The time to evaluate a one-dimensional range condition is mostly spent in OR'ing the bitmaps. Since the time to complete the OR operations is a linear function of the total size of bitmaps involved, and the total size is at worst a linear function of

number of 1's in all the bitmaps, the query response time is at worst a linear function of the number of 1's in the bitmaps, which is the number of hits. By definition, this query response time is optimal. Numerous timing results are presented in Section 7 of this paper to substantiate the analyses and to compare WAH compressed index against well-known indices including the projection index and the B$^+$-tree index.

1.4 Outline

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 the analysis of bitmap index sizes, and Section 5 presents the analysis of time complexity to answer a range query. We present some performance measurements in Section 6 and 7 to support the analyses. A short summary is given in Section 8.

2. RELATED WORK ON BYTE-BASED SCHEMES

In this section, we briefly review three well-known schemes for representing bitmaps and introduce the terminology needed to describe our new scheme. These three schemes are selected as representatives of a number of different schemes [Johnson 1999; Wu et al. 2001].

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

The second scheme in our comparisons is the general purpose compression scheme based on LZ77 called gzip [Gailly and Adler 1998]. The general purpose schemes are highly efficient in compressing text files. We use gzip as the representative because it is usually faster than others in decompressing files [Wu et al. 2001].

There are a number of compression schemes that offer good compression and also support fast bitwise logical operations. The Byte-aligned Bitmap Code (BBC) is a well-known scheme of this type [Antoshenkov 1994; Johnson 1999]. BBC performs bitwise logical operations efficiently and it compresses almost as well as gzip. The BBC implementation used in later performance measurements is a version of the 2-sided BBC code [Wu et al. 2001, § 3.2]. In an earlier comparison against another implementation 2-sided BBC derived from Johnson's 1-sided BBC [Johnson 1999], our implementation takes an average of 5% more space but reduces the average query response time by about 1/3 [Stockinger et al. 2002]. Because of this performance advantage, we choose to use our implementation of BBC for comparison with WAH. To better understand its performance advantages and to see how we might improve it, we next review the key characteristics of BBC.

The first reason that bitwise logical operations on BBC compressed data are more efficient than on gzip compressed data is that BBC is a lot simpler. Like many of the specialized bitmap compression schemes, BBC is based on the simple idea of run-length encoding. It represents 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 0, we call the fill a *0-fill*, otherwise it is a *1-fill*. Different run-length encoding schemes generally use different strategies to represent run lengths.

Typically, if a fill is short, say one-bit long, the run-length encoding may use more space than the literal representation. Different run-length encoding schemes usually differ on what is considered short. However, if a fill is considered short, it is usually stored literally. This is a practical way to improve the basic run-length encoding scheme.

Given a bitmap, the BBC compression scheme first divides it into bytes and then groups the bytes into *runs*. Each BBC run consists of a fill followed by a *tail*. A BBC fill is a consecutive group of bytes where the bits are either all 0 or all 1, and a tail is a consecutive group of bytes with a mixture of 0 and 1 in each byte. Since a BBC fill always contains a number of whole bytes, it represents the fill length as the number of bytes rather than the number of bits. This reduces the space required for storing the run lengths. In addition, it uses a multi-byte scheme to further limit the space requirement [Antoshenkov 1994; Johnson 1999]. This strategy often uses more bits to represent a fill length than others such as ExpGol [Moffat and Zobel 1992]. However it allows for faster operations [Johnson 1999].

BBC leaves fills shorter than a byte in tail bytes. This causes all encoded bits to be byte-aligned and makes it more efficient to access the encoded bitmaps. More importantly, during any bitwise logical operations, a tail byte is never broken into individual bits. Because working on whole bytes is usually more efficient than working on individual bits, this byte alignment property makes BBC more efficient than others techniques such as ExpGol.

3. WORD-ALIGNED BITMAP COMPRESSION SCHEME

BBC supports more efficient bitwise logical operations than other compression schemes such as LZ77 because it uses a simpler compression algorithm and it obeys byte alignment [Johnson 1999; Wu et al. 2001; Stockinger et al. 2002]. Since most general-purpose CPUs can operate on a word more efficiently than on a byte, a compression scheme that observes word alignment may perform even better. Following this observation, we extended BBC to be word-aligned rather than byte-aligned, and developed the Word-aligned Bitmap Code (WBC) [Wu et al. 2001]. Similar to BBC, WBC divides a bitmap into runs, where each run is a fill followed by some tail words. Each WBC run requires at least one header word. Since a header word can contain more information than a BBC header byte, we choose to use the simplest option of having only one type of WBC run instead of four or more types in BBC. This simplifies WBC and our tests show that WBC performs logical operations two to four times faster than BBC [Wu et al. 2001].

The fastest compression scheme we developed is the Word-Aligned Hybrid (WAH) code. In our tests that compare WBC and an early version WAH, we observed that they used about the same amount of space but WAH can perform logical operations faster than WBC, in many cases by a factor of two [Wu et al. 2001, Figures 16 – 18]. We believe that much of this performance difference is due to the dependency among the words in WBC. In particular, a WBC header word has to be fully decomposed before the tail words and the next header word can be processed. This causes the CPU pipelines to stall and increases the total processing time. Because of this, we have chosen to only analyze WAH in this paper. For performance comparisons, we will compare against the well-known BBC rather than our own WBC.

128 bits	1*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
literal (hex)	40000380	00000000 00000000	001FFFFF	0000000F
WAH (hex)	40000380	80000002	001FFFFF	0000000F

Fig. 2. A WAH compressed bitmap. Each WAH word (last row) represents a multiple of 31 bits from the input bitmap, except the last word that represents the four leftover bits.

Similar to WBC and BBC, WAH is also based on the basic idea of run-length encoding. However, unlike WBC or BBC, WAH does not use any header words or bytes, which removes the dependency mentioned above. In BBC, there are at least four different types of runs and a number of different ways to interpret a byte. In WAH, there are only two types of regular words: *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 any bit.

Let w denote the number of bits in a word, the lower $(w-1)$ bits of a literal word contain the literal bit values. The second most significant bit of a fill word is the fill bit (0 or 1), and the remaining bits store the fill length. WAH imposes the word alignment requirement on the fills; all fill lengths are integer multiples of the number of bits in a literal word, $(w-1)$. For example, on a 32-bit CPU ($w = 32$), all fill lengths are integer multiples of 31 bits. In an implementation of a similar compression scheme without word alignment, tests show that the version with word alignment frequently outperforms the one without word alignment by two orders of magnitude [Wu et al. 2001]. The reason for this performance difference is that the word alignment ensures logical operations only access whole words, not bytes or bits.

Figure 2 shows the WAH compressed representation of 128 bits. We assume that 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 the bitmap as 31-bit groups, and the third line shows the hexadecimal representation of the groups. The last line shows the WAH words also as hexadecimal numbers. The first three words are regular words, the first and the third are literal words and the second one is a fill word. The fill word 80000002 indicates a 0-fill of two-word long (containing 62 consecutive 0 bits). Note that the fill word stores the fill length as two rather than 62. The fourth word is the *active word*, it stores the last few bits that could not be stored in a regular word[1].

For sparse bitmaps, where most of the bits are 0, a WAH compressed bitmap would consist of pairs of a fill word and a literal word. If the bitmap is truly sparse, say only one bit in 1000 is 1, then each literal words would likely contain a single bit that is 1. In this case, for a set of bitmaps to contain N bits of 1, the total size of all compressed bitmap is about $2N$ words. In the next section, we will give a rigorous analysis of sizes of compressed bitmaps.

[1]Note that we need to indicate how many bits are represented in the active word, and we have chosen to store this information separately (not shown in Figure 2). We also chose to store the leftover bits as the least significant bits in the active word so that during bitwise logical operations the active word can be treated the same as a regular literal word.

	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

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

The detailed algorithms for performing logical operations are given in the appendix. Here we briefly describe one example (C = A AND B) as shown in Figure 3. To perform a logical AND operation, we essentially need to match each 31-bit group from the two operands, and generate the corresponding groups for the result. Each column of the table is reserved for one such group. A literal word occupies the location for the group and a fill word is given at the first space reserved for the fill. 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 0 bits. The active words are always treated separately.

When two sparse bitmaps are lined up for a bitwise logical operation, we expect most of the literal words not to fall on top of each other or adjacent to each other. In this case, the results of logical OR and XOR are as large as the total size of the two input bitmaps. It is easy to see that the time required to perform these logical operations would be linear in the total size of the two input bitmaps. We have observed this linearity in a number of different tests [Wu et al. 2001; 2002; 2004]. We formally show this linearity in Section 5.

The version of WAH presented in this paper has two important improvements over the earlier version [Wu et al. 2001]. The first change is that we use the multiple of $(w-1)$ bits to measure the fill length rather than the number of bits. The second change is that we assume the length of a bitmap can be represented in one computer word. These changes allow us to store a fill of any length in one fill word, which reduces the complexity of encoding and decoding procedures. In addition, our current implementation of the bitwise logical operations also takes advantages of short-cuts available for the specific logical operations, while the original performance measurement [Wu et al. 2001] use the generic algorithm shown in the appendix. All of these changes improved the efficiency of bitwise logical operations.

Operations on WAH compressed bitmaps should be faster than the same operations on BBC compressed bitmaps for three main reasons.

(1) The encoding scheme of WAH is simpler than BBC, therefore the algorithms for performing logical operations are also simpler. In particular, the header byte of a BBC run is considerably more complex than any WAH word.

(2) The words in WAH compressed bitmaps have no dependency among them, but the bytes in BBC have complicated dependencies. Therefore, accessing BBC compressed bitmaps is more complicated and more time consuming than accessing WAH compressed bitmaps.

(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 such a fill it starts a new run, while WAH represents such fills in literal words. It takes much less time to operate on a literal word in WAH than on a run in BBC.

One way to improve the speed of bitwise logical operations on bitmaps with many short fills is to decompress all the short fills. Indeed, decompressing all short fills in both WAH and BBC could decrease the time to perform bitwise logical operations on these bitmaps. However, it usually increases the time to perform operations between a uncompressed bitmap and another compressed bitmap. It is possible that decompressing selected bitmaps could reduce the average query response time, but in all tests, the query response time increased rather than decreased. Further investigation may reveal exactly what to decompress to improve the query response time; however, we will leave that for future work. For the remainder of this paper, unless explicitly stated, all bitmaps are fully compressed.

4. COMPRESSED SIZES OF BITMAP INDICES

In this section, we examine the space complexity of the WAH compression scheme. A compressed bitmap index typically consists of three parts: the bitmaps, the attribute values, and the data to associate each bitmap with an attribute value. Usually, the data to associate a bitmap with an attribute value is trivial to account for. The main quantity to be computed for the space complexity analysis is the total size of all bitmaps in a compressed bitmap index. Without loss of generality, we only consider integer attributes when computing the total size of bitmaps. We give exact formulas for two types of attributes, the uniform random attributes and the attributes generated from a Markov process. As the attribute cardinality increases, the expression for the uniform random attributes asymptotically approaches the upper bound [Wu et al. 2004]. The index sizes of attributes generated from the Markov process can be much smaller than those of uniform random attributes. The main steps to compute the total size of bitmaps include computing the number of counting groups, computing the sizes of individual bitmaps, and finally the total sizes of all bitmaps in an index. These three steps are each described in the following subsection.

4.1 The counting groups

We need to define a few symbols to represent the basic quantities to be discussed. A list of frequently used symbols is in Figure 4. Let w be the number of bits in a computer word. On a 32-bit CPU, $w = 32$, and on a 64-bit CPU, $w = 64$. In the following analyses, we assume the number of bits N of a bitmap (or equivalently the number of rows) can be represented in a computer word[2], i.e., $N < 2^w$.

Given a bitmap, the WAH compression scheme divides the bits into $(w-1)$-bit groups. For convenience, we call a $(w-1)$-bit group a *literal group*. For a bitmap

[2]If the database contains more records than 2^w rows, multiple bitmaps can be used each to represent a subset of the rows. To improve the flexibility of generating the indices and caching the indices in memory, a bitmap may contain only a few million bits, corresponding to a small partition of the data. This will create more than c bitmaps. The total size of the bitmap index due to the per bitmap overhead will increase accordingly; however, the number of words required to represent the bulk of 0s and 1s will not change.

c :	Cardinality of an attribute, the number of distinct values in the dataset.
d :	The bit density, the fraction of bits that are 1.
f :	Clustering factor of a bitmap, the average number of bits in 1-fills.
h :	Number of 1s in a bitmap.
H :	Number of hits of a query.
I :	Number of iterations. I_w number of iteration through the main while-loop in Algorithm `generic_op` defined in the Appendix.
L :	Total length of all 1-fills in a WAH compressed bitmap (in number of words). L_y is the total length of 1-fills in `y`.
m :	Number of regular words in WAH compressed bitmap, $m_R(d)$ is the expected number words for a random bitmap and $m_M(d, f)$ is the expected number of words for a bitmap generated from a two-state Markov process.
M :	The maximum number of regular words required to store N bits using WAH compression, $M = \lfloor N/(w-1) \rfloor$.
N :	Number of bits in a bitmap, also number of records in a table.
s :	Expected size of a compressed bitmap index, s_U is the index size for an uniform random attribute and s_M is the index size of an attribute generated from a Markov process.
t :	Time to perform a bitwise logical operation, t_G using `generic_op`, t_I using `inplace_or`.
T :	Time to answer a one-dimensional range query, T_G using `generic_op` only, T_I using `inplace_or` only.
w :	Word size, the number of bits in a word, typically, 32 or 64.

Fig. 4. A list of frequently used symbols.

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 an active word. The regular words could all be literal words, each representing one literal group. In later discussions, we refer to this 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, this gives 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 integer numbers of literal groups. In the example bitmap shown in Figure 2, the longest fill without the word alignment restriction contains 79 bits of 0. However, 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 of 2^w bits or less 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 is $2^{w-2}-1$. Since the fill length is measured in number of literal groups, the maximum number of bits represented by one fill word is $(w-1)(2^{w-2}-1)$. 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,

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

Fig. 5. Breaking the 31-bit groups from Figure 2 into three counting groups.

counting the number of fills is messy. Instead we define an auxiliary concept called the *counting group* to simplify the counting procedure. 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 5. 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 form a fill, then the $(l-1)$ counting groups are also fills. The two literal groups at the ends 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 a different type, (2) a literal group with a mixture of 0 bits and 1 bits, 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 remaining cases, the counting groups constructed with one literal group inside the fill and one literal group outside the fill 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 denote the number of counting groups that are fills and let M denote the number of literal groups. Then 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 $L=\sum_{i=1}^{L} l_i - \sum_{i=1}^{L}(l_i-1)$, and the number of groups that are not in any fills is $M-\sum_{i=1}^{L} l_i$. Altogether, the number of fills plus the number of literal groups 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 a compressed bitmap we only need to compute the number of counting groups that

are fills. Because the counting groups have a fixed size, it is relatively easy to compute G for random bitmaps.

4.2 Sizes of random bitmaps

Now that we have the basic tool for counting, we can examine the size of some random bitmaps. The 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. These bitmaps can be characterized with one parameter; the *bit density* d, which is defined to be the fraction of bits that are 1 ($0 \le d \le 1$).

The efficiency of a compression scheme is often measured by the *compression ratio*, which is the ratio of its compressed size to 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 to represent incompressible bitmaps. For WAH, this overhead is one bit per word. When a word is 32 bits, the overhead is about 3%. The overhead for BBC is about one byte per 15 bytes, which is roughly 6%.

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 1, 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

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

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. We loosely refer to it as the "per bitmap overhead". This overhead 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 any 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 cannot be compressed with WAH. For sparse bitmaps, say $2wd \ll 1$ we have $m_R(d) \approx 2dN$, because $d^{2w-2} \to 0$ 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 1. By definition, $h = dN$. The compressed size of a sparse bitmap is related to h by 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 1, and each literal word is separated from the next one by a 0-fill. On average, two words are used for each bit that is 1.

Next, we compute the compressed sizes of bitmaps generated from a two-state Markov process as illustrated in Figure 6. These bitmaps require a second param-

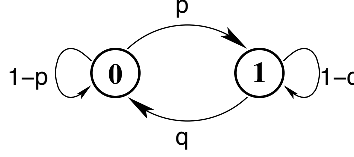


Fig. 6. An illustration of the two-state Markov process.

eter to characterize them. We call this parameter the *clustering factor* f. The clustering factor of a bitmap is the average number of bits in a 1-fill. The bit density d and the clustering 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 its 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 from state 1, the probability of its staying at state 1 for i more steps is proportional to $(1-q)^i$. The expected number of bits[3] in a 1-fill is as follows[4].

$$f = \frac{\sum_{i=0}^{\infty}(i+1)(1-q)^i}{\sum_{i=0}^{\infty}(1-q)^i} = q\sum_{i=0}^{\infty}(i+1)(1-q)^i = \frac{1}{q}.$$

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

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

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

---

[3]The authors gratefully acknowledge the suggestion from an anonymous referee to simplify the derivation of f.
[4]Note $\sum_{i=0}^{\infty}(1-q)^i = 1/q$. Let $S \equiv \sum_{i=0}^{\infty}(i+1)(1-q)^i$, since $\sum_{i=0}^{\infty}(i+1)(1-q)^i = 1/q + \sum_{i=1}^{\infty} i(1-q)^i = 1/q + (1-q)\sum_{i=0}^{\infty}(i+1)(1-q)^i$, $S = 1/q + (1-q)S$. Therefore, $S = 1/q^2$.
The computation of f is similar to that of the rehash chain length [O'Neil and O'Neil 2000, p. 520].

4.3 Total size of bitmaps

The first index size we compute is for a discrete random attribute following the simplest uniform probability distribution. Let c be the attribute cardinality, each value appears with the same probability of $1/c$. It is easy to see that the bitmap corresponding to a value would be a uniform random bitmap with a bit density $d = 1/c$. Since a bitmap index consists of c such bitmaps, the total size of the bitmaps (in number of words) is

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

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.

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

This formula is based on an approximation of Equation 1 which neglected the per bitmap overhead, the constant 2. Since there are c bitmaps, the total per bitmap overhead is $2c$ words. For very large c, a better approximation is

$$s_U \approx 2N + 2c. \tag{6}$$

For non-uniform random attributes, let the ith value have a probability of p_i. The total size of the bitmap index compressed with WAH is

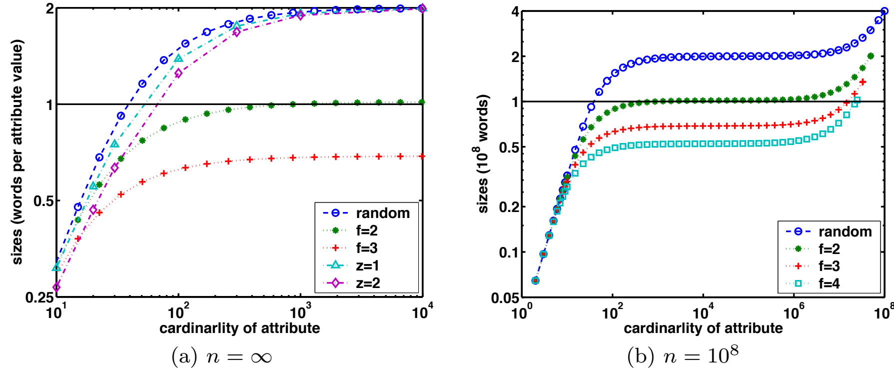
$$s_N \equiv \sum_{i=1}^{c} m_R(p_i) \approx \frac{N}{w-1}\left(c - \sum_{i=1}^{c}(1-p_i)^{2w-2} - \sum_{i=1}^{c} p_i^{2w-2}\right). \tag{7}$$

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 many cases, the probability distribution of a random attribute depends on existing values. The simplest model that captures this dependency is the Markov process. An attribute of cardinality c can be modeled by a c-state Markov process. For simplicity, we again assume a uniform probability distribution for the attribute. In other word, from any state, this Markov process has the same transition probability q to an another state, and it selects one of the $c-1$ states with equal probability. In this case, each bitmap corresponding to the c values are the same as those generated by the two-state Markov process described earlier. The total size of the bitmaps is

$$\begin{aligned} s_M &\equiv cm_M(\frac{1}{c}, f) \\ &\approx \frac{Nc}{w-1}\left(1 - \left(1-\frac{1}{c}\right)\left(1 - \frac{1}{(c-1)f}\right)^{2w-3} - \frac{1}{c}\left(1-\frac{1}{f}\right)^{2w-3}\right). \end{aligned} \tag{8}$$

Figure 7 shows the total bitmap index sizes of a number of different synthetic attributes. The line marked "random" is for the uniform random attributes. The lines marked with "f=2," "f=3," and "f=4" are attributes generated from the Markov



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

Fig. 7. The expected size of bitmap indices on random data and Markov data with various clustering factors.

process with the 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 7 displays the total sizes of the bitmaps under two conditions, the infinite N and the finite N. In the first case, the cardinality is always much less than N. The vertical axis in Figure 7(a) is s/N. As the cardinality increases, it is easy to see that $s/N \to 2$. When N is finite, it is possible for c to be as large as N. In this case, we cannot simply drop the small constants in Equations 1 and 3. If $c = N$, the total size of all bitmaps is nearly $4N$ words[5]. The majority of bitmaps have three regular words plus the active word[6]. There are a few bitmaps using two or three words rather than four[7]. For a large range of high cardinality attributes, say, $c < N/10$, the maximum size of WAH compressed bitmap indices is about $2N$ words.

For attributes with a clustering factor f greater than one, the stable plateau is reduced by a factor close to $1/f$. Another factor that reduces 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 $2N$ words. The following proposition summarizes these observations.

[5] The exact maximum is $4N - 2w - 2(N\%(w - 1))$, where operator % denotes modulus.

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

[7] The three regular words in the majority of the bitmaps represents a 0-fill, a literal group, and a 0-fill. There are w bitmaps without the first 0-fill and w bitmaps without the last 0-fill. There $2w$ bitmaps uses three words each. There are also $(N\%(w - 1))$ bitmaps whose 1 bits are in their active words. In these bitmaps, only one regular word representing a 0-fill is used.

PROPOSITION 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 total size s of all compressed bitmaps in an index is such that,*

(1) *it never takes more than $4N$ words,*

(2) *if $c < N/10$, 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 $c < N/10$, 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 proposition, item (2) is a restatement of Equation 5, and item (3) is an approximation of Equation 8. Item (1) can be proved 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 1. 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 7. 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 scan the data to find out how many times a particular value appears consecutively, 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 an index without first estimating its size.

The above formulas only include the size of the compressed bitmaps of an index. They do not include the attribute values or other supporting information required by a bitmap index. When stored on disk, we use two arrays in additions to the bitmaps, one to store the attribute values and the other to store the starting position of the bitmaps. Since we pack the bitmaps sequentially in a single file, there is no need to store the end positions of most of the bitmaps except the last one. For an index with c bitmaps, there are c attribute values and c starting positions. If each attribute value and each starting position can be stored in a single word, the file containing a bitmap index uses $2c$ more words than the total size of bitmaps. For most high cardinality attributes, the index file size is at most $2N + 4c$ because the total bitmap size is about $2N + 2c$ as shown in Equation 6. In the extreme case where $c = N$, the index file size is $6N$. It may be desirable to reduce the size in this extreme case, for example, by using different compression schemes or using an RID

list [O'Neil 1987]. In our experience, we hardly ever see this extreme case even for floating-point valued attributes. The maximum size of WAH compressed bitmap indices in a typical application is $2N$ words, and the average size may be a fraction of N words.

5. TIME COMPLEXITY OF QUERY PROCESSING

To answer a one-dimensional range query using a compressed bitmap index, we first need to examine the attribute values to determine what bitmaps need to be OR'ed together as explained in the example given in Section 1.1. Typically, the bulk of the query processing time is used to read these bitmaps from disk and perform the bitwise logical operations. Because the time to read the bitmaps is known to be a linear function of the total size of the bitmaps, we concentrate on the CPU time required by the logical operations. More specifically, we compute the time complexity of two algorithms used to perform the logical operations. The first one called `generic_op` shown in Listing 1 is designed to perform any binary bitwise logical operations including AND, OR, and XOR. The second one called `inplace_or` shown in Listing 2 is designed to perform bitwise OR operations only. It takes a uncompressed bitmap and a compressed bitmap, and places the result in the uncompressed bitmap. It is intended to OR multiple bitmaps of an index during the evaluation of a range condition. Because it avoids allocating intermediate results, it is usually faster than `generic_op` to operate on a large number of bitmaps. Before we give the detailed analyses, we first summarize the main points.

The time to perform an arbitrary logical operation between two compressed bitmaps is a linear function of the total size of the two bitmaps. The exception is when the two operands are nearly uncompressed, in which case the time needed is constant. The time to perform a logical OR operation between a uncompressed bitmap and a compressed one is linear in the size of the compressed one. When OR'ing a large number of sparse bitmaps using `inplace_or`, the total time is linear in the total size of all input bitmaps. When answering a query, this total size is at most a constant times the number of hits; therefore, the query response time is at worst a linear function of the number of hits. This means the compressed bitmap index is optimal for answering one-dimensional range queries.

The remainder of this section is divided into three subsections each discussing the complexity of `generic_op`, the complexity of `inplace_or`, and the complexity of answering one-dimensional range queries respectively.

5.1 Complexity of algorithm **generic_op**

In the appendix, we show four pseudo-code segments to illustrate two algorithms used to perform bitwise logical operations, `generic_op` and `inplace_or`. The code segments follow C++ syntax and make use of the standard template library (STL). The first two listings contain the two main functions and the last two contains data structures and supporting functions. In these listings, compressed bitmaps are represented as `bitmap` objects. In a `bitmap`, the regular words are stored as a vector of unsigned integers named `vec`. The ith regular word can be addressed as `vec[i]`. The three key functions used for performing bitwise logical operations are `bitmap::appendLiteral`, `bitmap::appendFill`, and `run::decode`, where the first two are shown in Listing 3 and the last one is shown in Listing 4.

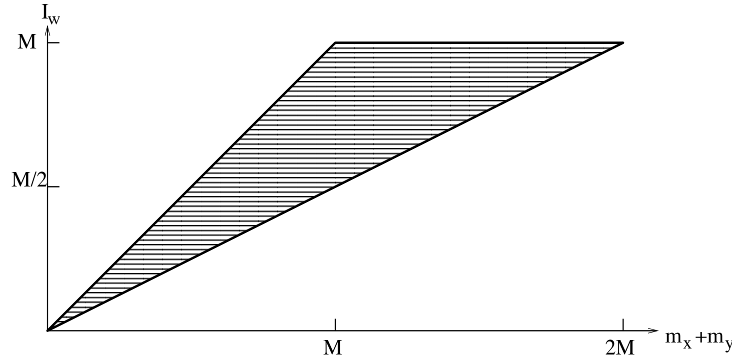


Fig. 8. The range of possible values for the number of iterations I_w through the main WHILE loop of function `generic_op` defined in Listing 1.

We first compute the time complexity of algorithm `generic_op`. To do this, we compute the number of iterations required for the main while-loop, and then show that the time complexity of each iteration of the main while-loop is bounded by a constant. In `generic_op`, each iteration either produces a literal word or a fill, which leads 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 uncompressed bitmap, then the number of iterations through the while-loop I_w satisfy the following condition,*

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

PROOF. To prove this lemma, we observe that each iteration of the while-loop consumes one word from either operand `x` or `y`, or one word from each 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, 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 $m_x + m_y - 1$. 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 8 pictorially depicts the range of values for I_w. Both the upper bound and the lower bound of the ranges are achievable; however, it is less likely to reach the lower bound than to reach the 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 $I_w = m_x = m_y$. This could happen, for example, if `x` and `y` are complements of each other. In a more typical case where the two operands are not related this way, we expect I_w to be close to its maximum. Since each iteration generates at most one word for the result `z`, we have the following corollary from the above lemma.

COROLLARY 6. *Using the algorithm* `generic_op` *to perform a bitwise logical operation between* `x` *and* `y` *gives the following:*

(a) *the number of iterations* $I_w \leq \min(m_x + m_y, M)$*, and*

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

A quick examination of the various functions invoked by `generic_op` reveals that the only function that might take an undetermined amount of time is an STL function called `push_back`, since this function may need to expand the storage allocated to `z.vec`. Because we know the maximum size required, we can allocate the maximum amount of space for `z.vec` and avoid the dynamic memory allocation. Consequently, each invocation of `push_back` would take only a constant amount of time.

Most functions and operators in `generic_op` are used I_w times except three functions, the function `run::decode` which is invoked m_x+m_y times, and functions `bitmap::appendLiteral` and `bitmap::appendFill` which are invoked a total of I_w times. Since the cost of both `bitmap::appendLiteral` and `bitmap::appendFill` are bounded by a constant, we can use the a common bound for both of them. This leads to the formula for the time as follows

$$t_G \leq C_d(m_x + m_y) + C_1 I_w \quad (9)$$

$$\leq C_2(m_x + m_y), \quad (10)$$

where C_d is the cost of decoding one WAH word, C_1 is the cost of all operations other than decoding compressed words, and C_2 is the sum of C_d and C_1. This proves the following theorem.

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

For sparse bitmaps, where $m_x + m_y \ll M$, the equality in Equation 10 is actually achievable.

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 operands is a 0-fill, 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 or consuming more than one word from each operand. In addition, if we assume that both operands are properly compressed, when appending a number of consecutive literal words, we may invoke the function `bitmap::appendLiteral` only on the first word, and use `vector::push_back` on the remaining words. Since the chance of discovering fills in 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**

In an example given in Section 1.2, we need to OR 5,000 bitmaps in order to answer a query. If each OR operation generates a new compressed bitmap, the time of memory management may dominate the total execution time. To reduce the memory management cost, we copy one of the bitmaps to a uncompressed

bitmap and use it to store the result of the operation[8]. We call this the in-place OR operation and denote it as $x \mid= y$. In addition to avoiding repeated allocation of new memory for the intermediate results, the in-place OR also avoids repeatedly generating 0-fills in the intermediate results.

Following the derivation of Equation 10, 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 $C_d m_y$. Let L_y denotes the length of all 1-fills in `y`, the total number of iterations through the inner loop marked "assign 1-fill" is L_y. Assume each inner iteration takes C_4 seconds, then the total cost of this inner loop is $C_4 L_y$. The main loop is executed m_y times, and 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 of algorithm `inplace_or` is

$$t_I = C_3 m_y + C_4 L_y. \tag{11}$$

Let d_y denote the bit density of `y`, for sparse bitmaps where $2wd_y \ll 1$, $L_y = Nd_y^{2w-2}/(w-1) \to 0$. In this case, the above formula can be stated as follows.

THEOREM 8. *On a sparse bitmap y, the time complexity of algorithm* `inplace_or` *is* $\mathcal{O}(m_y)$*, where* m_y *is the number of regular words in y.*

5.3 Optimal time complexity for range queries

In a previous paper, we analyzed five options for OR'ing multiple bitmaps of an index to answer a range query [Wu et al. 2004]. Those analysis was informal because we did not establish the complexity of a logical operation involving two operands. Now that we have formally established the time complexities of a binary logical operation, we formally analyze two of the most useful options: one using `generic_op` only and the other using `inplace_or` only. In tests, we saw that the first option is more efficient for a relatively small number of bitmaps and the second option is more efficient for a larger number of bitmaps [Wu et al. 2004]. The theoretical complexity of query processing is dominated by the second option. In this section, we show that the time complexity of answering a one-dimensional range query is linear in the total size of bitmaps involved and linear in the number of hits.

Because the answer to a multi-dimensional range query can be easily computed from answers to one-dimensional queries, our analysis here concentrates on range conditions on one attribute. When evaluating these one-dimensional range conditions, e.g., "`150 <= Energy < 200`," it is easy to determine the total size of bitmaps involved. If this total size is more than half of the total size of all bitmaps, we can evaluate the complement of the range condition and then complement the solution. This guarantees that we never need to access more than half of the bitmaps.

When processing range conditions involving high cardinality attributes, we may need to operate on a large number of bitmaps, but the size of each individual bitmap is much smaller than M. Using `generic_op` to operate on two such bitmaps, the number of iterations I_w is usually close to the upper bound $m_x + m_y$. To make it

[8]This is similar to the *basic* method used by Johnson [1999], however Johnson's implementation involves a literal bitmap and a compressed bitmap, only WAH compressed bitmaps are used in our case.

easier to reveal the worst case behavior, we simply assume I_w is $m_x + m_y$. Under this assumption, the time used by `generic_op` is $t_G = C_2(m_x + m_y)$ and the result z has $m_x + m_y$ regular words. In later discussions, we will refer to this set of assumptions as the *sparse bitmap assumption*.

Let $m_1, m_2, \ldots, m_k$ denote the sizes of the k compressed bitmaps to be OR'ed. The time required to call function `generic_op` $(k-1)$ times is

$$\begin{aligned} T_G &\leq C_2(m_1 + m_2) + C_2(m_1 + m_2 + m_3) + \ldots + C_2(m_1 + m_2 + \ldots + m_k) \\ &= C_2 \left[\left(\sum_{i=1}^{k} (k + 1 - i) m_i \right) - m_1 \right]. \end{aligned} \tag{12}$$

Assuming 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 quadratically in the number of input bitmaps[9]. In an earlier test, we actually observed this quadratic relation [Wu et al. 2004]. Next we show that a linear relation is achieved with `inplace_or`.

To use `inplace_or`, we first need to produce a uncompressed bitmap. Let C_0 denote the amount of time to produce this uncompressed bitmap. To complete the operations, we need to call `inplace_or` k times on k input compressed bitmaps. The total time required is

$$T_I = C_0 + C_3 \sum_{i=1}^{k} m_i + C_4 \sum_{i=1}^{k} L_i,$$

where L_i denotes the total length of the 1-fills in the ith bitmap. Under the sparse bitmaps assumption, the term $C_4 \sum_{i=1}^{k} L_i$ in the above equation is much smaller than others. This leads to

$$T_I \approx C_0 + C_3 \sum_{i=1}^{k} m_i, \tag{13}$$

which proves the following proposition.

Proposition 9. *The time required to OR a set of sparse bitmaps using* `inplace_or` *is a linear function of the total size of input bitmaps.*

If all input bitmaps have the same size m, Equation 13 is linear in k, while Equation 12 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 faster. In an earlier paper, we identified a simple algorithm to ensure the faster option is used in practice [Wu et al. 2004]. This guarantees that the actual query response time is never worse than using `inplace_or` alone.

[9]Note that this quadratic relation holds only if all the intermediate results are also sparse bitmaps. If some intermediate results are not sparse, we should use Equation 9 rather than Equation 10 to compute the total time. This would lead to a more realistic upper bound on time. When a small number of bitmaps are involved, using `generic_op` is faster than using `inplace_or`. As the number of bitmaps increases, before Equation 12 becomes a gross exaggeration, using `inplace_or` already becomes significantly better [Wu et al. 2004]. For this reason, we have chosen to omit the formula for dense intermediate results.

Proposition 10. *Using a WAH compressed bitmap index to answer one-dimensional range queries is optimal.*

Proof. 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 a long query response time is expected, there must be a large number of sparse bitmaps involved. Under the sparse bitmap assumption, the total size of all bitmaps is proportional to the number of 1s as shown in Equation 2. According to Proposition 9, the total search time using a compressed bitmap index is linear in the number of hits. Therefore, the compressed bitmap index is optimal. □

We did not label the last two propositions as theorems because there are a number of factors that may cause the observed time to deviate from the expected linear relation. Next we describe three major ones.

The first one is that the "constant" C_0 actually is a linear function of M. To generate a uncompressed bitmap, one has to allocate $M + 2$ words and fill them with (zero) values. Because the uncompressed bitmap is generated in memory, the procedure is very fast. The observed value of C_0 is typically negligible.

The second factor is the memory hierarchy in 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 uncompressed bitmap, and the content of the uncompressed 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. Many 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 increase as the attribute cardinality increases. In the extreme case, H cache lines are loaded, one for each hit. In short, the value of C_3 depends on some characteristics of the data, but it approaches an asymptotic maximum as the attribute cardinality increases.

The third factor is that the above analysis neglected the time required to determine what bitmaps are needed to answer a query. Our test software uses binary searches on the attribute values to locate the bitmaps. Theoretically, a binary search takes $\mathcal{O}(\log(c))$ time. One way to reduce this time would be to use a hash function which can reduce this time to $\mathcal{O}(1)$ [Czech and Majewski 1993; Fox et al. 1991]. Timing measurements show that the binary searches take negligible amount of time, therefore, we have not implemented the hash functions.

6. PERFORMANCE OF THE LOGICAL OPERATIONS

In this section, we review the performance of bitwise logical operations. Since these operations are the main operations during query processing, their performance is important to the overall system performance. This section contains three subsections, one to describe the experiment setup, one to discuss the logical operation time, and one to discuss the bitmap index size. The goal is to confirm the formulas derived in the two previous sections.

6.1 Experiment setup

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 STAR [Shoshani et al. 1999; Stockinger et al. 2000]. The data used in our tests can be viewed as one relational table consisting of about 2.2 million rows with 500 columns. The bitmap indices used in this test are from the 12 most frequently queried attributes. The second application dataset is from a direct numerical simulation of a combustion process [Wu et al. 2003]. This dataset contains about 25 million rows with 16 columns. Both application datasets contain mostly 4-byte integers and floating-point values, only one attribute in the STAR dataset has 8-byte floating-point values.

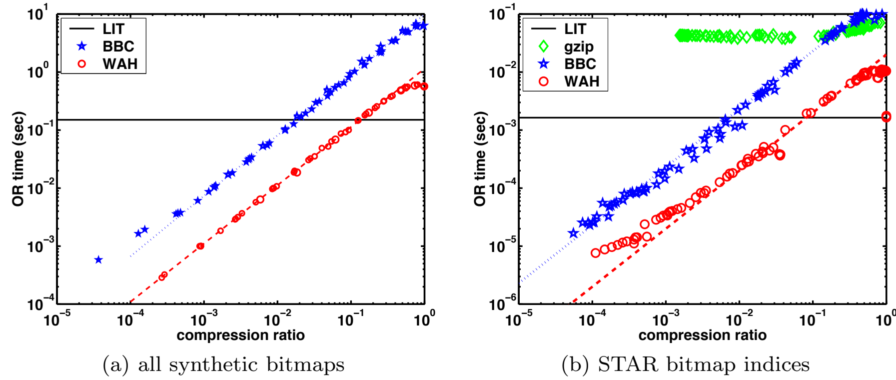
We conducted a number of tests on different machines and found that the relative performances among the different compression schemes are independent of the specific machine architecture. This was also observed in a different performance study [Johnson 1999]. The main reason for this is that most of the clock cycles are consumed by table look ups and 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[10] 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[11].

To avoid cluttering the graphs, 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 as the logical OR operation. Usually, the XOR operation takes a few percent more time than the other two.

6.2 Logical operation time scales as predicted

The most likely scenario of using the bitmaps in a database system is to read a number of them from disks and then perform bitwise logical operations on them. With most of the compression schemes tested, the logical operations can directly use the bitmaps stored in files; only gzip 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 of the operations are not compressed. 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 [Wu et al. 2001]. The timing results for WAH and BBC are for logical operations on two

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



(a) all synthetic bitmaps

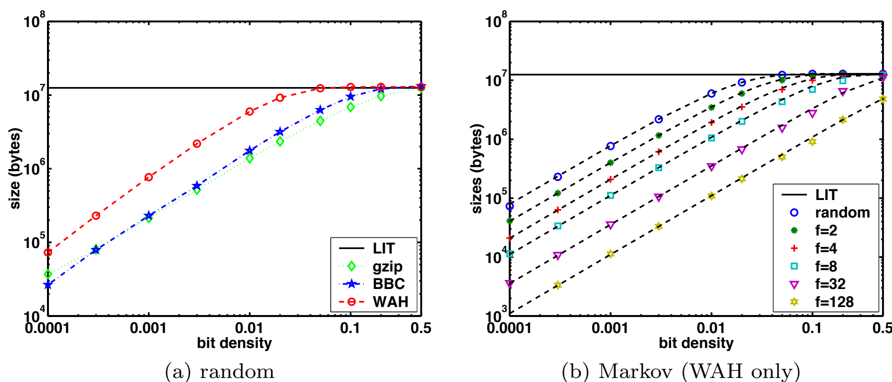
(b) STAR bitmap indices

Fig. 9. Logical operation time is proportional to compression ratio of the operands, and therefore the total size of the operands. On the STAR bitmap indices, the total CPU time used by BBC is about 12 times of that of WAH.

compressed bitmaps that produce one compressed result, i.e., the direct method used by Johnson [1999]. The main objective of this subsection is to verify whether the logical operation time is proportional to the total sizes of the operands as Theorem 7 predicts. Since Theorem 7 is only concerned about the CPU time, we only measured the CPU time.

We measure the CPU time of logical operations on many pairs of bitmaps and plot the results in two graphs according to the number of bits in the bitmaps. The timing results on the two sets of synthetic data are in Figure 9(a) and the results on the STAR bitmaps are in Figure 9(b). In both cases, the compression ratio (the ratio of a compressed size to its uncompressed counterpart) 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 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 1 approach a constant. For very small bitmaps, 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 for WAH. Much of this difference can be attributed to the large number of relative short fills in the bitmaps. BBC is more effective in compressing these short fills, but it also takes more time to use these fills in bitwise logical operations. In contrast, WAH typically does not compress these short fills. When performing bitwise logical operations on these uncompressed fills, WAH can be nearly 100 times faster than BBC. The performance differences between WAH and BBC are the smallest on



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

Fig. 10. The sizes of the compressed bitmaps. The symbols for the Markov bitmaps are marked with their clustering factors. The dashed lines are predictions based on Equations 1 and 3.

sparse bitmaps. On very sparse bitmaps, WAH is about four to five times faster than BBC when bitmaps are in memory. When the bitmaps are read from disk, WAH is about two to three times faster than BBC.

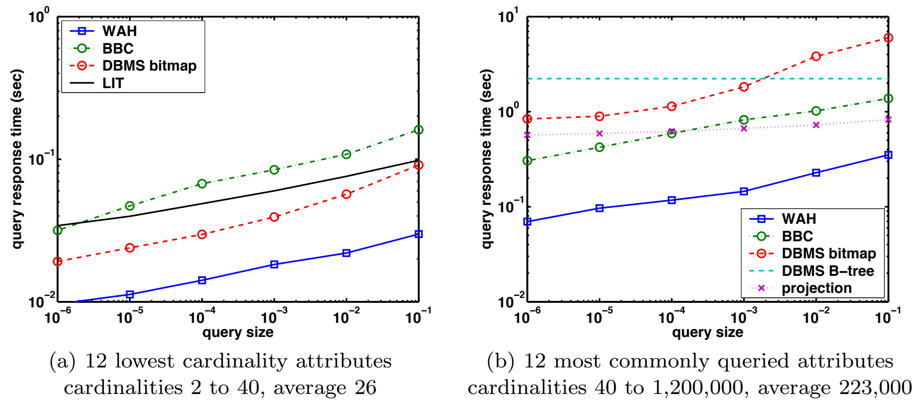
6.3 Sizes of compressed bitmaps agree with analyses

Figure 10 shows the sizes of the four types of bitmaps. Each data point in this figure represents the average size of a number of bitmaps with the same bit density and clustering factor. The dashed lines in Figure 10 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 they take more space. 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, most bitmaps are incompressible, and all compression schemes 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. 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 compression ratios of bitmaps vary widely, and the total size is dominated by bitmaps with the largest compression ratios. Since most schemes use about the same amount of space to store these incompressible bitmaps, 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% ($186/118 \sim 1.6$) bigger than those compressed with BBC as shown in Table I.

7. PERFORMANCE ON RANGE QUERIES

This research was originally motivated by the need to manage the volume of data produced by the STAR experiment. In this case, the queries on the data are multi-dimensional range queries. The range conditions on each attribute are of the form $x_1 \leq X < x_2$, where X is an attribute name, x_1 and x_2 are two constants. An equivalent statement in SQL is "`select count(*) from T where x1 <= X and X`



(a) 12 lowest cardinality attributes cardinalities 2 to 40, average 26

(b) 12 most commonly queried attributes cardinalities 40 to 1,200,000, average 223,000

Fig. 11. The average response time of 5-dimensional queries on the STAR dataset. The query size is the expected fraction of events that are hits.

< x2." A typical query from STAR involves a handful of this type of conditions in conjunction. In this section, we use random range conditions for our performance measurements[12]. The main objective is to confirm the analysis presented in Section 5.

The average query response time shown in Figure 11 and Table I is measured as the elapsed time by the client program. In this case, all indices from both the commercial DBMS and our own implementation, except the uncompressed (LIT) index for high cardinality attributes, fit into memory. The query response time in this case mostly reflects the in-memory performance of the bitmap compression schemes. In contrast, the time values shown in Figures 12 and 13 are measured with cold disks, where nothing is cached in memory[13].

7.1 Compressed indices perform well on multi-dimensional queries

Figure 11 shows the average query response time of 5-dimensional range queries where each query is a conjunction of five random range conditions on the STAR dataset. The left plot is for low cardinality attributes and the right plot is for high cardinality attributes. Each data point is the average time of 1000 different queries. For both high and low cardinality attributes, we see that WAH compressed bitmap indices use significantly less time than others. On low cardinality attributes, the uncompressed bitmap indices are smaller than projection indices and B-tree indices, and are also much more efficient. This agrees with what has been observed by others [O'Neil and Quass 1997]. We report the performance of a commercial implementation of the compressed bitmap index to validate our own implementation. Our implementation of BBC compressed index and the DBMS implementation perform about as well as the uncompressed bitmap indices (marked LIT in Figure 11) on

[12]The values x_1 and x_2 are chosen separately and each is chosen from the domain of X with uniform probability distribution. If $x_1 > x_2$, we swap the two. If $x_1 = x_2$, the range is taken to be $x1 \leq X$.

[13]In this case, before a query is run, the file system containing raw data and indices are unmounted and mounted again.

		projection	commercial DBMS B-tree	commercial DBMS bitmap	our bitmap indices LIT	our bitmap indices BBC	our bitmap indices WAH
12 lowest cardinality	size (MB)	113	370	7	84	4	7
	time (sec)	0.57	0.85	0.01	0.01	0.007	0.003
12 commonly queried	size (MB)	113	408	111	726,484	118	186
	time (sec)	0.57	0.95	0.66	-	0.32	0.052

Table I. Total sizes of the indices on STAR data and the average time needed to process a random one-dimensional range query.

the low cardinality attributes. On the 12 most frequently queried attributes, the query response time is longer than that on the low cardinality attributes; however, the WAH compressed indices are still more efficient than others. On these high cardinality attributes, projection indices are about three times as fast as B-tree indices; the WAH compressed indices are at least three times as fast as projection indices.

Table I shows the total sizes of various indices and the average time required to answer a random one-dimensional query on the STAR data[14]. We consider the projection index as the reference method since it is compact and efficient [O'Neil and Quass 1997]. The projection index size is the same as the raw data, which is smaller than most indices on high cardinality attributes. The bitmap index sizes reported are the index file sizes which include the bitmaps, attribute values, and the starting positions of bitmaps. The particular B-tree reported in Table I is nearly four times the size of the raw data. We did not generate the uncompressed bitmap indices for high cardinality attributes because they would take 726 GB of disk space and clearly would not be competitive against other indices.

On the high cardinality attributes, the WAH compressed bitmap index is about 60% ($186/118 \sim 1.6$) larger than the BBC compressed index. In terms of query response time, the WAH compressed indices are about 13 ($0.66/0.052 \sim 13$) times faster than the commercial implementation of the compressed index, and about 6 ($0.32/0.052 \sim 6$) times faster than our own implementation of the BBC compressed index. In the previous section, we observed that WAH performs bitwise logical operations about 12 times as fast as BBC. The difference in query response time is less than 12 for two main reasons. First, the average query response time weighs operations on sparse bitmaps more heavily because it takes more sparse bitmaps to answer a query. On sparse bitmaps, the performance difference between WAH and BBC could be as low as 2. Secondly, the query response time also includes time such as network communication, parsing of the queries, locking and other administrative operations. In this test, all indices can fit into memory. For larger datasets, where more I/O is required during query processing, the relative difference between using WAH and using BBC would be smaller [Stockinger et al. 2002; Wu et al. 2004]. However, unless the I/O system is extremely slow, say, 2 MB/s, using WAH is preferred [Stockinger et al. 2002]. This observation was made with two different BBC implementations, one by the authors and one by Dr. Johnson [Johnson 1999; Stockinger et al. 2002].

[14]The size reported for the commercial DBMS are the actual bytes used, not the total size of the pages occupied.

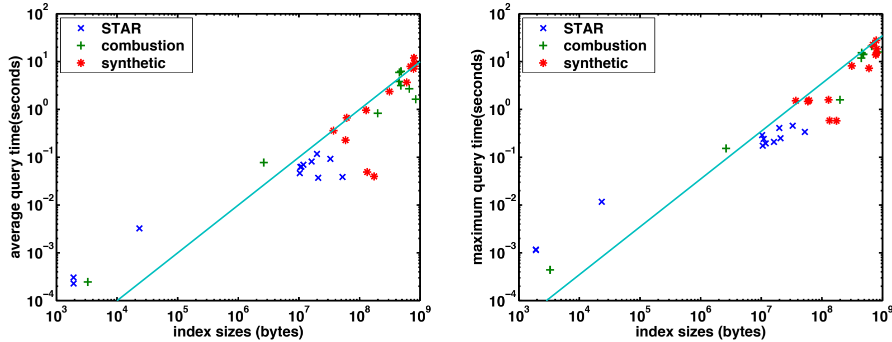


Fig. 12. The average time and the maximum time needed to process a random range query using a WAH compressed bitmap index.

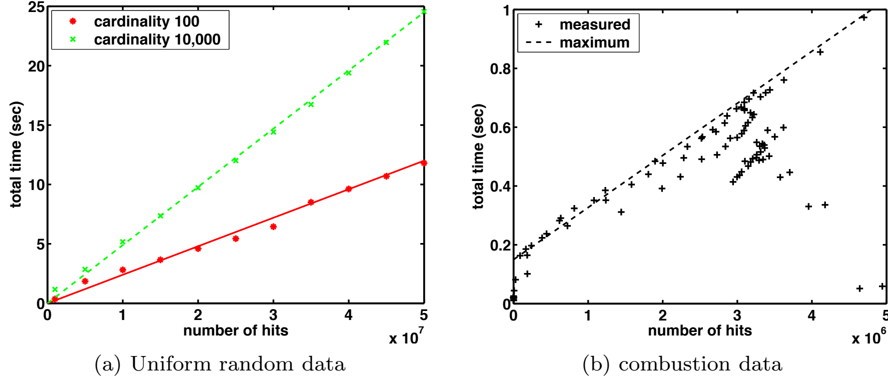
The bitmap index is known to be efficient for low cardinality attributes. For scientific data such as the one from STAR, where the cardinalities of some attributes are in the millions, there were doubts as to whether it is still efficient. In Figure 11(b), we see in many cases where compressed bitmap indices perform worse than projection indices. However, WAH compressed indices are more efficient than projection indices in all test cases. This shows that WAH compressed indices are efficient for both low and high cardinality attributes. Next, we demonstrate that the WAH compressed indices scales linearly.

7.2 Average query response time scales linearly

Proposition 9 indicates that the query response time is a linear function of the total size of bitmaps involved. In the worst case, half of the bitmaps are used in answering a query. Therefore the maximum query response time should be a linear function of the total size of the bitmap index. Similarly, we would also expect the average query response time to be a linear of the index size as well.

One thousand random queries were tested for each attribute. The average and the maximum query response time are reported in Figure 12. All four datasets are used in this test. Each point in Figure 12 is for one attribute. To illustrate how WAH compressed indices might behave on even larger datasets, we draw two lines to represent what can be expected in the average case and in the worst case. Since the maximum query response time is more closely related to the index size, we see that the points in Figure 12(b) are indeed close to the trend line than the points in Figure 12(a).

Next, we make some quantitative observations based on Figure 12 to show that indeed only half of the bitmaps are read and WAH compress indices perform better than projection indices. Let S denote the index size in bytes, based on regression, the average query response time was $10^{-8}S$ seconds on the particular test machine, and the maximum query response time was $3.5 \times 10^{-8}S$ seconds. From earlier tests shown in Table I, we know that the average reading speed is about 16 MB/s. If the query response time is spent only to read bitmap indices, the measured worst



(a) Uniform random data (b) combustion data

Fig. 13. query response time using WAH compressed bitmap indices is at worst linear in the number of hits.

case time can be used to read about half of the compressed bitmaps[15]. 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.

7.3 Query response time is optimal

Next we verify the linear relation predicted in Proposition 10 between the query response time and the number of hits. Figure 13 shows the time required to answer random queries on two sets of data, Figure 13(a) for uniform random data and Figure 13(b) for the combustion data. Each data point is a single query response time measurement. The performance of bitmap indices for uniform random attributes is very close to the worst case; therefore, the points follow the expected straight lines. In general, the straight lines are expected 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 13(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 13(b) is slightly smaller than that for the random attribute with $c = 100$. This verifies that the worst case query response time is indeed linear in the number of hits. Therefore, the WAH compressed bitmap index is optimal for one-dimensional range queries.

[15]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 $\frac{\beta S}{16\times10^6}$ seconds. The measured time is $3.5\times10^{-8}S$ seconds. The two time values must be the same, $\frac{\beta S}{16\times10^6} = 3.5\times10^{-8}S$, which leads to $\beta = 16\times10^6\times3.5\times10^{-8} = 0.56$.

8. SUMMARY AND FUTURE WORK

In this paper, we present a compression scheme for bitmaps named the Word-Aligned Hybrid (WAH) code and demonstrate that it not only reduces the bitmap index sizes but also improves the query response time. We prove that WAH compressed bitmap indices are optimal for one-dimensional range queries by showing that the query response time is a linear function of the number of hits. While a number of indexing schemes such as B^+-tree indices and B^*-tree indices have this optimality property, the WAH compressed bitmap index is more useful because it is also efficient for arbitrary multi-dimensional range queries. On large high-dimensional datasets common to many commercial data warehouses and scientific applications, projection indices are known to be the most efficient for answering range queries. Our tests on cold caches, where data is always brought in from disk, show that WAH compressed bitmap indices never perform worse than projection indices. On average, WAH compressed bitmap indices are three times as fast as projection indices.

The bitmap index is known to be efficient for low cardinality attributes. A number of articles in trade journals have advised to not use bitmap indices for high cardinality attributes. With our analyses and performance measurements, we show that the WAH compressed bitmap index is also efficient for high cardinality attributes.

This paper concentrates on the efficiency of using compressed bitmap indices to answer multi-dimensional range queries. There are many open issues still to be investigated.

—How do bitmap indices perform on other types of queries, such as θ-joins, top-k queries, similarity queries and queries with complex arithmetic expressions?

—How to update the bitmap indices in the presence of frequent updates? One approach might be to mark the bits corresponding to the modified records in the bitmap indices as invalid, store the updated records separately, say, in memory, and update indices when the system is idle.

—How does one deal with the extreme case where every value is distinct? In this case, the compressed bitmap index may be three times as large as the normal case. Model 204 uses a compact RID list as an alternative to the uncompressed bitmap index when the attribute cardinality is high [O'Neil 1987]. It might be worthwhile to consider using a compact RID list instead of the compressed bitmap index as well.

—How to efficiently generate the bitmap indices and organize them on disk? Our implementation currently uses high-level I/O functions to lay out the index without considering the intrinsic paging and blocking structure of the file systems. Systematically studying the index generation process could be an interesting activity especially if the issues such as paging/blocking, recovery, and maintainability are also considered.

—How does one handle the frequent queries on the same set of attributes? For ad hoc range queries on high-dimensional data, generating one compressed bitmap index for each attribute is an efficient approach. However, if some attributes are frequently used together, it may be more efficient to use a composite index. A

WAH compressed version of this composite index should be optimal for range queries on the indexed attributes. Demonstrating this can also be interesting for future work.

—How to fully explore the design choices of different compression schemes for bitmaps? We can not expect any compression scheme to make the compressed bitmap indices scale better than WAH; however, there might be compression schemes with smaller scaling constants. In Section 3, we mentioned two potential ways of improving bitmap compression: one is to decompress some dense bitmaps and the other is to explore different ways of extending BBC to be word-aligned. There may be many other options.

—How do other compressed indices scale? There are some indications that BBC may scale linearly as well [Wu et al. 2004]. It should be an interesting exercise to formally prove that it indeed scales linearly. It is possible that all compression schemes based on run-length encoding have this property.

9. ACKNOWLEDGMENTS

The authors wish to express our sincere gratitude to Professor Ding-Zhu Du for his helpful suggestion of the inequality in Lemma 5, and to Drs. Doron Rotem and Kurt Stockinger for their help in reviewing the drafts of this paper.

This work was supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

REFERENCES

Amer-Yahia, S. and Johnson, T. 2000. Optimizing queries on compressed bitmaps. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, Eds. Morgan Kaufmann, 329–338.

Antoshenkov, G. 1994. Byte-aligned bitmap compression. Tech. rep., Oracle Corp. U.S. Patent number 5,363,098.

Antoshenkov, G. and Ziauddin, M. 1996. Query processing and optimization in ORACLE RDB. *VLDB Journal 5*, 229–237.

Chan, C.-Y. and Ioannidis, Y. E. 1998. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD: International Conference on Management of Data*. ACM press, 355–366.

Chan, C. Y. and Ioannidis, Y. E. 1999. An efficient bitmap encoding scheme for selection queries. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, A. Delis, C. Faloutsos, and S. Ghandeharizadeh, Eds. ACM Press, 215–226.

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

Comer, D. 1979. The ubiquitous B-tree. *Computing Surveys 11,* 2, 121–137.

Czech, Z. J. and Majewski, B. S. 1993. A linear time algorithm for finding minimal perfect hash functions. *The Computer Journal 36,* 6 (Dec.), 579–587.

Fox, E. A., Chen, Q. F., Daoud, A. M., and Heath, L. S. 1991. Order-preserving minimal perfect hash functions and information retrieval. *ACM Trans. Inf. Syst. 9,* 3, 281–308.

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

GAILLY, J. AND ADLER, M. 1998. *zlib 1.1.3 manual*. Source code available at http://-www.info-zip.org/pub/infozip/zlib.

ISHIKAWA, Y., KITAGAWA, H., AND OHBO, N. 1993. Evalution of signature files as set access facilities in OODBs. In *Proceedings ACM SIGMOD International Conference on Management of Data, May 26-28, 1993, Washington, D.C.*, P. Buneman and S. Jajodia, Eds. ACM Press, 247–256.

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

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

KNUTH, D. E. 1998. *The Art of Computer Programming*, 2nd ed. Vol. 3. Addison Wesley.

KOUDAS, N. 2000. Space efficient bitmap indexing. In *Proceedings of the ninth international conference on Information knowledge management CIKM 2000 November 6 - 11, 2000, McLean, VA USA*. ACM, 194–201.

LEE, D. L., KIM, Y. M., AND PATEL, G. 1995. Efficient signature file methods for text retrieval. *IEEE Transactions on Knowledge and Data Engineering 7,* 3, 423–435.

MOFFAT, A. AND ZOBEL, J. 1992. Parameterised compression for sparse bitmaps. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval, Copenhagen, June 1992*, N. Belkin, P. Ingwersen, and A. M. Pejtersen, Eds. ACM Press, 274–285.

O'NEIL, P. 1987. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems, Asilomar, CA*. Lecture Notes in Computer Science, vol. 359. Springer-Verlag, 40–59.

O'NEIL, P. AND O'NEIL, E. 2000. *Database: pronciples, programming, and performance*, 2nd ed. Morgan Kaugmann.

O'NEIL, P. AND QUASS, D. 1997. Improved query performance with variant indices. In *Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, J. Peckham, Ed. ACM Press, 38–49.

O'NEIL, P. E. AND GRAEFE, G. 1995. Multi-table joins through bitmapped join indices. *SIGMOD Record 24,* 3, 8–11.

SHOSHANI, A., BERNARDO, L. M., NORDBERG, H., ROTEM, D., AND SIM, A. 1999. Multidimensional indexing and query coordination for tertiary storage management. In *11th International Conference on Scientific and Statistical Database Management, Proceedings, Cleveland, Ohio, USA, 28-30 July, 1999*. IEEE Computer Society, 214–225.

STOCKINGER, K., DUELLMANN, D., HOSCHEK, W., AND SCHIKUTA, E. 2000. 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*.

STOCKINGER, K., WU, K., AND SHOSHANI, A. 2002. Strategies for processing ad hoc queries on large data warehouses. In *Proceedings of DOLAP'02*. McLean, Virginia, USA, 72–79. A draft appeared as tech report LBNL-51791.

WONG, H. K. T., LIU, H.-F., OLKEN, F., ROTEM, D., AND WONG, L. 1985. Bit transposed files. In *Proceedings of VLDB 85, Stockholm*. 448–457.

WU, K., KOEGLER, W., CHEN, J., AND SHOSHANI, A. 2003. Using bitmap index for interactive exploration of large datasets. In *Proceedings of SSDBM 2003*. Cambridge, MA, USA, 65–74. A draft appeared as tech report LBNL-52535.

WU, K., OTOO, E. J., AND SHOSHANI, A. 2001. A performance comparison of bitmap indexes. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, November 5-10, 2001*. ACM, 559–561.

Wu, K., Otoo, E. J., and Shoshani, A. 2002. Compressing bitmap indexes for faster search operations. In *Proceedings of SSDBM'02*. Edinburgh, Scotland, 99–108. LBNL-49627.

Wu, K., Otoo, E. J., and Shoshani, A. 2004. On the performance of bitmap indices for high cardinality attributes. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, Eds. Morgan Kaufmann, 24–35.

Wu, K., Otoo, E. J., Shoshani, A., and Nordberg, H. 2001. Notes on design and implementation of compressed bit vectors. Tech. Rep. LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA.

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

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

Ziv, J. and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23,* 3, 337–343.

Appendix: Algorithms to perform logical operations

Listing 1. Given two bitmaps x and y, perform an arbitrary bitwise logical operation (denoted by ∘, can be any binary logical operator, such as, AND, OR, and XOR) to produce a bitmap z.

```
z = generic_op(x, y) {
Input: Two bitmap 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, yrun.nWords = 0;
        ELSEIF (yrun.isFill)
            z.active.value = yrun.fill ∘ *xrun.it,
            z.appendLiteral(),
            -- yrun.nWords, xrun.nWords = 0;
        ELSE
            z.active.value = *xrun.it ∘ *yrun.it,
            z.appendLiteral(),
            xrun.nWords = 0, yrun.nWords = 0;
    }
    z.active.value = x.active.value ∘ y.active.value;
    z.active.nbits = x.active.nbits;
```

```
}
```

LISTING 2. Given two bitmaps x and y, perform a bitwise logical OR operation. The bitmap x is assumed to be uncompressed and the result is written back to x.

```
inplace_or(x, y) {
Input: Two bitmap x and y containing the same number of bits and x is uncompressed.
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;
};
```

LISTING 3. Data structure (classes) to store the WAH compressed bitmaps.

```
class bitmap {
    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
    };
};

bitmap::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
}
```

bitmap::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 4. An auxiliary data structure used in Listings 1 and 2.

```
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() { // nWords and fill not used if isFill=0
            isFill = (*it > 0x7FFFFFFF),
            nWords = (*it & 0x3FFFFFFF),
            fill = (*it≥0xC0000000?0x7FFFFFFF:0);
        }
};
```