LBNL/PUB-3161

Notes on Design and Implementation of Compressed Bit Vectors

Kesheng Wu[‡] Ekow J. Otoo[‡] Arie Shoshani[‡] Henrik Nordberg[‡]

September 27, 2001

Abstract

Many bitmap based indexing schemes have been shown to work effectively in database applications. To further improve their effectiveness, we study compression schemes that have the potential to reduce the index size without increasing query processing time. A bitmap index is stored as a collection of bitmaps and the most frequent operations on these bitmaps are bitwise logical operations. Since we are interested in using these indices on large databases, reducing the sizes of the indices is essential. In addition, we also want to be able to perform logical operations on the compressed bitmaps efficiently. This note contains a study on a number of generic lossless compression algorithms and a number of specialized compression schemes. Generic compression schemes, such as gzip, are effective in reducing the storage requirement, but don't support faster bitwise logical operations. Most specialized bitmap compression schemes are byte based, i.e., they access memory one byte at a time, and can not take full advantage of today's computing hardware that supports fast word operations. In addition to studying the performance of these schemes, we also introduce a number of word-based compression schemes with the expectation that they may support faster logical operations than the byte-based schemes. What is surprising is that the word based scheme can be dozens of times faster than the byte based ones. In many cases, our word based compressed bitmap also perform bitwise logical operations faster than the uncompressed bitmaps.

1 Introduction

Recently, using bitmap indices in database applications has gained considerable research and commercial interests [2, 20, 30]. The term bitmap index is used here to refer to a broad category of indexing schemes that store their indices as bit sequences and use bitwise logical operations as the primary means to extract answers from the indices. In general, bitmap index is effective for data warehouses and similar applications where the databases are read-mostly and the user queries usually involve more than one attribute [6, 7, 20, 30]. Many commercial databases, e.g., Oracle 8, IBM DB2, Sybase IQ, now implement some form of the bitmap index scheme in addition to the more traditional B-tree based schemes. The credit of popularizing this type of bitmap index is often attributed to O'Neil who successfully demonstrated its usefulness in his database Model 204 [20]. Other bit sequence based schemes for database systems include bit transposed files [29] and

[‡]Lawrence Berkeley National Laboratory/NERSC, Berkeley, CA 94720. Email: {kwu, ejotoo, ashoshani, hnordberg}@lbl.gov.

signature files [9, 10, 12, 22]. In addition to being a useful technique for database systems, similar techniques are also extensively used in other applications such as information retrieval [4, 5, 8, 13].

To illustrate how bitmap index is generated and used, we show a small example in Figure 1. This figure only shows two attributes each with eight values. The attribute **R** contains categorical values. There are four distinct categories and a bit sequence is produced for each of the four values to indicate whether or not $\mathbf{R}_i$ equals to the specified value. For example, the ith bit in sequence b_1 indicates whether $\mathbf{R}_i = A$. The attribute **X** contains integer values, this example divides its domain into four ranges (also called bins) and use one bit sequence to indicate whether $\mathbf{X}_i$ falls in a range. We have labeled the bit sequences b_1 through b_8 in Figure 1. Using these bit sequences, it is easy to answer some range queries. For example, query "$\mathbf{R} = B$ AND $\mathbf{X} < 4$" can be translated to the following logical operations among b_2, b_5 and b_6:

$$a = b_2 \text{ AND } (b_5 \text{ OR } b_6),$$

where a is the bit sequence storing the result. If the ith bit of a is 1, then the ith tuple (row) of the database qualifies the query. It is easy to see that the efficiency of bitwise logical operation is a crucial to the overall effectiveness of the bitmap indexing scheme.

The above example is tiny. Database in real word are much larger. For instance, in a specialize database system for some high energy physics experiment, the database contains hundred of attributes and millions of tuples [25, 26]. In the future, this database may eventually contain hundreds of millions of tuples. A bitmap index for this database may contain thousands of bit sequences. Since each bit sequence has one bit for every data object, see Figure 1, each bit sequence contains hundreds of millions of bits. Other database systems such as those involved in biometric identification and global climate modeling may need indices of similar size. Storing these bit sequences in a minimal amount of computer memory is a crucial requirement on the database system. The basic strategy for reducing storage requirement is compression [16, 18, 28]. There are well studied general purpose algorithms such as gzip and bzip2, and mature software packages implementing these algorithms [14, 24]. These generic compression schemes are effective in compressing the bit sequences, but bitwise logical operations on the compressed bit sequences are much slower than operating on their uncompressed versions. Since the logical operations are the crucial operations during query processing, in order to enhance the overall efficiency of the database system, the logical operations need to be as fast as possible. To achieve this goal, we need to use compression schemes that can support faster bitwise logical operations. This note records our experiences of searching for these schemes.

The Standard Template Library (STL) of C++ language supports two schemes to represent bit sequences, namely `vector<bool>` and `bitset` [27]. The container type `vector<bool>` is more suitable for storing long bit sequences. The schemes implemented in STL typically use one bit of computer memory to represent each bit of a bit sequence. The STL standard does not require the data to be compressed and does not provide any bitwise logical operations. For these reasons, specialized compressed bit sequence storage schemes are needed. The computer data structure we use to represent these bit sequences is called a *bit vector*. We use this term rather than bitmap to emphasis that the logical operations are crucial part of the data structure.

A straightforward way of representing such a bit sequence is to use one bit of computer memory to represent each bit of the sequence. We call this a literal bit vector. A simple idea of compressing a bit sequence is to break it into a series of smaller sequences of consecutive identical bits. In

		bitmap index			
		b_1	b_2	b_3	b_4
i	$\mathbf{R}_i$	A	B	H	W
1	W	0	0	0	1
2	B	0	1	0	0
3	W	0	0	0	1
4	H	0	0	1	0
5	W	0	0	0	1
6	W	0	0	0	1
7	B	0	1	0	0
8	W	0	0	0	1

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

Figure 1: Two sample bitmap indices for two attributes, **R** and **X**.

this note, we call a sequence of identical bits a *fill*. Since each fill can be recorded with a counter representing its length plus one bit indicating the actual bit values, this representation may use less space compared to the literal version. It is know as the *run-length encoding*. If a fill is long enough, then this run-length encoding will use less space than the literal representation. Most schemes to be discussed here represent long fills using this run-length encoding and represent the remaining bits literally. They differ in details of how to represent the lengths and how to organize the literal bits. When performing a logical operation, working directly with the counters ensures that a minimum amount of computer memory is used to represent the operands and the result. Compared to the alternative of decompressing the operands first then operating on the uncompressed bit vectors, operating directly on the compressed data needs less time because it avoids producing the intermediate uncompressed versions. In this paper, we will focus on the type of compressed bit vectors that can support this type of *direct logical operations*.

In terms of compression effectiveness, we expect these specialized compression schemes to be less effective than the generic compression schemes, e.g., gzip [14] and bzip2 [24], because gzip and bzip2 are more complex compression algorithms. However, by using simpler algorithms, we should be able to achieve faster logical operations. As in many cases, we trade space for speed. What we strive for is to trade a small amount of space for a large gain in speed.

We are aware of one prior study on the performances of logical operations on compressed bit vectors [11]. In that study, the author compared a number of schemes including Byte-aligned Bitmap Code (BBC) and gzip. We extend this previous study in two aspects: we introduce the word-aligned schemes which have not been studied before, and we study more carefully how to directly perform logical operations on the compressed bit vectors. In a large database system, the indices are stored on disk as files. Each time a bit vector is used, some IO operations are likely also involved. For this reason, we also carefully measure the IO time associated with each bitwise logical operation.

The remainder of this note consists of five sections. In the first three, we discuss three types of bit vector schemes, namely, the literal scheme (including gzip compressed literal scheme), byte based schemes, and word based schemes. Each of these three sections also contains small amount of performance information on the methods described. We present some overall comparison and analyses in Section 5 and a short summary in Section 6.

2 Literal bit vector and generic compression schemes

In this section, we will briefly describe the literal version of the bit vector and explore how to use the generic lossless compression programs to reduce the size of the files. The literal version of the bit vector uses one bit of computer memory to represent each bit of a bit sequence. For the example shown in Figure 1, eight bits would be needed to represent each of the bit sequences. In a typical computer memory system, bits are organized into bytes and words. Though a word typical contains several bytes, most likely it takes the same amount of work to access either a word or a byte [23]. In fact it may be slower to access a byte than to access a word. This is because most CPUs can only operate on data one word at a time. In our implementation of this literal scheme, we choose to only access words. We store the first bit of the bit sequence in the Most Significant Bit (MSB) of the first word. The rest of the bits are stored consecutively in computer memory. The last word may be partially filled and another word is needed to indicate how many bits are actually used. We refer to this last word as the *active word*. All versions of the bit vector need to have this word in order to deal with the last few bits that do not fill up a whole word. The active word is always represented literally and it is stored separated from the others. Our future discussions will concentrate on the regular words of the bit vectors that take up most of the space. To simplify memory management, we store these regular words in a STL vector container [27]. The following algorithm describes how a bitwise logical operation can be performed. The subscripting operator [] and functions *size*(), *back*(), and *push_back*() are part of STL vector container.

ALGORITHM **1** To perform a bitwise logical operation between two literal bit vectors x and y, and store the result in z. Symbol $\circ$ denotes one of the three logical operations, AND, OR, or XOR.

for $(i = 0;\ i < x.vec.size();\ ++i)$ **do** $z.vec[i] = x.vec[i] \circ y.vec[i];$ **od**
$z.active.nbits = x.active.nbits;$
$z.active.value = x.active.value \circ y.active.value.$

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

It is clear that the logical operations between two literal bit vectors are very simple. Common logical operations, such as AND, OR and XOR, are supported by computer hardware. When all three bit vectors x, y and z can be stored in a computer's first level cache, the main loop may take only one clock cycle per iteration. When three bit vectors can't fit in the cache but can fit into the main memory, it may take a number of clock cycles, typically less than 10, per iteration.

No. of bits	10^7	10^8	10^9
No. of bytes	1.25×10^6	1.25×10^7	1.25×10^8
read only (sec)	0.020	0.195	2.00
read to bit vector (sec)	0.048	0.490	4.89
logical operation (sec)	0.010	0.127	1.280

Table 1: Performance information of the literal (LIT) version of bit vector.

For convenience of testing, we have chosen to use a set of bit sequences where each bit is generated independently from the preceding ones and each has the same prescribed probability of being one[1]. Three different sequence lengths and ten different probabilities are used. For each probability value we generated two bit sequences. We name this set of 60 bit sequences the *random test data set*. Since this set is close to what was used in the earlier comparison study [11], we will use it as our primary test data. Most probability values are less than 0.5 because all the compression schemes to be discussed work equally well when all bits of a sequence are complemented. In other words, before and after a bit sequence is complemented, the bit vector occupies the same amount of storage and bitwise logical operations take the same amount of time.

Table 1 shows some performance information about the literal bit vector implementation (LIT for short). It is clear that memory requirement of this implementation is directly proportional to the number of bits it represents. In fact, only the sequence length affects the performance of the operations. The timing results reported in this table are obtained on a Sun Enterprise 450[2] using a 400MHz UltraSPARC-II processor, and the files reside on a file system consisting of five disks connected to the internal UltraSCSI controller and managed by a VERITAS Volume Manager[3]. The machine has four gigabytes (GB) of RAM which is large enough to store all test programs in memory. The CPU has a 16 KB first level data cache and 4 MB second level external cache. The smallest test cases can barely fit into the second level cache, but the large ones cannot. We will only report timing results on this machine because tests on other type of machine produce similar relative performances among the different bit vector schemes. Johnson has also observed the same trend during his study of bitmap compression schemes [11].

Table 1 contains three timing results, (1) time to read a file one page at a time and discard the content, marked "read only" (2) time to read a file and form a bit vector from the file content, marked "read to bit vector", and, (3) time to perform a single bitwise logical operation between two bit vectors. In this implementation, the three logical operations AND, OR, and XOR, take the same amount of time. The timing results reported in Table 1 are recorded using `getrusage` function which is also used in all future tests unless specifically noted otherwise. The granularity of the timing function of the solaris operating system is 0.01 second. Each time a read operation is performed, it is performed five times and the average read time is recorded. The read time reported

[1] The random numbers are generated using a pseudorandom number generator called Mersenne Twister [15] because it is faster than others we have tried.

[2] More information about the E450 is available at `http://www.sun.com/servers/workgroup/450`.

[3] More information about VERITAS Volume Manager is available at `http://www.veritas.com/us/products/-volumemanager`.

in Table 1 are averages of 100 such runs. Each time a logical operation is performed, the operation is repeated so that it is equivalent to working on a bit sequence with at least 10^9 bits and the average time is reported. This arrangement tends to to produce stable timing results but hide the initial startup cost involved in reading a file or performing a logical operation.

The results in Table 1 indicate a reading speed of about 60MB/s which is within the capability of the IO system but is considerably higher than those quoted in some database textbooks [21]. The logical operation time on two 10^8-bit sequences is more than 10 times that on two 10^7-bit sequences because the data involved in the shorter bit sequences can fit into the 4MB data cache of the UltraSPARC-II processor but not the longer ones.

We don't report the time needed to write the bit vectors because writing is often faster than reading due to buffering. More importantly, when used to represent the bitmap indices, these bit vectors are much more likely to be read than be written. The variations in the measured time for logical operations are usually only a few percent of the average reported. However, because the complexity of the IO system and the fact that other programs may be using the IO system at the same time when we were conducting the tests, the standard deviations of the read time are as large as the averages reported in Table 1. Reading a file and forming a bit vector takes about twice as long as the simply reading the file because we need to copy the content into the vector container. Replacing the STL vector with a custom data structure that avoids this copying would remove most of the overhead in forming the literal bit vector. We verify this in the latter part of this note. Ideally, a good compressed bit vector should use less memory (no. of bytes) and perform logical operations in less time than this literal scheme.

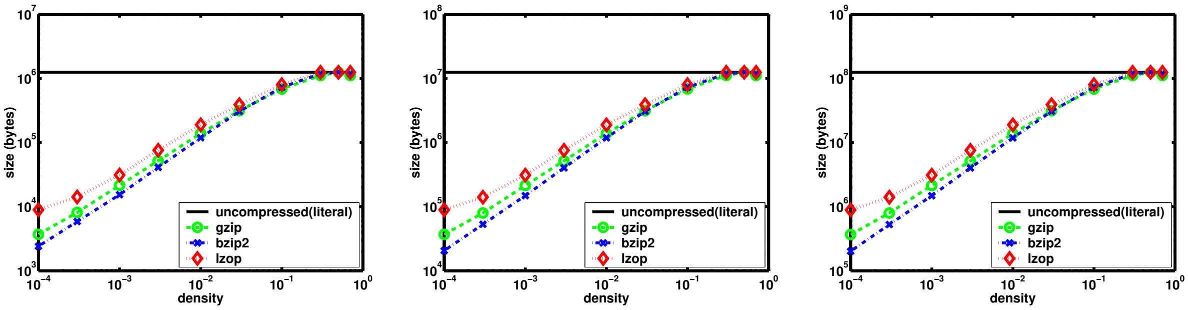
Compression tools such as gzip [14] are commonly used to reduce file sizes. They are simple to use and quite effective in reducing the file size. For compressing bit vectors, we can only use lossless compression schemes. After some search, we found three compression programs that are potentially good candidates for our use.

- **gzip**. This is one of the most popular compression utilities [14]. It uses Lempel-Ziv coding (LZ77) which is known to be asymptotically optimal. It is freely available and it also comes with a set of easy to use API (zlib) for performing bitwise logical operations.

- **bzip2**. This is another popular one. It uses Borrows-Wheeler text compression and Huffman coding [24]. Its documentation claims that it is often faster then gzip and even compresses better than gzip in many cases. It also offers similar API as zlib.

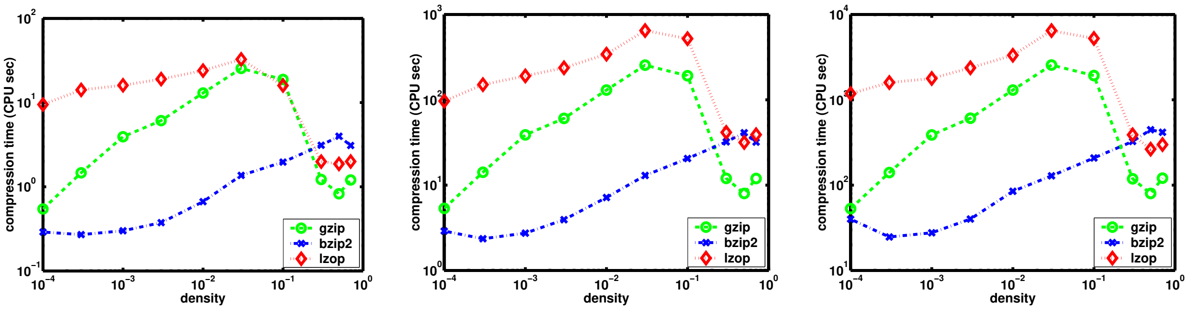
- **lzop**. This one implements a modified version of the Lempel-Ziv scheme [19]. The decompresser is very efficient and it is able to achieve a significant fraction of the speed of `memcpy()` function. This makes it particularly attractive because the decompresser is likely to be used more often than the compressor.

Figure 2 shows how well the three compression programs work on the test bit sequences. In this figure, the horizontal axes of the three plots are the bit density, i.e., fraction of bits that are set, and the vertical axes are the sizes of the files containing the compressed data. When the bit density is low, the compression programs are very effective; the compressed files are much smaller than uncompressed files. Let *compression ratio* be the ratio of file sizes after compression to the



(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 2: Sizes of the bit vectors compressed with the three compression packages.



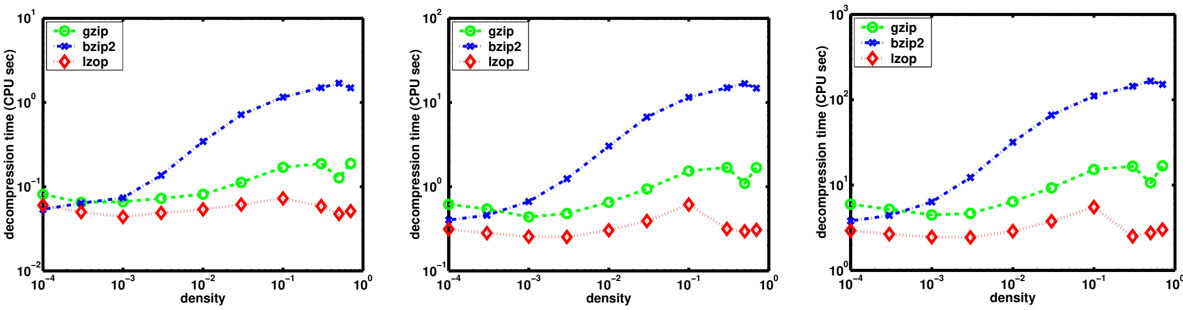
(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 3: CPU time (seconds) used to compress the bit vectors with the three compression utilities.

file size before compression. When bit density is 10^{-4}, the compression ratios are: gzip 0.003, bzip2 0.002, and lzop 0.007. As bit density increases, all three programs become less effective. Among the three, bzip2 appears to be most effective in compressing the test data and lzop the least effective. The compression ratios do not exhibit any significant change when more bits are stored.

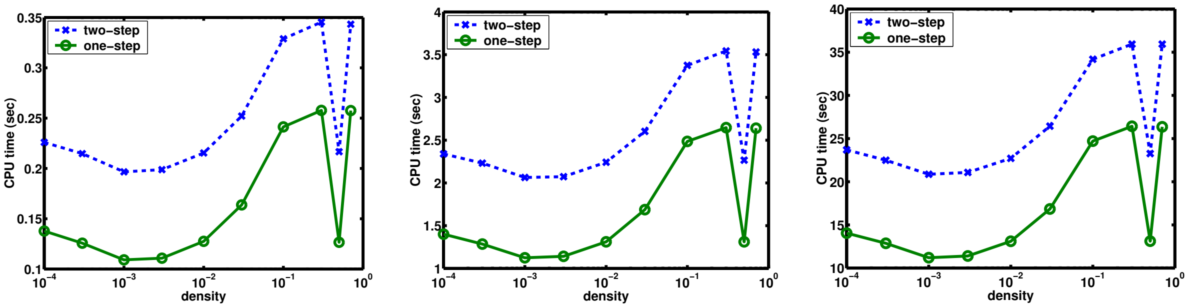
Figure 3 shows the CPU time[4] used to compress the test data set. The compression programs take very different amount of time to compress different bit vectors. In addition, the three programs used very different amount of time to compress the same file. In seven out the 10 test cases, bzip2 uses less time to compress the bit vectors. However, for the three high density bit sequences, bit density 0.3, 0.5 and 0.7, bzip2 uses more time than the other two and gzip uses the least amount of time on these files. With high bit densities (0.3, 0.5, and 0.7), the compressed files generated by all three programs are slightly larger than the uncompressed files because of the compression overheads. We say these bit vectors as incompressible.

[4]The time reported are the total execution time of running gzip, bzip2 and lzop, which include reading the uncompressed file, compress the data and write out the compressed file. Time values are recorded by `tcsh` internal function `time` which only prints two decimal points for fractions of a second. The CPU time is the sum of user time and the system time.



(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 4: CPU time (seconds) used to decompress the bit vectors with the three compression utilities.



(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 5: CPU time used to read two gzipped literal bit vectors from files and perform bitwise OR.

Figure 4 shows the CPU time[5] used to decompress the files. In most cases, lzop is considerably faster than gzip and bzip2. The decompression time of lzop and gzip show only small changes as bit density varies, the ratio of the longest decompression time and the shortest decompression time for lzop is about two and the same ratio is about three for gzip. However the decompression time of bzip2 shows a clear upward trend as bit density increases. In this case, decompressing a bit sequence with bit density 0.5 take about 40 times longer than decompressing a bit sequence with bit density 10^{-4}. The time used by lzop to decompress the files are comparable with the time needed to read in the uncompressed file, see Table 1. In fact, it takes only about $2.5-5$ seconds to decompress 1 billion bits, see Figure 4, and it takes about 2 seconds to read the uncompressed file, see Table 1. The decompression speed of lzop is indeed impressive. However, since it does not provide a convenient API for decompressing the files one piece at a time, we only use gzip to conduct the next set of tests.

Figure 5 and Table 2 show timing results of two ways of processing the compressed bit vector files and performing bitwise logical operations on them. The first approach does this in two steps:

[5]The time reported are the total execution time of running gzip, bzip2 and lzop, which include reading a compressed file, decompress the data and write out data to `/dev/null`, see also previous footnote.

density	10^{-4}			10^{-3}			0.5		
No. bits	10^7	10^8	10^9	10^7	10^8	10^9	10^7	10^8	10^9
two-step	0.226	2.339	23.69	0.197	2.064	20.87	0.217	2.266	23.27
one-step	0.138	1.401	14.05	0.109	1.124	11.20	0.127	1.310	13.12

Table 2: CPU time used to read two gzipped literal bit vectors from files and perform bitwise OR.

(1) read two files containing two operands of the logical operation and decompress the data into two literal bit vectors, and (2) perform the logical operations on these two bit vectors. In Figure 5 and Table 2, this approach is marked as the "two-step" approach. The second approach reads the two files one piece at a time using zlib functions, perform the bitwise operation on the portion of data and store the results in a bit vector. This is marked at the "one-step" approach in Figure 5 and Table 2. If only one logical operation is performed, the "one-step" approach clearly needs less time, about 40% less, because it doesn't form the uncompressed versions of the operands. However, if the same operands are used elsewhere in the program, it might be worthwhile to use the "two-step" approach and store the bit vectors in memory so that the later operations do not need to read the files again. In this set of test data, if the operands are used twice, then it is worthwhile to save them in memory.

The logical operations produce uncompressed bit vectors. If one also needs to compress the results, add the decompression time shown in Figure 4. In all future discussions, we will consider compressed bit vectors that enable direct logical operation where the operands of the logical operations are compressed and the results are also compressed.

3 Byte based schemes

When used to store images, the smallest unit of data accessed by most of the image formats is a byte. For this reason, most existing bitmap compression schemes are byte based. As in the literal scheme, the last few bits of the bit sequence are always stored literally along with a counter. The only difference is that the value is stored in a single byte. We refer it as the *active byte*. The remaining bits can be stored in a compressed form. Among these compressed schemes, we will review three that appear to be well suited for representing bitmap indices, Byte-aligned Bitmap Code (BBC) [1], PackBits [3] and a modified version of PackBits. BBC is very effective in compressing bit sequences. PackBits is much simpler than BBC, therefore logical operations between two PackBits bit vectors may be faster. The last scheme combines some advantages of the first two. All three schemes are byte aligned, that is, there is no need to break up any byte during a bitwise logical operations, not even the active byte. In addition, it is easy to construct functions to perform logical operations without converting the compressed bit vector into the literal form. If the compressed bit vectors are much smaller than the literal versions, operating on the compressed ones directly will access less memory and may take less time than operating on the uncompressed versions.

In reading the descriptions of the compressed schemes, it is helpful to assume a literal version of the bit sequence is available before compression. To help visualize the compressed scheme, we use a 160-bit sequence as an example, see the first line of Figure 6 for the hexadecimal version of this sample sequence. This sample sequence can be thought as an extension of bit sequence b_6 in Figure 1. All compression schemes divide the bit sequences into short sequences called runs. Each scheme will define its own runs and encode them differently.

3.1 PackBits (PAC)

This is a simple scheme designed for compressing bitmap images [3]. The PackBits scheme first divides a long bit sequences into bytes and then groups the consecutive bytes into two type of runs: literal runs and fill runs. A *literal run* can have 0 – 127 bytes of arbitrary values. A header byte is used to indicate the run length, i.e., the number of bytes in the run. A *fill run* can represent 2 – 129 bytes of the same value. It is represented by two bytes, a header byte and the literal value of the bytes. The header byte is an unsigned integer that equals to the run length plus 126. This scheme does not require all the bits in the fill to be the same, only that all the bytes be the same. Because each run of PackBits represents multiple of 8 bits, we say this scheme is byte-aligned.

ALGORITHM **2** To append a literal byte to a list of regular encoded bytes stored in a vector container called *vec*. The active byte is used as a place holder for this literal byte.
When the bit vector is empty, the vector container *vec* is empty. In this case, set the first byte in *vec* to be 1 ($vec.push\_back(1)$), the second byte to be the value of the active byte ($vec.push\_back(active.value)$), and $lastHeader = 0$. Otherwise, do the following. (Operator && indicates logical AND operation.)

```
if active.value = vec.back(),
  then if vec[lastHeader] > 127  &&  vec[lastHeader] < 255, then ++vec[lastHeader];
       elsif vec[lastHeader] = 1, then vec[lastHeader] = 128;
       elsif vec[lastHeader] < 127, then lastHeader = vec.size() − 1;
                                         vec.back() = 128;
                                         vec.push_back(active.value);
                                    else lastHeader = vec.size();
                                         vec.push_back(1),
                                         vec.push_back(active.value);
       fi
elsif vec[lastHeader] < 127, then ++vec[lastHeader];
                                  vec.push_back(active.value);
                             else lastHeader = vec.size();
                                  vec.push_back(1),
                                  vec.push_back(active.value).
fi

class activeByte {
      byte value;                      // the value of the active byte
      byte nbits;                      // the number of bits
};
class PackBits {
      std::vector<byte> vec;           // code words
      unsigned long lastHeader;        // index of the last header byte in vec
      activeByte active;               // the active byte
};
```

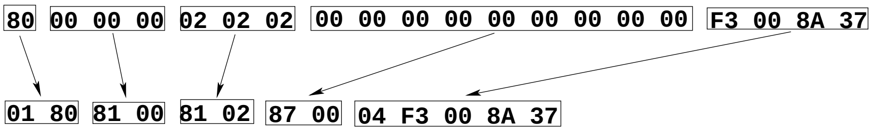


Figure 6: A PackBits bit vector.

Figure 6 shows an example of PackBits code, where the first line shows the literal version of the bit sequence as hexadecimal integers and the second line shows the PackBits representation. Each run is surrounded by a box. The first byte of each PackBits run is the header byte. The MSB of the header byte of a literal run is zero and the MSB of the header byte of a fill run is one. In this example, PackBits scheme uses 13 bytes which is seven bytes less than the literal version. This example demonstrates a simple fact that is true to most compressed bit vector schemes, it represents fill runs in less space than in the literal scheme, but represents literal runs in slightly more space due to compression overhead.

If all the bytes are the same, this scheme uses two bytes to represent every 129 bytes. This best case compression ratio is 0.0155. If all bytes are literal bytes, it uses 128 bytes to represent every 127 literal bytes, a 0.8% overhead. If a bit sequence consists of two bytes that are the same followed by a third byte that is different from the first two, this scheme needs four bytes to represent these three bytes. This worst case overhead is 33%.

The basic procedure of performing bitwise logical operation is to match up the bits from two operands and then use the bitwise logical operation supported by the computer hardware to produce the result. The following algorithm depicts the process of a bitwise logical operation. We have not given the algorithm to append a fill run to a PAC bit vector, but it easy to see that the process is similar to Algorithm 2. For this reason, adding a literal byte to the result is nearly as expensive as adding a whole fill run. We can count the complexity of Algorithm 3 by just counting how many literal words and how many fill runs are generated.

ALGORITHM **3** Given two PackBits bit vectors x and y, perform an arbitrary bitwise logical operation (denoted by $\circ$) to produce a bit vector z.

run $xrun, yrun$;
$xrun.it = x.vec.begin()$;
$yrun.it = y.vec.begin()$;
$xrun.decode()$; $yrun.decode()$;
while $x.vec$ and $y.vec$ are not exhausted, **do**
    $nBytes = \min(xrun.nBytes, yrun.nBytes)$;
    if $xrun.isFill$ && $yrun.isFill$,
      then append a fill run of $nBytes$ with value $(*(xrun.it) \circ *(yrun.it))$ to z.
    elsif $xrun.isFill$,
        then for $(i = 0;\ i < nBytes;\ ++i)$ **do**
            append a literal byte of value$(*(xrun.it) \circ *(yrun.it))$ to z;
            $++yrun.it$. **od**
    elsif $yrun.isFill$,
        then for $(i = 0;\ i < nBytes;\ ++i)$ **do**

```
                    append a literal byte of value(*(xrun.it) ∘ *(yrun.it)) to z;
                    ++xrun.it. od
            else for (i = 0; i < nBytes; ++i) do
                    append a literal byte of value(*(xrun.it) ∘ *(yrun.it)) to z;
                    ++xrun.it; ++yrun.it. od
        fi
        xrun.nBytes− = nBytes; xrun.nBytes− = nBytes;
        if xrun.nBytes = 0, then if xrun.isFill then ++xrun.it; fi
                                 xrun.decode(). fi
        if yrun.nBytes = 0, then if yrun.isFill then ++yrun.it; fi
                                 yrun.decode(). fi
od

class run { // used to hold basic info about a run
    std::vector<byte>::const_iterator it;      // pointer to byte values
    byte                              nBytes;  // number of bytes in the run
    int                               isFill;  // is it a fill run

    decode() {
        if *it < 127, then xrun.isFill = 0, xrun.nBytes = *it;
                 else xrun.isFill = 1, xrun.nBytes = *it − 126.fi
        ++it;
    }
};
```

3.2 Byte-aligned Bitmap Code (BBC)

The original version of this scheme was invented by G. Antoshenkov and patented by Oracle [1, 2]. Since this scheme always operates on bytes, it is faster than those that are not byte-aligned. An independent study [11] also shows that it compresses random test data very well. More specifically, BBC compresses almost as well as gzip. Bitwise logical operations on sparse bit vectors are usually faster using BBC coding than using gzip compression, it may even be faster than using the literal bit scheme.

Similar to PackBits coding scheme, BBC scheme also first organizes bits into bytes, but all runs of BBC are of the form of a fill followed by some literal bytes called a *tail*. All bits in a BBC fill must be the same. A fill is a 0-fill if all bits are zero, a 1-fill if all bits are one. The fill bit of a 0-fill is zero and the fill bit of a 1-fill is one. There are two types of BBC code, one-sided code that only compresses 0-fills and two-sided code that compresses both 0-fills and 1-fills. The one-sided version is suitable for sparse bit vectors. Since the application of interests to us [25, 26] will produce bit vectors that are not sparse, we have chosen here to study a two-sided version of the BBC code, see Figure 7 for an example. BBC encoding scheme is more complex than others described in this paper. To simplify the programming for the logical operations we modified the BBC code to reduce the number of run types to four.[6]

[6]This modified version of BBC uses about the same amount of time as gzip (one-step) when the bit density is around 0.01, see Figures 5 and 11. The location of this cross over point between gzip and BBC is similar to what was observed in [11]. In [11], the figure showing actual time of various schemes is based on a test setup the author named

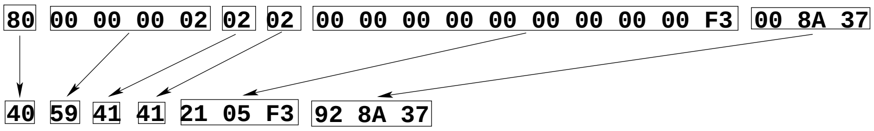


Figure 7: A BBC bit vector.

1. A type 1 run contains 0–3 bytes in the fill followed by 0–15 literal bytes. The header for this type of run is only one byte, and the eight bits of the header are

   `1 [fill bit] [fill length (2 bits)] [tail length (4 bits)].`

   The literal tail follows the header byte. The two bits representing fill length (number of bytes in the fill) and the four bits representing the tail length are to be interpreted as unsigned integers.

   The last three bytes of the example bit sequence form a type 1 run, see Figure 7, where the header byte written in binary form is `1 0 01 0010`.

2. A type 2 run contains 0–3 bytes in the fill followed by a tail of a single byte with one bit that is different from the fill bit of the run. The header for this run type is also one byte and the single byte tail is not stored. The layout of the header byte is

   `0 1 [fill bit] [fill length (2 bits)] [odd bit position (3 bits)].`

   Since there are only eight bit in a byte, three bits are enough to represent the position of the odd bit, the one that is different from the majority. The first four runs of the example shown in Figure 7 are of this type. The binary form of the first run is `0 1 0 00 000` which indicates that this run contains zero bytes of 0-fill followed by a literal byte whose odd bit is the 0th bit.

3. A type 3 run contains more than 3 bytes in the fill and 0–15 bytes in the tail. In this case, a multibyte counter is used to represent the fill length. The bytes in this multibyte counter follows the header byte of the form

   `0 0 1 [fill bit] [tail length (4 bits)].`

---

the "basic" scheme. Under that setup, only one of the two operands of a logical operation is compressed and the logical operation produces uncompressed result. In our tests, both operands of a logical operation are compressed, the gzip scheme produces uncompressed result but the BBC scheme produces compressed result. Since the compression and decompression take most of the time during the logical operations, our functions that perform logical operations on gzip compressed bit vectors perform about twice as much work as those used by Johnson. The functions on BBC bit vectors effectively perform more than twice as much work as those used by Johnson, however, because they perform direct logical operations the performance line for them actually crosses over the line for gzip at the same location. Our figures also include the IO time. However since both gzip and BBC compressed files are about the same sizes, the differences in the logical operation time should be attributed to the different the computation parts.

Each counter byte contains seven bits of significant information. The counter byte that is closer to this header byte is more significant than those are further away. The Most Significant Bit (MSB) of each byte is used to indicate whether there are more bytes to come. Every byte of the multibyte counter has this bit set except the last one. The literal value of the counter is formed by concatenating the lower seven bits of the counter bytes. The actual number of bytes in the fill is this literal value plus four.

The fifth run of the example shown in Figure 7 is this type. It contains nine zero bytes followed by a byte with value `F3` (hex). Only one byte is used to represent the fill length and the literal value of the this counter is the second byte in the run (`05` = 9 - 4).

4. A type 4 run contains more than 3 bytes in the fill followed by a tail of a single byte with one odd bit. This is a modification of the type 3 run. The multibyte counter is constructed in the same way. The header byte is defined as

   `0 0 0 1 [fill bit] [odd bit positin (3 bits)].`

Since all runs are a fill followed by a tail. If a new run starts with a literal byte, we assume the leading fill to be a zero-byte long 0-fill. Our modified version of BBC allows one to simply shift the bits in the header byte when the type of run changes during an operation. Constructing the algorithm for appending a new byte to a BBC bit vector is straightforward but laborious because the many cases involved.

Figure 7 shows an example BBC bit vector that encode the same 20 bytes shown in Figure 6. This time only 10 bytes are used. In this case, the BBC bit vector takes only half the space as the literal version, three bytes less than the PAC scheme.

This scheme can reduce storage because it uses multibyte counters for long fills and it does not use a separate byte to count the number of bytes in the tail. Further reduction of space is achieved by using only use three bits to represent each tail byte of a type 2 or type 4 run. If the majority of the bits are the same, BBC code can be very compact. If all bits are the same, it only needs a few bytes to represent the bit sequence, in other word, the minimum compression ratio can be arbitrarily close to zero. However, if the bit vector can not be compressed at all, BBC code uses 16 bytes to represent every 15 literal bytes, a 6.7% overhead.

3.3 PackBits with multibyte counter (PBM)

When a fill is longer than 129 bytes, PackBits scheme need to represent a long fill as multiple short fills. This limits the minimum compression ratio and make it less effective for representing bit sequences with long fills. An easy modification is to change the representation of fills so that the fill length is represented using a multibyte counter. Following the PackBits scheme, for a literal run, the header word is the number of literal bytes in the run. We also require all the bits to be the same in a fill run. The MSB of the header byte for a fill run is one and the second most significant bit is the fill bit. The third bit is used to indicate whether or not the length of the fill takes up any other bytes. If the fill has less than 32 bytes, it is zero, otherwise it is one. The lower five bits of the header are the five least significant bits of the integer representing the fill length. The

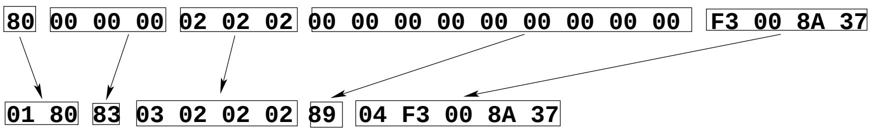
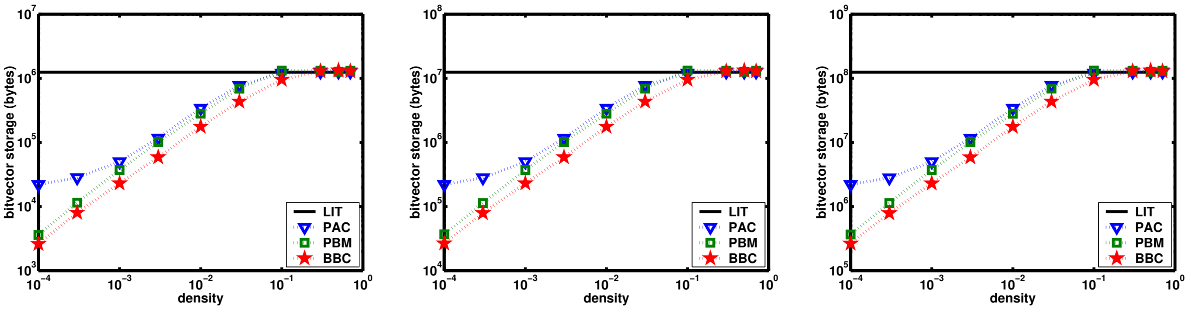


Figure 8: A PBM bit vector.



(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 9: Sizes of the byte-based compressed bit vectors containing bit sequence with specified bit density.

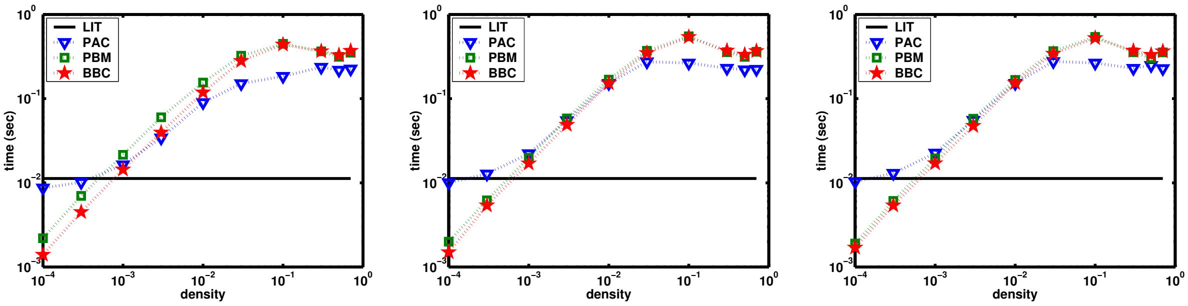
remaining bits of the fill length are written into bytes following the header where the lower seven bits of each byte are seven bits of the fill length and the MSB is one except the last byte.

Figure 8 shows an example of PBM bit vector. The 20 literal bytes in the example is represented by 13 bytes using this scheme. Compared with PackBits, this one uses more space to represent the three consecutive bytes with value `02`, but it uses less bytes to represent consecutive bytes that are `00`. The literal runs are the same in PackBits and PBM, see Figures 6 and 8. Compared to PackBits, the multibyte counter of this scheme decreases the best case compression ratio to zero. However, the example is too short to demonstrate this particular feature.

3.4 Performance

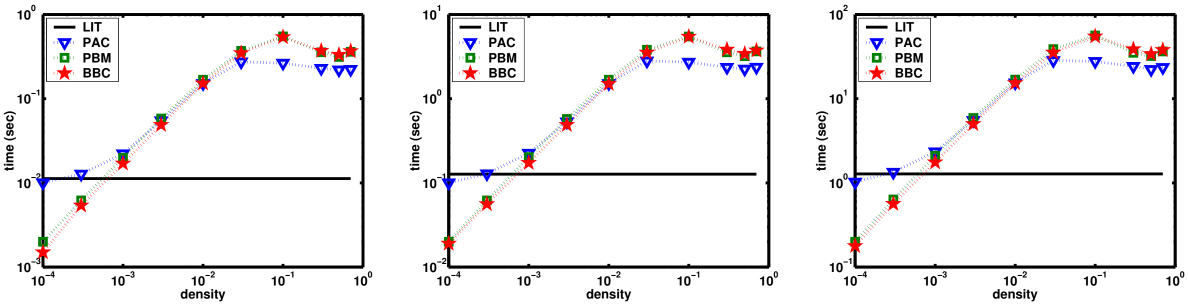
For a performance evaluation, we again tested our implementations on the random test data used earlier. Figure 9 shows the size information against the bit density. All three versions of the compressed bit vectors take up relatively small amount space when the bit density is low. When bit density is 10^{-4}, the compression ratios are: BBC 0.002, PAC 0.0175, and PBM 0.0028. At higher bit density, say, around 0.5, all three schemes use slightly more memory than the literal version because of their overheads. BBC compresses well when bit density is low, but it also has slightly higher overhead when the bit density is high.

Figure 10 shows the CPU time used to perform the three types of bitwise logical operations. One clear trend is that as bit density increases, all three bit vector schemes take more time. These compressed bit vectors are able to outperform the literal version only when bit density is very low, say $< 10^{-3}$. In these cases, operating on BBC bit vectors uses the least amount of time. When bit



(a) AND(&) (b) OR (|) (c) XOR (∧)

Figure 10: CPU time needed to perform a bitwise logical operation between two bit vectors containing 10 million bits.



(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 11: CPU time needed to perform a bitwise OR operation between two bit vectors of various sizes.

density is high, operating on PackBits bit vectors uses slightly less time. Overall the three logical operations have similar performance characteristics. The AND operation takes slightly less time compared to the other two logical operations especially when the bit densities are low because the resulting bit vector has lower bit density than that of the other two. The XOR operation takes slightly more time in some cases because the function has slightly more complex logic and the result of this operation may has more complex bit patterns.

Figure 11 shows the time to perform logical OR operations between two bit vectors. This is to demonstrate how the performances vary with number of bits. On this set of test data, the time used by the logical operations appears to be proportional to the number of bits stored. Table 3 shows the exact time used by the three programs to perform the logical operations on bit vectors of various densities. This table makes it easier to tell the differences among the different logical operations. For example, the time to perform AND operation in BBC scheme can be 25% lower than that of PBM when bit density is 10^{-4}. Having this table should also make it easier to compare with the performance of the other bit vectors schemes to be discussed later.

	density	10^{-4}			10^{-3}			0.5		
	No. bits	10^7	10^8	10^9	10^7	10^8	10^9	10^7	10^8	10^9
PAC	AND	0.009	0.087	0.878	0.010	0.102	1.068	0.218	2.218	22.148
	OR	0.010	0.101	1.028	0.013	0.129	1.354	0.220	2.255	22.342
	XOR	0.010	0.102	1.048	0.013	0.131	1.344	0.248	2.656	25.186

	density	10^{-4}			10^{-3}			0.5		
	No. bits	10^7	10^8	10^9	10^7	10^8	10^9	10^7	10^8	10^9
PBM	AND	0.002	0.021	0.228	0.022	0.216	2.150	0.314	3.301	33.034
	OR	0.002	0.020	0.200	0.020	0.203	2.104	0.314	3.206	32.142
	XOR	0.002	0.019	0.194	0.020	0.200	2.038	0.303	3.009	35.352

	density	10^{-4}			10^{-3}			0.5		
	No. bits	10^7	10^8	10^9	10^7	10^8	10^9	10^7	10^8	10^9
BBC	AND	0.001	0.015	0.148	0.014	0.144	1.432	0.329	3.374	34.050
	OR	0.002	0.019	0.178	0.017	0.173	1.756	0.333	3.421	33.960
	XOR	0.002	0.018	0.170	0.017	0.168	1.690	0.332	3.723	34.610

Table 3: Average CPU time (seconds) to perform one bitwise logical operation between two byte-based bit vectors.

4 Word based schemes

Most general purpose computers access memory by words even if only a single byte or bit is requested [23]. A word is usually either four bytes or eight bytes long. There are some machines that are able to perform the same operation on multiple bytes at the same time, e.g., Pentium with MMX technology. These machines may automatically group operations on bytes into operations on words and reduce the execution time. In order for this type of transformation to take pace, the operations on the bytes must have no dependency or very simple dependencies on each other. However, during bitwise logical operations, there are complex dependencies among the bytes of BBC and other byte based bit vectors. In this case, it is unlikely that we can rely on the hardware to speed up the logical operations on byte based bit vectors. One alternative to consider is to device word based schemes. In this section, we will describe four different word based schemes. Among the four schemes described next, the first two, hybrid run-length encoding (HRL) and word-aligned hybrid run-length encoding (WAH), are based directly on the run-length encoding, and the last two, packed word code (PWC) and word-aligned bitmap code (WBC), are based on the schemes discussed in the previous section.

4.1 Hybrid Run-Length encoding (HRL)

The run-length encoding breaks up a bit sequence into alternating 0-fill and 1-fill, and record the number of bits of each fill instead of the literal bit values [18]. The straightforward run-length scheme encodes each fill with a word. This encoding takes up less space only if the average fill length is larger than the number of bits in a word. In the hybrid run-length encoding scheme, a word can store either some literal bits or a fill. For convenience, a word that stores literal bit values is called a *literal word* and a word that stores the fill length and the fill bit is called a *fill word*. One bit is need to distinguish between a literal word and a fill word. In our implementation, the MSB of a literal word is zero and the MSB of a fill word is one. All lower bits of a literal word are literal bit values. The second most significant bit of the fill word is the fill bit and the remaining bits form an unsigned integer to represent the number of bits in the fill. If a word is 4-byte long, up to 2^{30} (about one billion) bits can be represented in one fill word. A pure run-length code may use significantly more memory than the literal version, however, this hybrid version typically use less memory.

In terms of implementation, we can still view a HRL bit vector as containing two internal parts, a STL vector of integers and an active word as in Algorithm 1. In HRL scheme, the active word may contain a fill. The following algorithm describes the process of appending a single bit to a HRL bit vector. In the following discussions, we refer to each integer in the STL vector as a code word.

ALGORITHM **4** Append a bit b to a HRL bit vector. Initially, when a bit vector is empty, vec is empty and $active.nbits = 0$. (Assuming an *unsingned* integer is 32-bit long. Operator $\ll$ indicates bitwise left shift.)

```
if active.nbits < 31,
  then append b to active.value;
elsif active.nbits > 31   &&   b is the same as the fill bit in active.value,
     then if active.nbits < 2^30 - 1,
              then ++active.nbits; ++active;
               else vec.push_back(active.value); active.value = b; active.nbits = 1;
          fi
elsif every bit of active.value is the same as b,
     then active.nbits = 32; active.value = ((2 + b) << 30) + 32;
      else vec.push_back(active.value); active.value = b; active.nbits = 1;
fi

class HRL {
     std::vector<unsigned> vec;
     activeWord active;                    // see Algorithm 1
};
```

Figure 12 shows how the sample bit sequence used in earlier examples is encoded in a HRL bit vector. Since this is a static picture, we have chosen to represent the encoding process differently. As in previous examples, the top line is the hexadecimal version of the bit sequence. The second line shows the fills and the third line shows how the fills are broken into runs that fit in 4-byte code words. In this case, a single bit was left in the active word. The last two words in the bit vector are for storing this single bit. The rest of the 159 bits are stored in four words. Note that the active word and the associated counter are always present in all bit vectors whether or not there are any useful bits in the active word, we have not shown them in the previous schemes because they do not contain useful information.

If all words of a bit sequence has to be stored literally, this scheme wastes one bit in each word, a 3% overhead for a 32-bit word. If all bits in the bit sequence are the same, this scheme can

```
literal bytes (in hexadecimal integers)
80 00 00 00 02 02 02 00 00 00 00 00 00 00 00 00 F3 00 8A 37

run−length code (in base 10 integers)
1*1 37*0 1*1 7*0 1*1 7*0 1*1 73*0 4*1 2*0 2*1 8*0 1*1 3*0 1*1 1*0 1*1 3*0 2*1 1*0 3*1

hybrid run−length code (in base 10 integers)
[1*1 30*0] [7*0 1*1 7*0 1*1 7*0 1*1 7*0] [66*0] [4*1 2*0 2*1 8*0 1*1 3*0 1*1 1*0 1*1 3*0 2*1 1*0 2*1] [1*1]

hybrid run−length code (in hexadecimal integers)
[40 00 00 00] [00 80 80 80] [80 00 00 42] [79 80 45 9B] [40 00 00 00 00 00 00 01]
```

Figure 12: A HRL bit vector.

represent $2^{30}-1$ bits in a 32-bit word. The best case compression ratio is $32/(2^{30}-1) \sim 3 \times 10^{-8}$. If a word contains 64 bits, the worst case overhead is $1/64 \sim 1.6\%$, and the best case compression ratio is $64/(2^{62}-1) \sim 1.4 \times 10^{-17}$.

An easy way of implementing a bitwise logical operation between two HRL bit vectors is to break up the code words into fills. The bitwise logical operations between fills also generate fills. The process of appending a fill to a HRL bit vector follows a similar process as shown in Algorithm 4. One significant part of the cost of performing a bitwise logical operation using this approach is the cost of breaking up the literal words into short fills. The operations on these short fills also generate short fills in the result. Reassembling these short fills into words needs a significant amount of time as well. The main motivation for developing a word-aligned version of this scheme is to see how much time can be saved by avoiding breaking up code words into fills.

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

This scheme is a modification of the previous one. A restriction is imposed on the fill words so that each fill word can be turned into integer number of literal words. We see this as the essence of the alignment requirement. In the previous section, we reviewed three byte-aligned schemes. In each case, a fill is required to be multiple bytes in length. A naive definition of word-alignment may be taken as the fill sizes have to be integer number of words. Using a word contains 32 bits as example, we may require all fill words to represent multiple of 32 bits. Since each literal HRL word contains 31 useful bits, this causes bitwise logical operations between a fill word and a literal word to leave off an arbitrary number of excess bits. This will lead to the same problem that slows down the HRL scheme. To avoid this problem, all fill words need to have multiple of 31 bits. We call the HRL scheme with this extra requirement the *word-aligned hybrid run-length code*, or WAH for short.

Assuming a bit sequence is feed into a WAH bit vector one bit at a time, the incoming bits will be accumulated in the active word initially. This active word will alway contain less than 31 useful bits. Once the active word is full, i.e., having 31 bits, we store it as a regular literal word unless these bits are part of a long fill. In order to be part of a long fill, all the bits in the active word must be the same and same as the preceding fill. This fill can appear either a fill word or a literal word. When it is a literal word, all the bits in the literal word must be the same and the same as the bits in the active word. The process of appending a full active word is captured in the following pseudocode segment. The WAH bit vector will also have a STL vector container and an active word, similar to a HRL bit vector.

ALGORITHM **5** Append a full active word to *vec* of a WAH bit vector. (Operators $\ll$ and $\gg$ indicates bitwise left and right shift.)

```
if vec is empty,
  then vec.push_back(active.value);
elsif active.value = 0,
     then
          if vec.back() = 0,
            then vec.back() = (1 ≪ 31) + 62;
          elsif vec.back() ≫ 30 = 2   &&   ((vec.back() ≪ 2) ≫ 2) < 2^30 − 31,
               then vec.back()+ = 31;
                else vec.push_back(active.value).
          fi
elsif all 31 bits of active.value are one,
     then
          if vec.back() = active.value,
            then vec.back() = (3 ≪ 30) + 62;
          elsif vec.back() ≫ 30 = 3   &&   ((vec.back() ≪ 2) ≫ 2) < 2^30 − 31,
               then vec.back()+ = 31;
                else vec.push_back(active.value).
          fi
      else vec.push_back(active.value).
fi
```

This scheme may use more computer words than HRL to represent the same bit sequence. In particular, every fill word in this scheme is likely to be smaller than the corresponding fill word in the previous one, the bits not represented in the fill words will appear as parts of literal words. If a bit sequence is represented by only one fill word in the HRL code (plus a few bits in the active word), it is likely that the HRL fill word will be represented as a shorter WAH fill plus a literal word in a WAH bit vector. In this special case, the WAH bit vector has twice as many code words as the HRL bit vector. However this extreme case hardly ever occurs in practice. In many cases, a HRL bit vector contains alternating fill words and literal words. In a typical case, when HRL fill word is turned into a WAH fill word, a few bits is pushed into the ensuing literal word and the literal word needs to pass the same number of bits to the next word. Often, the extra bits pushed out of a literal word can be integrated into the next fill. This makes it possible for a WAH fill to be slightly longer than a HRL fill. In our tests, the average fill length of WAH bit vector is about the same as the HRL bit vector. We typically see the WAH scheme uses less than one percent more space than the HRL scheme.

Figure 13 shows a WAH bit vector. Compared with HRL, the fill run stored in the third word only represent 62 bits rather than 66 bits. The four bits not represented in the fill run are pushed to the next literal run and there are five bits in the active word, see the last two words. In this particular example, WAH and HRL use the same amount of space.

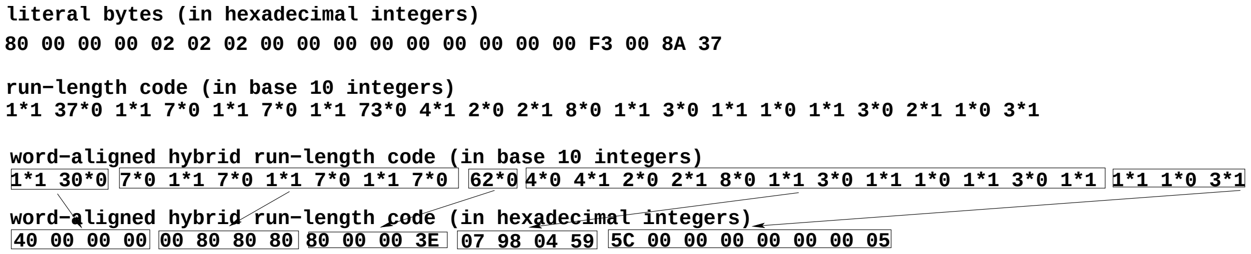


Figure 13: A WAH bit vector.

ALGORITHM **6** Append a fill to the vector container *vec* of a WAH bit vector (Assuming $active.nbits = 0$). Let $fillBit$ denote the fill bit and $nBits$ denote the number of bits in the incoming fill.

if *vec* is not empty,
  then
    if $fillBit = 0$,
      then
        if $vec.back() = 0$,
          then $nBits += 31$; $vec.pop\_back()$;
        elsif $(vec.back() \gg 30) = 2$,
          then $nBits += ((vec.back() \ll 2) \gg 2)$; $vec.pop\_back()$;
        fi
      else
        if $vec.back() = 2^{31} - 1$,
          then $nBits += 31$; $vec.pop\_back()$;
        elsif $(vec.back() \gg 30) = 3$,
          then $nBits += ((vec.back() \ll 2) \gg 2)$; $vec.pop\_back()$;
        fi
    fi
fi
while $nBits > (2^{30}/31) * 31$ **do**
    $vec.push\_back(((2 + fillBit) \ll 30) + (2^{30}/31) * 31)$;
    $nBits -= (2^{30}/31) * 31$;
od
$vec.push\_back(((2 + fillBit) \ll 30) + nBits)$;
$nBits = 0$;

The same process shown in Algorithm 3 can be used to perform bitwise logical operations on WAH bit vectors. One difference is that with WAH, literal words are processed one at a time. This makes the number of iterations of the *while* loop in Algorithm 3 exactly equals to the number of code words generated as the result of the logical operations. In each iteration of the algorithm, either a literal word or a fill word is generated and needs to be appended to the vector container *vec*. The process of appending a literal word and a fill word are described in Algorithms 5 and 6. When executing the two algorithms, the main cost is in executing the nested *if*-tests. Since the two have similar *if*-tests, they should take about the same amount of time. To simplify the discussion on the complexity of the logical operations, we assume that it takes the same amount of time to append a literal word or a fill word to a list of regular code words in *vec*. It is easy to see that this assumption is applicable to most other compressed bit vector schemes except HRL and gzip.

The total time spent in a bitwise logical operation should be proportional to be the number of iterations it goes through the *while* loop starting at the line 5 of Algorithm 3. Let M be the maximum number of words it takes to represent any bit sequence containing the same number of bits as $x.vec$, let N_x be the actual number of code words in $x.vec$ and N_y be the actual number of code words in $y.vec$. Since each iteration of Algorithm 3 at least consumes one code word from either x or y. The number of iterations is in the range of

$$[\max(N_x, N_y), \min(M, N_x + N_y - 1)].$$

When $x.vec$ and $y.vec$ are known, it is easy to determine the actual number of iterations. Let's define $n_x(i)$ to be length of the bit sequence represented by $x.vec[0] \ldots x.vec[i]$ and define $n_y(i)$ similarly. Compare all $n_x(i)$ against all $n_y(j)$. Every time there is a match, Algorithm 3 consumes two code words, one word from each operand. If there are m matches, the total number of iterations is

$$N_x + N_y - m.$$

Proposition 1 *Given that the test bit sequences are generated randomly, if $N_x + N_y$ is less than M, the number of matches m would be much smaller than $N_x + N_y$. Therefore, the total number of iterations is nearly $N_x + N_y$.*

The above arguments show that a logical operation needs to invoke Algorithms 5 and 6 about $N_x + N_y$ times. This is expected to be the most expensive part of the operation. The next time consuming part is to decode the code words of x and y. (The function should be similar to the *decode* function shown on line 30 of Algorithm 3.) Overall, if x and y represent reasonably sparse bit sequences, we expect the logical operation time to be proportional the total number of words in the two bit vectors. As the bit sequences become less compressible, the logical operation time should approach a maximum value.

4.3 Pack Word Code (PWC)

This scheme tries to mimic the behavior of PackBits scheme. It organizes bits first into words and then groups words into two types of runs, a fill run that contains either a 0-fill or a 1-fill, and a literal run that may contain arbitrary number of literal words. A run is represented by at least one word called the header. A fill run only need this header word; a literal run has some literal words following the header. The MSB of the header is zero for a literal run and one for a fill run. The second most significant bit of the header for a fill run is the fill bit and the remaining bits is the number of words in the fill run. We count the number of words instead of number of bits to enforce word-alignment. Assuming the number of bits is representable by one single word, this scheme can represent all possible fill sizes. In a literal run header, its value is the number of words in the literal run. It is valid to store a fill literally. However, if a fill has more than one word long, storing it as a fill run reduces the size of the compressed bit vector.

Figure 14 shows a PWC bit vector. In this example, the first two words are represented as a literal run which take up three words. The second group of two words in the literal version is

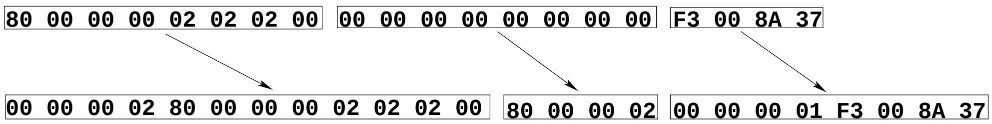


Figure 14: A PWC bit vector.

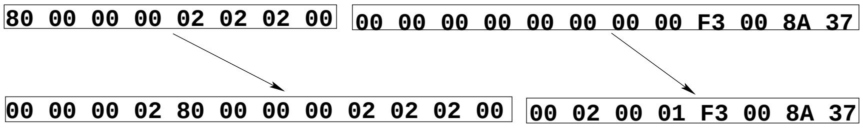


Figure 15: A WBC bit vector.

represented as a single fill run which is stored in one word. Because there are two literal runs which take up two header words, this PWC bit vector actually takes more space than the literal version.

If the entire bit sequence is one single fill, this scheme can represent it in one word. If every word has to be represented literally, the overhead is only a single header word, i.e., this scheme uses one more word than the literal scheme. In most cases, this scheme requires no more space than the literal scheme.

4.4 Word-aligned bitmap code (WBC)

Each time a literal run is started in the PWC scheme, a header word is needed. Since this header word does not directly represent any bit values, it is an overhead that we would like to reduce. In the BBC scheme, the header byte is used to represent both the fill length and the tail length. Following this strategy, we develop an encoding scheme with a similar header format. To simplify the encoding scheme, all runs in this scheme will consist of a fill followed by some literal words which will also be called a *tail*. A header word needs to be split into three part, a fill bit, a counter to represent the fill length and a counter to represent the tail length. As in the preceding scheme, we will count the fill length and the tail length as the number of words respectively. The fill bit takes up one bit, the other two share the rest of the word. One design decision is how many bits to give to each counter. If more bits are used to represent the fill length, the minimum compression ratio would be smaller. However, a long sequence of literal words would have to be represented as more runs and wasting more memory on header words. In our implementation of this scheme, we give two integers about the same number of bits.

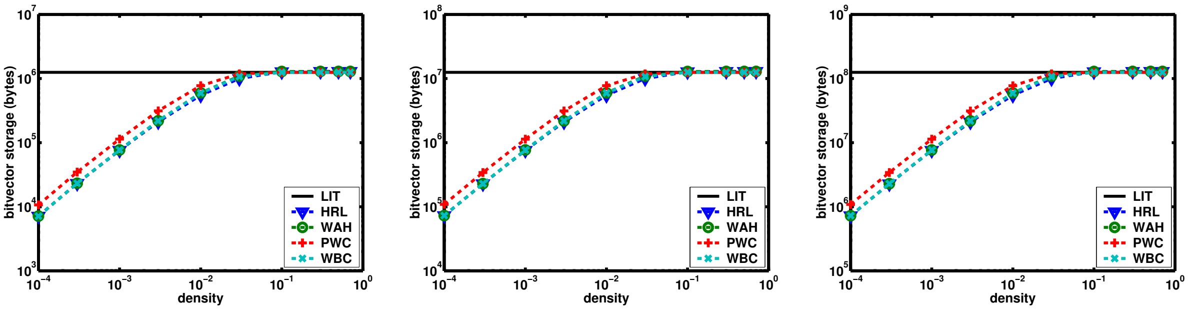
The MSB of the header word is the fill bit, the remaining bits of the upper half is the fill length. When this counter reaches its maximum value, another run has to be started. The lower half of the header contains an unsigned integer that represents the tail length. If the literal tail is not empty, the literal words are stored immediately following the header word. If a word contains four bytes, the maximum number of words in a single fill is 32,767. The minimum compression ratio of this scheme is limited to 1/32,767. In the case where there are only literal words, one header word is needed for every 65535 literal words. As in the previous scheme, we prefer to represent an isolated single word fill as a literal word. This does not impose any memory overhead, but it does require more checking when compressing a bit sequence. The main benefit of doing so is that we will have less header words in a bit vector, which may speed up the logical operations. In addition, it also makes sure that the maximum bit vector size is achieved when the bit sequence is incompressible, i.e., every word is a literal word. When an incompressible bit sequence contains n bits, assuming each word contains 4 bytes (32 bits), it will take $\lceil n/32 \rceil + \lceil n/32/65535 \rceil$ words, where the operator $\lceil\rceil$ computes the ceiling of a number. If there are any fill that is more than one word long, this WBC bit vector will use less storage.

Figure 15 shows the WBC encoding of the same bit sequence used in previous examples. Compared to the PWC scheme, this one represents the last three words of the literal version in one single run which contains one header word and one literal word. The first two words forms a literal run that takes up three words. This WBC bit vector uses the same amount of memory as the literal version but one word less than the PWC scheme.

The following pseudocode segment describes how a full active word is added to a WBC bit vector. The same process can be used to add any literal word. The process of adding a fill should be fairly close this one as well. Because the header words is split to serve multiple functions, this algorithm is more complex than Algorithm 5. Clearly it is still much simpler than the BBC one, see Algorithm 8 on page 38.

ALGORITHM **7** To append a full active word to *vec* of a WBC bit vector.

```
if vec is empty,
  then vec.push_back(1); vec.push_back(active.val); lastHeader = 0;
elsif active.value = 0,
     then
          if vec.size() = lastHeader + 1   &&   vec[lastHeader] < 2^31
                // the last run has only a 0-fill but no tail
            then if (vec[lastHeader] >> 16) < 32767,
                    then vec.back() += 2^16;
                     else ++vec.back(); vec.push_back(active.value);
                 fi
          elsif vec.back() = 0,     // the last tail word is zero
               then --vec[lastHeader];
                    if vec[lastHeader] > 0,
                      then      // change last code word to be a header
                           vec.back() = (2 << 16); lastHeader = vec.size() - 1;
                      else      // resue the last header word
                           vec[lastHeader] = (2 << 16); vec.pop_back();
                    fi
          elsif the tail length of the last run is less than 65535,
               then ++vec[lastHeader]; vec.push_back(active.value).
                else lastHeader = vec.size(); vec.push_back(1); vec.push_back(active.value);
          fi
elsif active.value = 2^32 - 1,
     then
          if vec.size() = lastHeader + 1   &&   vec[lastHeader] > 2^31,
            then if (vec[lastHeader] >> 16) < 65535,
```



(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 16: Sizes of the word-based compressed bit vectors.

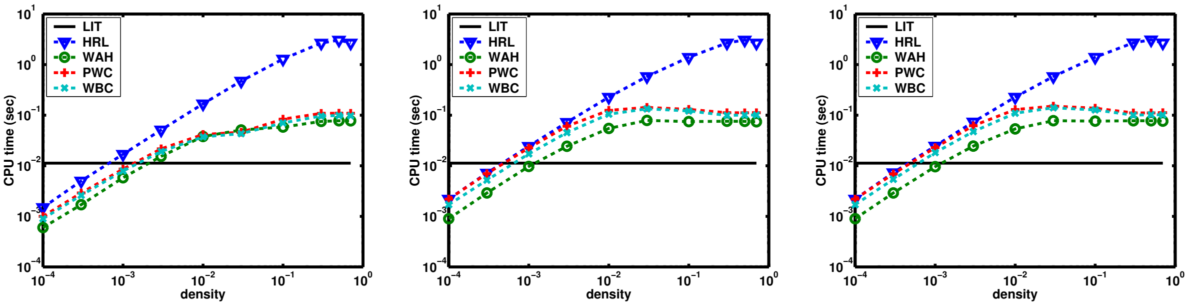
```
                    then vec.back()+ = 2^16;
                     else ++vec.back(); vec.push_back(active.value);
                  fi
          elsif vec.back() = active.value,
               then −− vec[lastHeader];
                    if vec[lastHeader] > 0,
                      then vec.back() = ((1 ≪ 15) + 2) ≪ 16; lastHeader = vec.size() − 1;
                       else vec[lastHeader] = ((1 ≪ 15) + 2) ≪ 16; vec.pop_back();
                    fi
          elsif the tail length of the last run is less than 65535,
               then ++vec[lastHeader]; vec.push_back(active.value).
                else lastHeader = vec.size(); vec.push_back(1); vec.push_back(active.value);
          fi
elsif the tail length of the last run is less than 65535,
     then ++vec[lastHeader]; vec.push_back(active.value).
      else lastHeader = vec.size(); vec.push_back(1); vec.push_back(active.value);
fi

class WBC {
     std::vector<unsigned> vec;
     unsigned              lastHeader;     // the index to the last header word in vec
     activeWord            active;
};
```

If we count the header word toward the representation of the fill, then the similar argument used to estimate the complexity of the logical operations on WAH bit vectors can also be used here. On sparse bit vectors, we also expect the time needed to perform a bitwise logical operation to be proportional the total number of code words in the two operands of the logical operation.

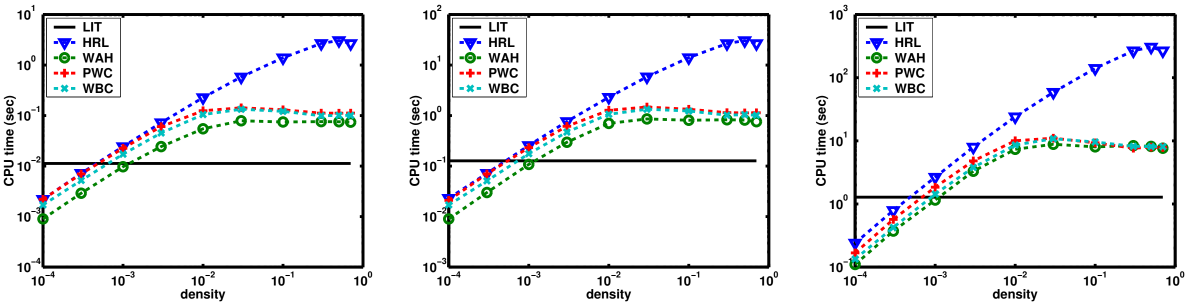
4.5 Performance

As before, we again test our bit vectors on the same set of random bit sequences. The first set of results reported in Figure 16 is about the sizes of the four compressed bit vectors introduced in this section. In general as bit density increases, the bit sequences are less compressible and it



(a) AND(&) (b) OR (|) (c) XOR (∧)

Figure 17: CPU time needed to perform a bitwise logical operation between two bit vectors containing 10 million bits.



(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 18: CPU time needed to perform a bitwise OR operation between two bit vectors of various sizes.

takes more computer memory to represent them. Among the four schemes, HRL, WAH and WBC have very similar compression ratios; PWC usually uses more space than the other three. When bit density is 10^{-4}, the compression ratios are: HRL 0.0055, WAH 0.0055, PWC 0.0083, and WBC 0.0055. These compression ratios are considerably larger than the byte based versions reported in the previous section, which confirms what is clearly demonstrated in the examples shown in Figures 6 – 8 and 12 – 15. Since the set bits are uniformly distributed, it is much more likely for a byte to contain only bits that are zero. On the test machine, each word is four bytes. When the fills are short, it is much more efficient to store the fill length in a byte than in a word. If a word contains eight bytes, we would see the compression ratios to be even worse. The advantage of using word based scheme is that we might have faster bitwise logical operations.

Figures 17 and 18 and Table 4 show some timing results of performing bitwise logical operations on the random bit sequences. Overall, HRL takes the longest time to perform the same operation, and WAH usually uses the least amount of time. When bit densities are low, all four versions of word based compressed bit vector use less time than the literal version. At bit density around 10^{-3}, three out of the four schemes use more time than the literal scheme. From Table 4 we see that in

HRL

density	10^{-4}			10^{-3}			0.5		
No. bits	10^7	10^8	10^9	10^7	10^8	10^9	10^7	10^8	10^9
AND	0.002	0.014	0.156	0.017	0.170	1.748	3.057	30.36	305.3
OR	0.002	0.023	0.240	0.024	0.256	2.654	3.062	30.54	305.6
XOR	0.002	0.022	0.232	0.025	0.255	2.544	3.093	30.88	308.7

WAH

density	10^{-4}			10^{-3}			0.5		
No. bits	10^7	10^8	10^9	10^7	10^8	10^9	10^7	10^8	10^9
AND	0.001	0.005	0.054	0.006	0.059	0.622	0.078	0.834	8.732
OR	0.001	0.009	0.110	0.010	0.108	1.154	0.076	0.816	8.158
XOR	0.001	0.009	0.098	0.010	0.106	1.040	0.078	0.837	8.222

PWC

density	10^{-4}			10^{-3}			0.5		
No. bits	10^7	10^8	10^9	10^7	10^8	10^9	10^7	10^8	10^9
AND	0.001	0.008	0.084	0.009	0.084	0.872	0.109	1.138	8.102
OR	0.002	0.021	0.168	0.023	0.232	1.844	0.112	1.131	7.948
XOR	0.002	0.021	0.154	0.023	0.229	1.680	0.111	1.148	7.902

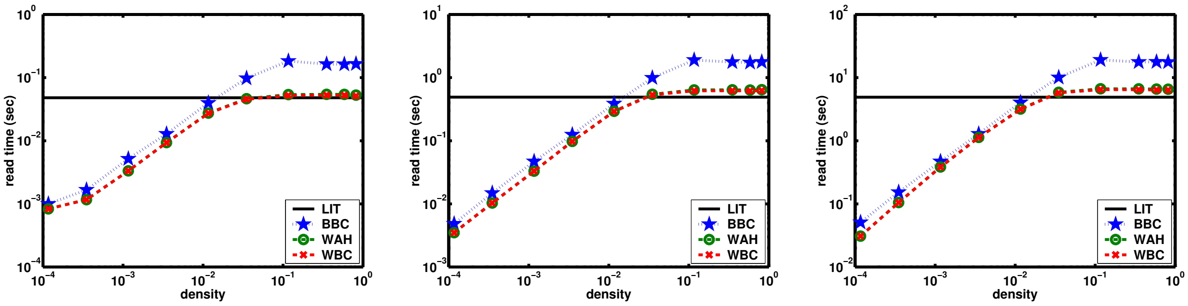
WBC

density	10^{-4}			10^{-3}			0.5		
No. bits	10^7	10^8	10^9	10^7	10^8	10^9	10^7	10^8	10^9
AND	0.001	0.009	0.072	0.008	0.076	0.744	0.099	1.028	8.214
OR	0.002	0.017	0.134	0.017	0.176	1.434	0.101	1.042	8.100
XOR	0.002	0.017	0.124	0.018	0.178	1.356	0.102	1.038	7.894

Table 4: Average CPU time (seconds) to perform one bitwise logical operation between two word-based bit vectors.

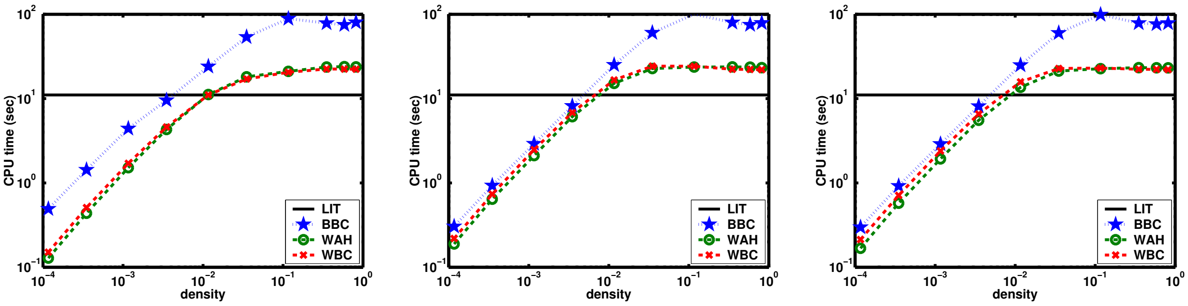
many cases WAH needs only 50% – 70% of the time used by PWC and WBC. When bit densities are high, logical operations on two HRL bit vectors take almost 40 times longer than performing the same operations using any one of the three word-aligned schemes. This clearly demonstrates the importance of maintaining word-alignment in the word based scheme. When the bit sequences are not compressible, these word-aligned bit vectors takes as much as eight times longer than the literal version.

Compared to the byte based schemes, see Tables 3 and 4, at bit density of 10^{-4}, operating with PWC and WBC bit vectors uses about the same amount of time as on BBC bit vectors even though the word based schemes need to go through more memory than the byte based versions. Logical operations on two WAH bit vectors needs about half the time of that on two BBC bit vectors. As the bit density increases, the time used by the word based schemes grow much slower than that of the byte based schemes. When the bit density is close to 0.5, where the word based schemes and the byte based schemes uses about the same amount of space, BBC takes about 34 seconds to perform one bitwise logical operation on one billion bits, see Table 3, but WBC takes about 8 seconds for the same operation, see Table 4. In many cases, the byte-aligned versions take about four times as long to perform a logical operation than the word-aligned versions, see Tables 3 and



(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 19: Average time (seconds) to read one bit vector from file.



(a) AND(&) (b) OR (|) (c) XOR (∧)

Figure 20: CPU time needed to read two bit vectors containing 1 billion bits and perform a bitwise logical operation.

4.

Figures 19 and 20 show how the IO affects the bit vector performances. Figure 19 shows the time needed to read one bit vector from file. Among the three word-aligned bit vectors, in terms of storage requirement, PWC does not compress as well as WBC or WAH; in terms of bitwise logical operation speed, PWC is not any faster than the other two. For these reasons, we have left out PWC in this set of tests. The time reported in Figure 19 including reading the file content and forming the bit vector data structure in memory. Since the sizes of WAH and WBC are fairly close, the time needed to read them are very close as well. The average reading speed for WAH and WBC bit vectors is about 23 MB/s. The raw reading speed, i.e., only read the file content but not forming the bit vector, is about 60 MB/s. From this, we see that almost two thirds of the total reading time is actually spent on forming the bit vector. If we can avoid forming the bit vector during bitwise logical operation, we can reduce the total execution time. Reading and forming BBC bit vectors is slower than performing the same operation on the two word-aligned bit vectors. The average reading speed is about 6.5 MB/s, which is about a quarter of the speed of reading WBC bit vectors. This is largely because the reading program has to insert each byte into the STL vector container one byte at a time. The reading program for WBC can perform the same operation one word at a time. Since moving one byte takes about the same amount of time as moving one word, it is clear that insert the same number of bytes one byte at a time takes four times as long.

Figure 20 shows the total time of reading two files, forming two bit vectors, performing a bitwise logical operation and storing the result in a new bit vector. In most cases, more than half of the time is spent in the first two steps. Overall, working with WAH bit vectors takes the least amount of time. Operating on BBC bit vectors takes about 60 – 300 % more time than operating on WAH bit vectors. The difference between time used by BBC scheme and WAH scheme increases as bit densities increases. Comparing between WBC and WAH, when bit density is low, WAH is slightly better – the time difference is less than 25%; when bit density is high, the two scheme use about the same amount of time.

5 Observations and Analyses

We have tested our implementations of a number of compressed bit vectors on a simple set of test data and have made some observations. In this section we will analyze the observations to see whether they can be generalized.

5.1 Alignment is good for logical operations

In subsections 4.1 and 4.2, we have given reasons for adopting a word-aligned bit vector. Here we merely recapitulate the main points. Nonaligned versions usually are slightly more compact, but the aligned versions offer much faster logical operations. In this paper, we have only one set of such examples, HRL vs. WAH. However other examples are easy to find. For instance, in a previous study, Johnson showed that a scheme called ExpGol [17] achieves better compression than BBC but bitwise logical operations on ExpGol bit vectors usually take longer [11]. Part of the reason for this performance difference is that ExpGol has no alignment requirement. Since ExpGol is quite different from BBC, the relative performance is not due to alignment alone. The only difference between HRL and WAH is the alignment, the performance difference should be due to this alignment difference. In our tests, WAH bit vectors are not more than 1% bigger than the corresponding HRL bit vectors, but logical operations on WAH bit vectors are 2 – 40 times faster than that on HRL bit vectors, see Table 4 and Figures 16 – 18.

5.2 Word-aligned bit vectors are faster

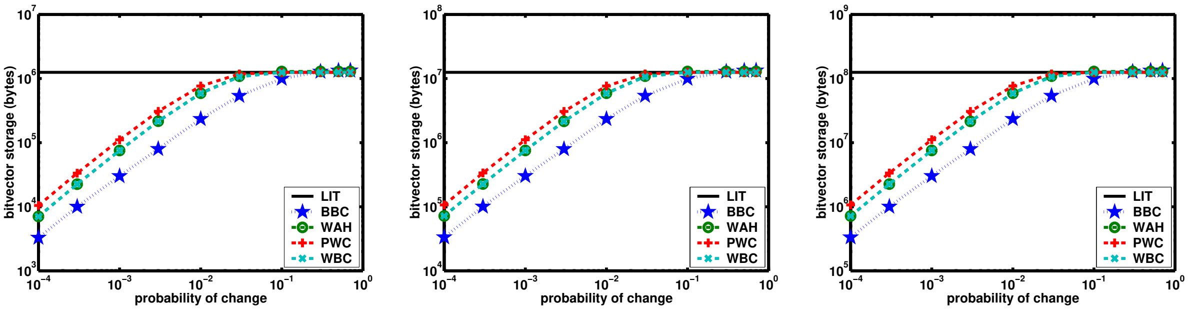
Word-aligned bit vectors do not compress as well as byte-aligned bit vectors, but logical operations on the word-aligned bit vectors are faster. The compression difference is easy to see. For example, in BBC scheme, three consecutive bytes of zero can be represented in one byte, but a word based version will represent it as part of a literal word. When bit density is 10^{-4}, the WBC bit vector representing the same bit sequence takes almost three times the space than the BBC bit vector. On the random test data set, when bit density is 10^{-4}, the most common BBC run is a type 4 run representing around 10,000 bits using three bytes. The same 10,000 bits in WBC encoding is represented using two words (8 bytes). The difference of memory usage is close to a factor of three. Though the differences are large when bit density is low, we also see that the differences are much smaller when the bit density is higher. In fact, for incompressible bit sequences, WBC bit vectors are slightly smaller than BBC bit vectors. If a bitmap index has a mixture of high and low bit density sequences, most of the space is taken up by the high density ones. The differences in the amount of storage used for the very sparse ones don't significantly change the total storage requirement. For the set of 20 test bit sequences of the same length, the total storage used by the WBC bit vectors is about 25% more than that of the BBC version. Though 25% is a noticeable difference, but this is not nearly as bad as a difference of a factor of three. In addition, the small size of BBC bit vectors may not translate to faster logical operations even when we also include the time read bit vectors from disk, see Figure 20.

The performance difference between the word-aligned bit vectors and the byte-aligned bit vectors can be partly attributed to a design characteristic of modern computer architecture. In most of the current general-purpose CPUs, it takes about the same effort to retrieve a byte as to retrieve a word, and it takes about the same effort to perform a logical operation on two bytes than to perform the same operation on two words. When a word is four bytes long, logical operations on two word-aligned bit vectors could be four times faster than the same operation on two byte-aligned bit vectors. For example, it takes about 8 seconds to operate on two WBC bit vectors with bit density of 0.5, and it takes about 34 seconds to do the same on two BBC bit vectors, see Tables 3 and 4. This explains the performance differences when the bit sequences are incompressible. To explain the performance differences when the bit sequences are compressible, we need to look at the logical operation algorithm.

As explained in Section 4.2, the most important quantities that determines the complexity of a logical operation are the complexity of the compression and the decompression algorithm and the number of times they are invoked. The decompression algorithms for most schemes discussed are relatively simple, see the *decode* function on line 31 of Algorithm 3. The important parts of the compression algorithm are the process of appending a literal word/byte and a fill to a list of regular code words/bytes. Compared against the word-aligned schemes, the byte-aligned schemes either have more complex append functions or they need more iterations during the logical operations. Take the example of BBC vs. WBC. It easy to see that the algorithm of appending a literal word to a BBC bit vector (Algorithm 8) is much more complex than that of the WBC scheme (Algorithm 7). Because both algorithms consist of a large number of complex *if* tests, we would expect the one with more layers of nested *if* tests, that is, Algorithm 8, to take longer. This shows that each iteration of the *while* loop will take longer for BBC bit vectors. When the test bit sequences are very sparse, there are about the same number of runs in both BBC and WBC bit vectors. However, if the bit sequences are not so sparse, the BBC scheme divides a bit sequence into more runs than the WBC scheme because each BBC run can only have 16 bytes in its tail and it starts a new run for every short fill. The effect of this is especially strong when the bit sequences are hard to compress.

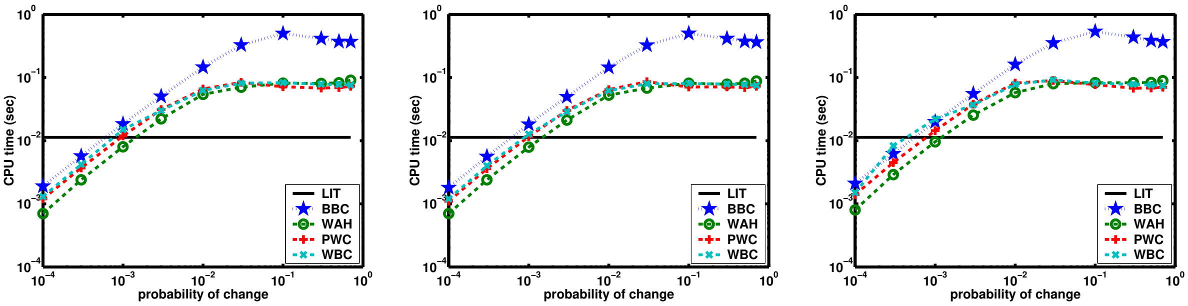
5.3 Sizes of operands determine speed

In all our tests, the bit vector sizes appear to be strongly related to the bit density. However, this is a superficial connection. In section 4.2 (page 21), we shown that the logical operation time is actually proportional to the number of code words in the operands, i.e., the space taken by the two



(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 21: Sizes of the compressed bit vectors containing bit sequence generated by a simple Markov process.

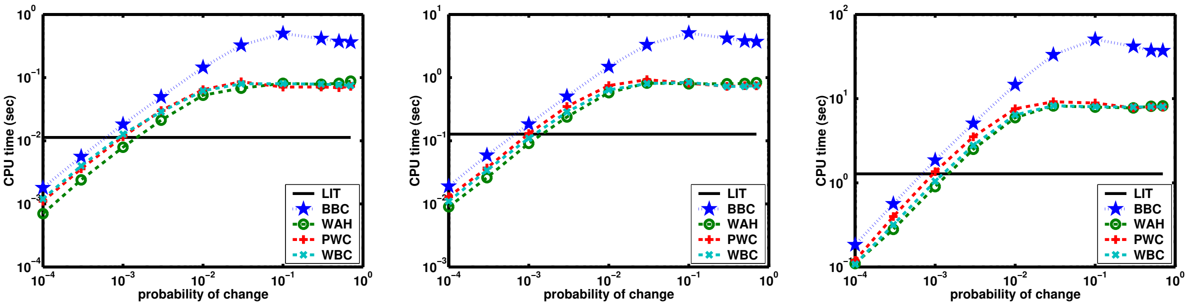


(a) AND(&) (b) OR (|) (c) XOR (∧)

Figure 22: CPU time needed to perform a bitwise logical operation between two bit vectors containing 10 million bits generated by a simple Markov process.

operands of a logical operation. The apparent relation between bit density and logical operation time is only true because the bit density is directly related to the compressibility in the random data set, see Figures 9 and 16. Since all bit vectors schemes compress consecutive identical bits, i.e., fills, what makes a bit sequence compressible is that it contains a large number of long fills. When the bit density is low, it is more likely there are long 0-fills and the bit sequences are easier to compress. To demonstrate this point, we repeat some of our tests on a different set of test data. This time, we use a simple Markov process to generate bit sequences and assign the same transition probability for a bit to change from zero to one and from one to zero. This simple Markov process let us control the size of the fills. When the transition probability is low, the fills will be long, and when the transition probability is high, the fills will be short. The bit sequences generated in this case is expected to have bit density 0.5 unless the transition probability is exceedingly low, say, less than or close to one over the total number of bits. To distinguish from the previously used random data set, we call this one the *Markov data set*.

Figures 21 – 23 show some performance information of four different bit vectors on this new set of bit vectors. The horizontal axes are probability of change, i.e., the transition probability. The four



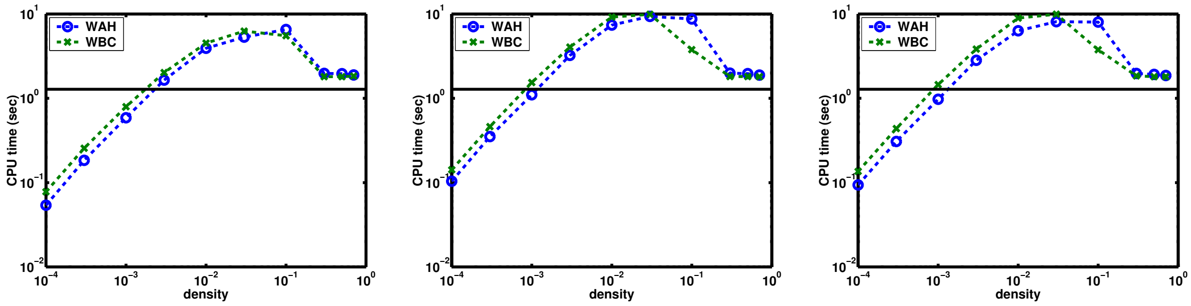
(a) 10 million bits (b) 100 million bits (c) 1 billion bits

Figure 23: CPU time needed to perform a bitwise OR operation between two bit vectors generated by a simple Markov process.

schemes shown are BBC, WAH, PWC and WBC. Figure 21 shows the size of the compressed bit vectors. At lower transition probability, the bit sequences are more compressible. When transition probability is 10^{-4}, the compression ratios are: BBC 0.0026, PWC 0.0082, WBC 0.0055, and WAH 0.0055. This compression ratios are very close to those observed when bit density is 10^{-4} in the random data set, see Figure 9 and 16, because the average runs in two data set contains about the same number of bits and the same size fills. The bit vectors storing the Markov data set is slight larger, because random data set contains mostly isolated set (one) bits, most runs are of type 4 and type 2 where the tail bytes are not stored. In contrast, most tail bytes have to be stored explicitly in the Markov data set. When bit density is 10^{-4} in the random data set, the most common BBC run is a type 4 run represented by three bytes. When transition probability is 10^{-4} in Markov data set, the most common BBC run is a type 3 run represented by four bytes, one byte header, two bytes representing the fill length, and one byte represent the literal tail. Therefore, BBC bit vectors for Markov sequences are about 30% bigger than the BBC bit vectors for random data set. As the transition probability increases, the bit sequences become less compressible and the storage requirements for all versions of the bit vectors increase.

Figures 22 and 23 show the average CPU time required to perform a single bitwise logical operation. This set of plots confirms that as the size of the bit vector increases, the logical operation time increases as well. Comparing the four methods, logical operations on BBC bit vectors take the most time, and logical operations on WAH takes the least time in most cases. When the bit sequences are highly compressible, operating on WAH bit vectors takes about half as much time as operation on BBC bit vectors; when the bit sequences are not compressible, operating on the three word-aligned bit vectors are about four times faster. Though the difference among the three word-aligned versions are smaller on the Markov data than on the random data, see Figures 17 and 18, WAH is still the clear winner in most cases.

On the two test data sets, when the compression ratio is less than 0.05, the logical operations on the compressed bit vectors are faster than on the uncompressed ones.



(a) AND(&) (b) OR (|) (c) XOR (∧)

Figure 24: CPU time needed to perform a bitwise logical operation between two bit vectors with one billion bits each.

5.4 Recognizing the incompressible is useful

In our tests, the logical operations are the fastest on WAH bit vectors. However it is faster than the literal version in only three out of the 10 test cases. A natural question is whether it is possible for a compressed bit vector to always be faster than the literal version or at least not slower than the literal version. When the bit sequences are not compressible, logical operations on the word-aligned bit vectors take up to eight times longer than the same operations on the literal bit vectors. In these cases, the actual logical operations are the same in both the compressed and uncompressed versions, the extra time used by the compressed versions is in decoding the compressed code words and compressing the result of logical operation. If we accept the fact that these bit sequences are not compressible, to make the compressed versions competitive against the uncompressed version, it is necessary to remove these compression and decompression cost. To limit the amount of code modifications, we concentrate on two word-aligned versions of bit vectors, WAH and WBC.

In a WAH bit vector, each word need to be checked to determine whether it is a fill word or a literal word. This check translates into a comparison and a branch instruction at the hardware level. Since the branch condition depends on the comparison result, the branch operation has to wait for the result of comparison to be available. In a pipelined CPU like the UltraSPARC II used in our tests, this wait can waste a number of clock cycles. It is also known that branch operations effectively takes longer than others such as bitwise logical operations [23]. Together this seemingly simple check takes much longer than the actual time spent in a bitwise logical operation between two words. If we know for sure that there is no fill word in a bit vector, we can avoid performing this check and speed up the overall operation. Indeed if we skip this check and also skip checking whether the resulting bit vector is compressible, the logical operation speed is comparable with that of the literal version. For example, with these modifications the average time to perform a bitwise logical operation on two 100-million-bit incompressible sequences is roughly 0.145 seconds which is close to 0.127 seconds for the literal version.

In a WBC bit vector, it is also very easy to identify when a bit vector contains only literal runs. When this case is identified, the bitwise logical operations can be carried out without further checking the header words. If we also forgo the checking of whether the resulting bit sequence is compressible, the bitwise logical operations can be as fast as the literal version. After these modifications, the average time need to perform a bitwise logical operation on two incompressible WBC bit vectors of 100 million bits is 0.142 seconds which is close to 0.127 seconds for the literal version.

Figure 24 shows the CPU time of logical operations versus the bit density of the random test data. In this case, we have used the special functions to perform the bitwise logical operations when the bit sequences are incompressible. The lines for WAH and WBC should be the same as in Figure 18 except when the bit sequences are incompressible. When bit densities are close to 0.5, the bit sequences are incompressible and the compression ratios are slightly above 1. In Figure 24, we see that the time used to perform logical operations on these bit sequences are very close to that of the literal case.

After implementing this modification, the WAH and WBC bit vector implementations can efficiently handle highly compressible bit sequences and incompressible bit sequences. For the moderately compressible bit sequences where the compression ratio is large than 0.1, the speed of logical operations on WAH and WBC bit vectors can still be considerably slower than that of the literal bit vectors. On both test data sets, it may take up to eight times as long to perform a bitwise logical operation on the compressed bit vectors as on uncompressed ones.

5.5 Separated decompression is slow

One option not considered so far is to decompress the compressed bit vectors before performing the logical operations, an option that is extensively examined in a previous study [11]. The main reason that we have not considered it is that it requires the same amount of computer memory as the literal scheme even though the file storing the bitmap indices can be smaller than the literal scheme. Since after decompression the bit sequences are stored as literal bit vectors, the logical operations can always operate as fast as the literal scheme. Figure 25 compares the time it takes to perform a bitwise OR operation directly on two compressed bit vectors versus decompressing the two operands first then performing the logical operation. The line marked with "de-WAH" represents the time it takes to decompress two WAH bit vectors first then perform the logical operation. It is clear that performing decompression first always take more total time than operating on the compressed bit vectors directly.

If the bit sequences are relatively small compared to the available computer memory and the worst case scenario observed in Figures 18 and 24 must be avoided, then it is reasonable to decompress the bit vectors first when the compression ratio is above 0.1 (bit density $> 10^{-3}$ in the random data set). Under these circumstances, the uncompressed bit sequences should be stored in memory as long as possible to avoid the need of repeatedly decompressing them.

5.6 STL vector container is slow for IO

For convenience of testing, we have been using STL vector container to store our bit vectors. Table 1 shows that reading literal bit vectors and storing them in STL vector containers take more than twice the time as simply reading the files. The extra time is spent in copying the content from the temporary buffer to the vector containers. Since the STL vector container does not promise the data to be stored in any particular fashion, the copying has to be done one element at a time.

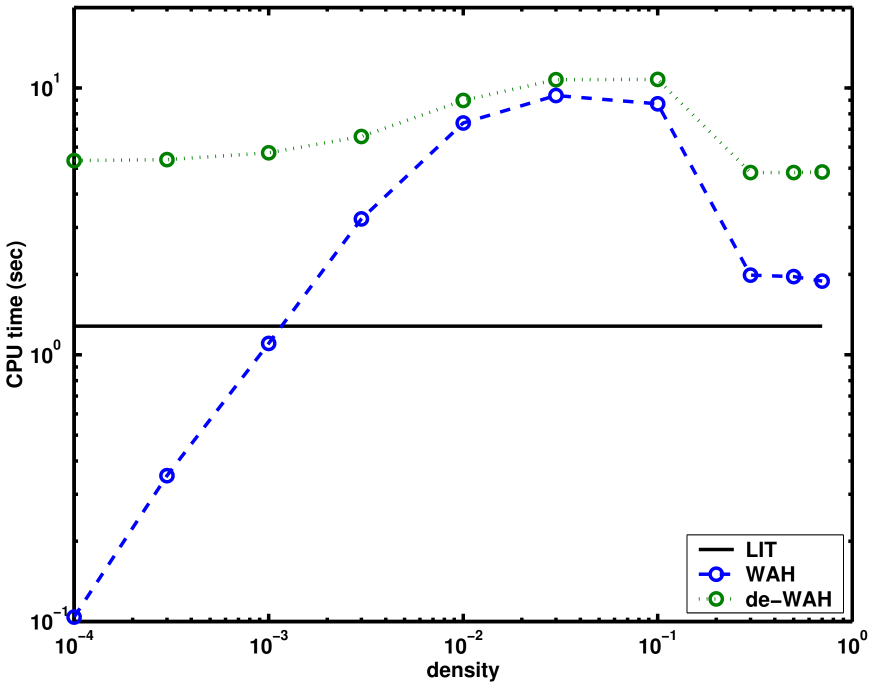


Figure 25: CPU time (seconds) needed to perform bitwise OR operations between two bit vectors of one billion bits each.

It could be much faster if we are able to rely on the fact the all elements of the vector container are stored consecutively and directly transfer the content of the file to the appropriate memory location. To verify this, we implemented our own version of the vector container that avoids the extra copying, and reading time using the new vector container is close to the "read only" time shown in Table 1. It takes 0.0157 seconds to read 10^7 bits, 0.160 seconds to read 10^8 bits and 1.77 seconds to read 10^9 bits. In fact, the read time with this new vector container is slightly better than the "read only" time shown in Table 1. This is because in the new scheme we directly transfer the content of a whole file to consecutive memory locations while in previous scheme the transfer is done one page at a time.

When the appropriate optimization flags (`-O3` or `-fast` with SUN WorkShop Compiler CC) are used, the speed of logical operations using STL vector container and our own implementation are identical. It can be further verified by examining the assembler code generated by SUN's compiler that neither the iterator or the subscripting operator (`[]`) introduces any extra work in the final binary code. Therefore using STL vector container has not affected the speed of the logical operations.

5.7 Putting it together

In this section, we have presented arguments for replacing STL vector container with a custom one and checking the total size of the WAH and WBC bit vectors to avoid decompressing individual

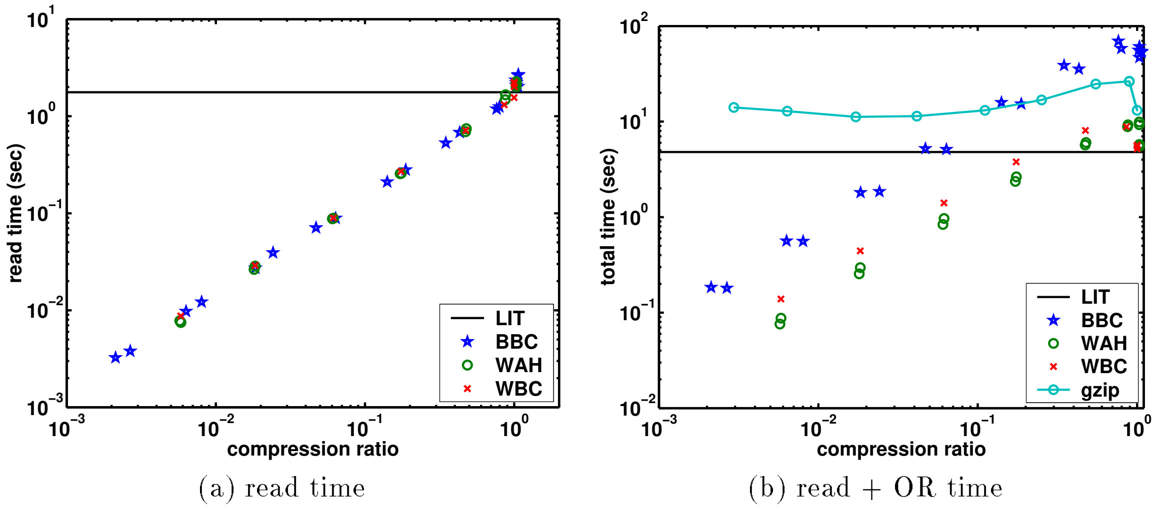


Figure 26: Average CPU seconds needed to read bit vectors from files and perform bitwise OR (10^9 bits, both random and Markov data).

runs. The data shown in Figure 26 shows the performance of the new implementations of bit vectors with these modifications. The logical operation time are not shown separately here because they are the same as in Figure 24. This figure differs from early ones in that the horizontal axes are compression ratio. It also contains the timing results on both the random test data and the Markov test data. The BBC scheme compresses better than WAH and WBC, this is indicated by the dots for BBC scheme appearing to the left of the corresponding dots for WAH and WBC. Since we have removed the overhead of individually copying each byte to the STL vector container, the reading speed for BBC bit vector is now comparable with that for others. However, the logical operations on two BBC bit vectors are still considerably slower than that on two WAH or WBC bit vectors.

Since the most likely scenario for using a bitmap index is to read a number of bit vectors from files and perform logical operations among them. We also show the total time of reading two bit vectors from two files and performing one logical operations on them, see plot (b) of Figure 26. Taking into account of the reading time, the overall performances of WAH and WBC compare quite well against that of LIT. The total time for either WAH and WBC scheme is significantly larger than that of the LIT scheme only in 20% of test cases. Even in the worst cases, the total time used by the WAH scheme is less than twice that of the LIT scheme and the total time used by the WBC scheme is less than 60% longer than that of LIT scheme. We are not able to beat the literal scheme in all cases, but we are close to this goal.

Comparing the word-aligned schemes again the byte-aligned schemes, we see that WAH and WBC are on average 10 times faster than BBC. The total size of the files storing all the test bit sequences in either WAH for WBC form is only about 25% more than the total size of the BBC files. In short WAH or WBC are significantly faster than BBC but only use slightly more space.

6 Summary and future work

We set out to search for an efficient compressed bit vector scheme that can decrease the response time of bitmap index for large high-dimensional database. Such a scheme would perform bitwise logical operations faster than the uncompressed version but use much less space. One of the best known scheme that is close to satisfy this requirement is the Byte-aligned Bitmap Code (BBC). Through this study, we see that some new word-aligned schemes, such as the word-aligned hybrid run-length (WAH) scheme and the word-aligned bitmap code (WBC), can be 10 times faster than BBC while using only 25% more space. Compared against the uncompressed literal scheme, logical operations on the BBC bit vectors are faster only when the operands are extremely sparse. However, WAH and WBC are slower than the uncompressed scheme in only 20% of the test cases. Clearly, the specific numbers is strong affected by the test data selected. However, this performance gain over the byte-aligned scheme is easy to understand.

One basic reason for the performance difference is that the computing hardware has better support for word-based operations than byte-based operations. In addition, because the special kind of bitmap is easy to compress, we are able to use very simple compression algorithms to achieve a good level of compression. A major benefit of using a simpler compression algorithm is that the decoding and encoding processes are much simpler and faster. More specifically, the time needed for a logical operation between two compressed bit vectors are dominated by two factors, the time to decode the fills and literal bytes/words in the two operands and the time to encode the fills and literal bytes/words produced by the logical operation. Decoding each code bytes in BBC is more complex than decoding a code word of either WBC or WAH. Given the same bit sequences, the BBC scheme breaks the sequences into more fills and literal bytes than WAH or WBC. A logical operation on BBC bit vectors has to invoke the decode functions more times. Furthermore, the operation to encode the fills and literal bytes generated by the BBC scheme is also much more time-consuming. All these reasons lead us to believe that WAH and WBC indeed have a significant performance advantage over BBC.

Through the analyses, we also learned that alignment is important for the compressed bit vector schemes. Forgoing alignment may increase compression effectiveness, but the difference might be small. For example, in our tests, a WAH bit vector is typically less than 1% bigger than the corresponding HRL bit vector. However, logical operations on two HRL bit vectors can take as much as 40 times longer than the same operations on two WAH bit vectors. It is usually worthwhile to use a scheme that is either byte aligned or word-aligned. Other lessons learned include,

- Logical operation time is usually proportional to the number of words in the two compressed operands.
- When considering the performance of the bitmap index scheme, it is often assumed that the logical operation time is negligible. Our tests shown that both the IO time and the computation time have to be considered together in order to achieve the optimal result.
- To achieve good IO performance, file content needs to be directly transfered into in-memory objects. Don't copy each data element to a STL container!
- Some conceptually simple operations are actually quite expensive. For example, checking to

see whether a bit is zero or one takes significantly more time than performing a bitwise logical operation between two words. It saves time to do more logical operations than to perform more *if* tests.

In order to use either WAH or WBC to represent a bitmap index, the data structures needs to support some additional functionality, such as the ability to address and modify individual bit, the ability to append and remove a number of bits at a time, the ability to identify the locations of the bits of a particular value, and so on. Furthermore, to enhance the efficiency of the logical operations, it is necessary to recognize more special cases, such as when only one operand is compressed or the compressed bit vector contains only one regular code word. The current implementation of the logical operations also compute the number of bits that are set. Since this information isn't needed in all cases, the process of counting set bits should be moved into a separate function. These engineering issues are important but are straightforward to resolve.

Acknowledgments

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

This work was supported by the Director, Office of Science, Office of Laboratory Policy and Infrastructure Management, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

BBC compression algorithms

The following algorithm shows how our modified version of the two-sided BBC scheme append a full active byte to the vector container *vec*. Refer to Section 3.2 on page 12 for the definition of our version of the coding scheme. The task of identifying run type based on the header byte can be done with suitable bit masks based on the definitions. In an attempt to keep the algorithm readable, we don't show the exact mask values. As usual, the subscripting operator [] and functions *size*(), *back*(), and *push_back*() are part of the API of the STL vector container class.

ALGORITHM **8** Append a full active byte (a literal byte) to *vec*. (Class activeByte is defined in Algorithm 2.)

```
if vec is empty,
  then lastHeader = 0;
       if active.value = 0,
         then vec.push_back(144); // start a type 1 run with a 1 byte 0-fill
       elsif active.value = 255;
           then vec.push_back(208); // start a type 1 run with a 1 byte 1-fill
       elsif active.value has only one set bit,
           then vec.push_back(64+odd bit position);
            else vec.push_back(1); vec.push_back(active.value);
```

```
        fi
elsif active.value = 0, // all bits are zero
    then if vec[lastHeader] indicates a type 1 run with a 0-fill and no tail,
        then if vec[lastHeader] < 176, // fill length is less than 3
            then vec[lastHeader]+ = 16; // increment the fill length by 1
             else // convert to a type 3 run with a 4-byte 0-fill
                  vec[lastHeader] = 32; vec.push_back(0);
           fi
      elsif vec[lastHeader] = 32, // a type 3 run with no tail
           then incrFillLength;
            else // start a new type 1 run
                 lastHeader = vec.size();
                 vec.push_back(144);
      fi
elsif active.value = 255, // all bits are one
    then if vec[lastHeader] indicates type 1 a run with a 1-fill and no tail,
        then if vec[lastHeader] < 240, // length less than 3
            then vec[lastHeader]+ = 16; // increment fill length by 1
             else // convert to a type 3 run with a 4-byte 1-fill
                  vec[lastHeader] = 48; vec.push_back(0);
           fi
      elsif vec[lastHeader] = 48, // a type 3 run with no tail
           then incrFillLength;
            else // start a new type 1 run
                 lastHeader = vec.size(); vec.push_back(208);
      fi
  else // active.value contains a mixture of zeros and ones
      if vec[lastHeader] indicates type 1 or type 3 a run without any tail byte,
        then if active.value contains only one bit different from the fill bit,
            then // turn type 1 run into a type 2 run, or type 3 run into a type 4 run
                  vec[lastHeader] >>= 1; vec[lastHeader]+ =the odd bit position;
             else // add the active byte as a tail byte
                  + + vec[lastHeader]; vec.push_back(active.value);
           fi
      elsif vec[lastHeader] indicates a type 1 or type 3 run with less than 16 tail bytes,
           then // increase tail length by one
                 + + vec[lastHeader]; vec.push_back(active.value);
            else // start a new type 1 or type 2 run with one tail byte
                 lastHeader = vec.size();
                 if active.value has only one set bit,
                   then vec.push_back(64+odd bit position);
                    else vec.push_back(1); vec.push_back(active.value);
                 fi
      fi
fi
funct incrFillLength ≡
  // this function increment the multibyte fill length by one
  if vec.back() < 127,
    then + + vec.back();
     else
         vec.back() = 0; // the seven least significant bits are zero
         i = vec.size() − 2;
         while i > lastHeader, do
```

```
                if vec[i] < 255,
                   then ++vec[i]; return;
                   else vec[i] = 128;  // the seven bits of this segment are zero
                        --i;
                fi
            od
            insert the value 129 behind lastHeader in vec
    fi.

class BBC {
    std::vector<byte> vec;           // code bytes
    unsigned long lastHeader;        // index of the last header byte in vec
    activeByte active;               // the active byte
};
```

References

[1] G. Antoshenkov. Byte-aligned bitmap compression. Technical report, Oracle Corp., 1994. U.S. Patent number 5,363,098.

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

[3] Apple Computer, Inc. *Understanding PackBits*, February 1996. URL `http://devworld.apple.com/technotes/tn/tn1023.html`.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press and Addison Wesley, New York, 1999.

[5] A. Bookstein and S. T. Klein. Models of bitmap generation: a systematic approach to bitmap compression. *Information Processing & Management*, 28:735–748, 1992.

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

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

[8] Y. Choueka, A. S. Fraenkel, S. T. Klein, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. In *Proc. 9'th ACM-SIGIR Conference on Information Retrieval, Pisa, Italy, September 1986. ACM, New York*, pages 88–97, 1986.

[9] K. Furuse, K. Asada, and A. Iizawa. Implementation and performance evaluation of compressed bit-sliced signature files. In Subhash Bhalla, editor, *Information Systems and Data Management, 6th International Conference, CISMOD'95, Bombay, India, November 15-17,*

1995, Proceedings, volume 1006 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1995.

[10] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evalution of signature files as set access facilities in OODBs. In P. Buneman and S. Jajodia, editors, *Proceedings ACM SIGMOD International Conference on Managerment of Data, May 26-28, 1993, Washington, D.C.*, pages 247–256. ACM Press, 1993.

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

[12] D. Lee and C. Leng. A partitioned signature file structure for multiattribute and text retrieval. In *Proceedings of the Sixth International Conference on Data Engineering, Los Angeles (Feb. 1990)*, 1990.

[13] D. L. Lee, Y. M. Kim, and G. Patel. Efficient signature file methods for text retrieval. *IEEE Transactions on Knowledge and Data Engineering*, 7(3), 1995.

[14] Jean loup Gailly and Mark Adler. *zlib 1.1.3 manual*, July 1998. Source code available at `http://www.info-zip.org/pub/infozip/zlib`.

[15] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998. Software available at `http://www.math.keio.ac.jp/~matumoto/-emt.html`.

[16] John Miano. *Compressed Image File Formats: JPEG, PNG, Gif, Xbm, BMP with Cdrom.* Addison Wesley Longman, Inc., July 1999.

[17] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In N. Belkin, P. Ingwersen, and A. M. Pejtersen, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval, Copenhagen, June 1992*, pages 274–285. ACM Press, 1992.

[18] Mark Nelson and Jean loup Gailly. *The Data Compression Book.* M&T Books, New York, NY, 2nd edition, 1995.

[19] Markus F. X. J. Oberhumer. *LZO – a real-time data compression library*, November 1999. Source code available at `http://wildsau.idv.uni-linz.ac.at/mfx/lzo.html`.

[20] P. O'Neil. Model 204 architecture and performance. In *2nd International Workshop in High Performance Transaction Systems, Asilomar, CA*, volume 359 of *Springer-Verlag Lecture Notes in Computer Science*, September 1987.

[21] Patrick O'Neil and Elizabeth O'Neil. *Database: pronciples, programming, and performance.* Morgan Kaugmann, 2nd edition, 2000.

[22] G. Panagopoulos and C. Faloutsos. Bit-sliced signature files for very large databases on a parallel machine architecture. In *Proceedings of EDBT'94. 4th International Conference on Extending Database Technology, 1994*, 1994. Appeared earlier as Technical Report CSC-809, Department of Computer Science, University of Maryland, April 1992.

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

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

[25] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional indexing and query coordination for tertiary storage management. In *11th International Conference on Scientific and Statistical Database Management, Proceedings, Cleveland, Ohio, USA, 28-30 July, 1999*, pages 214–225. IEEE Computer Society, 1999.

[26] K. Stockinger, D. Duellmann, W. Hoschek, and E. Schikuta. Improving the performance of high-energy physics analysis through bitmap indices. In *11th International Conference on Database and Expert Systems Applications DEXA 2000, London, Greenwich, UK*, September 2000.

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

[28] I. H. Witten, T. Bell, A. Moffat, and E. Fox, editors. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 2 edition, May 1999.

[29] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *Proceedings of VLDB 85, Stockholm*, pages 448–457, 1985.

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