Generalized Word Aligned Hybrid Bitset Compression Method

Stefan Wojciechowski
Computer Science UCLA

March 2012

Abstract

Compressed bitmaps indexes are a commonly used technique for efficient storage and querying of large databases. Compression algorithms attempt to reduce the size of these bitmaps while providing quick CPU computations without decompression. This paper explores 8, 16, 32, and 64 bit word implementations of the Word Aligned Hybrid (WAH)[6] compression technique. It is therefore a Generalized Word Aligned Hybrid (GWAH) with some necessary changes to the original algorithm. The original purpose of the implementation of GWAH was to compare run length encoding techniques against the BZET tree compression technique developed by Robert Uzgalis using his BZET API[5].

Uncompressed
0x00000000 0x00000000
Compressed
0x80000002

Figure 1: Uncompressed vs. Compressed WAH Representation for a 62 zero bit sequence

1 Introduction

A bitmap typically refers to an array data structure consisting of only bit values one or zero. By representing a set of data such as gender for a set of people as a bitmap, one can utilize bit-wise operations to perform efficient queries. Currently, these bitmaps can be extraordinarily large and thus costly to maintain in memory. Compression of these bit sets is a natural solution to the issue of space, but techniques such as LZ77[7] require decompression of the bitmap in order to perform bit-wise logical operations. WAH improves efficiency over these compression techniques by allowing operations upon compressed sets. WAH was designed to utilize the alignment of the word size with the machine processor to ensure efficient computation. WAH is composed of a sequence of words. These words are either a 'fill' word or a 'literal' word. A fill word is a representation of a run of zero or one bits. A literal bit is a literal representation of a segment of a bit set. To distinguish between a literal and a fill word, the most significant bit (MSB) acts as a flag. For fill words, the second most significant bit is a flag distinguishing a run of zeros or ones. Therefore a 32 bit literal word contains the natural representation of 31 bits. A fill has 30 bits to count the number of 31 bit runs of zeros or ones.

Figure 1 demonstrates the compression of 31+31 = 62 zero bits. The MSB(bit 31) is used to encode a fill word when set. The second MSB(bit 30) is used to encode a fill which is either zero or one. The 30 remaining bits represent the number of words it represents, that is, 2 words and thus 62 consecutive zero bits. One can see clearly the compression at work. The longer the string of consecutive zeros or ones the better the compression. This is of course dependent on the word length. A group of zeros less than the word size cannot be compressed. Conversely a dataset that has no consecutive grouping of ones or zeros greater than the word size will not be compressed either and in will take up more space than the uncompressed bit set because of the flag bits in each word. This opens up the possibility of various word sizes. A 16 bit implementation of WAH will compress a consecutive string of 15 bits where as a 32 bit implementation will not. GWAH may serve as a tool to compare the advantages and disadvantages of implementing particular word sizes.

The representation is canonical, and is therefore safe for usage when making comparisons and conducting bitwise operations. While non-canonical representations can also perform bitwise operations, they must consider all possible encodings of a bitmap which is computationally inefficient and carries the risk of bugs or possibly attacks. The original WAH algorithm does not address the overflow of fill count, chiefly because a 32 bit word would take more than $2^{30} - 1$ number of consecutive 31 bit segments representing approximately $3.3x10^{10}$ bits to cause an overflow. In the generalized version, suppose an 8 bit word length is used, only $2^6 - 1$ consecutive 7 bit segments representing 441 bits causes an overflow, which is quite possible within real and theoretical datasets. This required a significant alteration to the otherwise simple and elegant WAH. This change maintains a canonical form much like WAH and allows for safe bitwise operations.

2 Review of Prior Work

Byte-aligned Bitmap Compression (BBC)[1] was developed in the early 1990 by Gennady Antoshenkov. As the name suggests, BBC compressed bit sets along byte boundaries. Bytes are classified as either a gap or non-gap. An aligned byte is considered a gap byte if all the bits in a byte have the same value, otherwise it is a non-gap byte. Sequential same class bytes are then grouped together and encoded with a count representing the number of bits in a gap byte. This method maintains the ability to make computations on compressed sets and provides a good level of compression. However WAH performs better in CPU efficiency because of full utilization of CPU architecture.

Position List Word Aligned Hybrid (PLWAH)[3] is another form of WAH run length encoding that adds a position list of set/unset bits that succeed a run of zeros or ones. Upon observation that much of the counter bits in a fill word go unused, PLWAH uses these bits to represent a position list. PLWAH, for some datasets, has half the size and comparable speeds to WAH.

Compressed 'n' Composable Integer Set (CONCISE)[2] follows a similar observation to PLWAH and introduces the concept of a mixed fill word to represent a singly set bit that may occur in a sequence of 32 bits. Upon observation of the frequency of such a condition in bit sets, CONCISE is able to maximize compression whilst maintaining if not exceeding WAH in CPU efficiency.

Compressed Adaptive Index (COMPAX)[4] is another compression scheme that further improves the WAH technique. COMPAX elaborates on the literal and fill words by adding LFL (Literal Fill Literal) and FLF (Fill Literal Fill). These additional code words were added on the observation that such patterns dominated network flow traffic, which is what COMPAX was mainly developed for. COMPAX also omits the one fill word of WAH because it is very unlikely in network conditions.

3 GWAH

GWAH allows for 8, 16, 32, and 64 bit word sizes to be used in the WAH scheme. Testing has demonstrated a general trend of improvement in compression efficiency with reduced word sizes. By the same token, CPU time generally worsened with reduced word sizes.

Difference from WAH

The main change in GWAH was ensuring that overflow did not occur. The appendFill function contains most of the changes to the original WAH implementation. Upon initial inspection it appears significantly different from its original design, but it contains just a few alterations in actuality.

In the event of an overflow, GWAH simply appends the maximum count of words to the fill word operated on and pushes as many fill words as necessary until the entire count has been satisfied. This operation generates the same representation of the bit sets in all cases and is therefore canonical. The following sections provides details with respect to how GWAH operates and differs from WAH.

3.1 Binary Operations

BinOp (Algorithm 1) is the crux of the implementation. Currently, Bitwise operations AND, OR, and XOR use this generic function to return results in a bitVector z.

	bits 1,100 set
A	0x00000002 0x80000002 0x00000080
	bits 50,100 set
B	0x00000000 0x00080000 0x00000000 0x00000080
	AND Result
C	0x80000003 0x00000080

Figure 2: AND Operation on Two Compressed Sets

Figure 2 describes an AND operation on compressed bit sets for a 32-bit GWAH. A and B represents four 4-byte words. C represents 2 4-byte words. Fill words are intially decoded to reveal the word count and the type of fill it represents. Words are then aligned between the two sets and the logical operation is performed.

BinOp makes use of the run structure, runDecode and Oper function. The run data structure is used to iterate and store pertinent information regarding a word in GWAH. runDecode determines whether a word is a fill word, what type of fill, and how many words it represents. Oper is actually a function pointer to a binary operation that is passed to BinOP as an argument.

```
struct run {
align_t * it;
align_t fill;
align_t nWords;
align_t isFill;
}
```

Algorithm 1 z=BinOp(x, y)

```
//z is actually passed by reference as an argument.
while xrun.it and yrun.it are not exhausted do
    if xrun.nWords = 0 then runDecode(xrun)
    if yrun.nWords = 0 then runDecode(yrun)
    if xrun.isFill then
        if yrun.isFill then
            nWords=min(xrun.nWords,yrun.nWords)
            appendFill(z,nWords,oper(xrun.fill,yrun.fill))
            xrun.nWords, yrun.nWords −= nWords;
        else
            z.activeValue=oper(yrun.fill,*(xrun.it))
            appendLiteral(z)
            −−yrun.nWords; −−xrun.nWords
    else if yrun.isFill then
        z.activeValue=oper(*(xrun.it),*(yrun.it));
        appendLiteral(z)
        −−yrun.nWords; −−xrun.nWords
    else
        z.activeValue=oper(*(run.it),*(yrun.it))
        appendLiteral(z)
        −−yrun.nWords; −−xrun.nWords
    if xrun.nWords = 0 then ++xrun.it
    if yrun.nWords = 0 then ++yrun.it
z.activeValue=oper(x.activeValue, y.activeValue)
z.active.nbits=z.active.nbits
```

3.2 Append Fill

appendFill (Algorithm 2) checks the current fill word count and ensures that no more than the maximum count is appended to that fill word. If an overflow condition is found, another fill word is created with the remaining number of words. FILLZERO and FILLONES are the fillBit representations of a fill of zeros and ones respectively. couldAppend is the calculated space left in a fill word. In the event that a fill word is full, we simply call appendNewFills to create a new fill word.

Algorithm 2 AppendFill(bv, n, fillBit)

```
if $n > 1$ and vec is not Empty then
    if fillBit=0 then
        if bv.back is a fill of zeros then
            couldAppend=max appendable words
            willAppend=min(couldAppend,n)
            bv.back+=willAppend
            n−=willAppend
        appendNewFills(bv,n,FILLZERO)
    else
        if bv.back is a fill of ones then
            couldAppend=max appendable words
            willAppend=min(couldAppend,n)
            bv.back+=willAppend
            n−=willAppend
        appendNewFills(bv,n,FILLONE)
else if vec is empty then
    if fillBit=0 then
        appendNewFills(bv,n,FILLZERO)
    else
        appendNewFills(bv,n,FILLONE)
else
    bv.active.value = fillBit?FILLONES:0
    appendLiteral(bv)
```

3.3 appendNewFills

appendNewFills (Algorithm 3) is a helper function to appendFill. In the event of an overflow condition, a new fill word is appended until all runs have been accounted for.

Algorithm 3 appendNewFills(bv,n,fillBase)

```
while $n > 0$ do
    willAppend=min(max appendable words, n)
    bv.push(fillBase+willAppend)
    n-=willAppend
```

Set bits from 0-10,000 div by 2				
	8 bit	16 bit	32 bit	64 bit
uncomp. size(bytes)	1250	1250	1250	1250
comp. size(bytes)	1429	1334	1292	1272
comp. percentage	-14.32%	-6.72%	-3.36%	-1.76%

Figure 3: Heavy Density Set Compression Comparison

Set bits from 0-10,000 div by 300				
	8 bit	16 bit	32 bit	64 bit
uncomp. size(bytes)	1250	1250	1250	1250
comp. size(bytes)	88	134	268	536
comp. percentage	93%	89%	79%	57%

Figure 4: Medium Density Compression Comparison

4 Evaluation

The following tests were conducted on a Microsoft Windows 7 x86-64 platform, running on an Intel Core 2 Duo Processor E6700 2.66 GHz, on 3 gigabytes of 240-Pin DDR2 SDRAM DDR2 800 (PC2 6400). One core of the two were used and tests were conducted in-RAM. Prior to testing, the machine was in an idle state. The compression tests consisted of creating bit sets of varying density by setting bits if they were divisible by a particular number. CPU efficiency tests were conducted by performing a total of 100,000=10 operations * 10,000 for bitsets representing 10,000 bits. Each word size test was conducted 9 times consecutively with the time averaged. cProfile, a Python module, was used to benchmark operations. The times represented were real (wall-clock) times,

Set bits from 0-10,000 div by 1000				
	8 bit	16 bit	32 bit	64 bit
uncompressed size(bytes)	1250	1250	1250	1250
compressed size(bytes)	57	38	76	152
compression percentage	95.5%	96.96%	93.92%	87.84%

Figure 5: Sparse Density Compression Comparison

100,000 operations CPU test				
	8 bit	16 bit	32 bit	64 bit
Time (s)	2.323	1.248	0.438	0.605

Figure 6: CPU Efficiency Comparison

not CPU time. Figure 3, 4 and 5 demonstrate the general advantage of smaller word sizes on a naive bit set. The compression difference in this case is generally half the size per half the word size.
In a heavy density set (Figure 3), GWAH performs poorly across the spectrum of word sizes with 8 bit performing the worst. In a medium density set (Figure 4), the smaller the word, the better the compression. The difference between 8 and 64 bit is nearly 50%. Intuitively, the heavy and medium density tests make sense, however in a relatively sparse set as in Figure 5, 8 bit performs slightly worse than 16-bit in compression efficiency. Figure 6 demonstrates the advantages of larger sized words in CPU efficiency. This test revealed that the 32 bit word alignment was optimal while the 64 bit word, suffered a loss in efficiency. Analysis revealed that nearly all logical operations were slower for 64 bit than 32 bit usually a few ten-thousandth of a second. This could be due to compiler optimization issues which potentially may not have used some of the newer instructions available for the 6700 processor. 16 bit was approximately 3 times slower than 32 bit, and 8 bit was approximately 6 times slower.

5 Conclusions

The tests conducted on GWAH reveal important trends in computation and compression efficiency. An 8 bit word size proved to have the worst compression in the worst case as well as the worst operation time, while 64 bit had the best compression in the worst case, but worst compression in the best case, with close to the best CPU efficiency. In the middle of the word spectrum, 16 bit had better compression in the medium and sparse density tests compared to 32 bit, but fared poorly by comparison in CPU efficiency and compression efficiency in the worst case. Further, more complete testing is necessary to establish a clearer picture on the best overall choice of word size. The tests revealed however, that each word size had a distinct advantage in at least one of the tests compared to the others. Given the advantages and disadvantages of the varied word sizes, it may be possible to harness the compression advantages of variable word sizes, while maintaining a word-alignment and therefore CPU efficiency. 32 bit or 64 bit combinations of 8, 16, and 32 bit word representations may allow for such a possibility.

6 Future Work

Without a doubt, GWAH can be improved. Memory reallocation is currently implemented by doubling the currently allocated memory. Different memory allocation policies could be used in lieu of this with potentially better space efficiency.

The range function could be optimized using bit masks instead of a for loop to set bits, similar to the addBit function. Other functions could be improved by performing inplace operations. Many functions allocate a temporary bitVector such as binop (Algorithm 1).And it is probably possible to avoid this operation and thus optimize code.

Further testing is necessary to better understand the advantages and disadvantages of various word sizes. Exploration of the 64-bit word version of GWAH is necessary to determine the cause of the drop in performance, and perhaps a better optimized version for 64 bit is necessary. Testing may reveal a more concrete understanding of when certain word sizes are appropriate. It may be possible to create a variable word size aligned hybrid that capitalizes on these advantages to achieve a “best of both worlds” encoding scheme.

References

[1] Gennady Antoshenkov. Byte aligned data compression. (US 5363098), 11 1994. URL `http://www.google.com/patents?vid=5363098`.

[2] Alessandro Colantonio and Roberto Di Pietro. Concise: Compressed 'n' composable integer set. Technical report, Universit di Roma Tre, Dipartimento di Matematica, Roma, Italy, 2010, etc.

[3] François Deliège and Torben Bach Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 228–239, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-945-9. doi: 10.1145/1739041.1739071. URL `http://doi.acm.org/10.1145/1739041.1739071`.

[4] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic. *Proc. VLDB Endow.*, 3(1-2):1382–1393, September 2010. ISSN 2150-8097. URL `http://dl.acm.org/citation.cfm?id=1920841.1921011`.

[5] Robert C. Uzgalis. Bzet: A tree oriented method for compressing bitstrings. 2011.

[6] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, March 2006. ISSN 0362-5915. doi: 10.1145/1132863.1132864. URL `http://doi.acm.org/10.1145/1132863.1132864`.

[7] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. Technical report, IEE TRANSACTIONS ON INFORMATION THEORY, 1977.

Appendix

The Python API

The Bzet API uses c-types to utilize c-performance while maintaining a user friendly interface via python. GWAH supports almost all of the BZET API operations. Example Usage:

```
from RLE import rle
#create a bzet setting bits 1-100
bset1 = rle([(1,100)])
#create a bzet setting bits 1,5,10,50-100, and 300
bset2 = rle([1,5,10,(50,100),300])
bset3 = bset1.AND(bset2)
print(bset3)
```

Downloading and Installing GWAH

The GWAH source code can be found on: `https://github.com/stefanwoj89/BZET-RLE-API`
LINUX:
- run makefile
- Open RLE.py
- Edit wordSize variable to be either 8, 16, 32, 64.
- Run test *python3 compressionTest.py*

WINDOWS:
- create DLL via Visual Studio or command (see below)
- change directory to *<c:/...Microsoft Visual Studio/VC/bin>*
- Command: *cl /LD <directory of gwah.cpp>*
- Copy gwahXX-Y.dll into the main gwah directory
- Open RLE.py
- Edit wordSize variable to be either 8, 16, 32, 64.

- Uncomment the *.dll* and comment the *.so* library calls
- Change directory to your python3 directory
- Run compression test: *python compressionTest.py*

This will create the dynamic linked libraries for the various versions of GWAH in <*c:/...Microsoft Visual Studio/VC/bin*> and copy it to the folder where the other source files are.

This will create the shared objects for the various versions of GWAH.