A Concept for a Compression Scheme of Medium-Sparse Bitmaps

Andreas Schmidt*† and Mirko Beine*
* *Department of Informatics and Business Information Systems,*
University of Applied Sciences, Karlsruhe
Karlsruhe, Germany
Email: andreas.schmidt@hs-karlsruhe.de, mirko.beine@arsinventionis.de
† *Institute for Applied Computer Science*
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: andreas.schmidt@kit.edu

***Abstract*—In this paper, we present an extension of the WAH algorithm, which is currently considered one of the fastest and most CPU-efficient bitmap compression algorithm available. The algorithm is based on run length encoding (RLE) and its encoding/decoding units are chuncs of the processor's word size. The fact that the algorithm works on a blocking factor, which is a multiple of the CPU word size, makes the algorithm extremely fast, but also leads to a bad compression ratio in the case of medium-sparse bitmaps (1% - 10%), which is what we are mainly interested in. A recent extension of the WAH algorithm is the PLWAH algorithm, which has a better compression ratio by piggybacking trailing words, which look "similar" to the previous fill-block. The interesting point here is that the algorithm also is described to be faster than the original WAH version under most circumstances, even though the compression algorithm is more complex. Therefore, the concept of the PLWAH algorithm was extended to allow so-called "polluted blocks" to appear not only at the end of a fill-block, but also multiple times inside, leading to much longer fill lengths and, as a consequence, to a smaller memory footprint, which again is expected to reduce the overall processing time of the algorithm when performing operations on compressed bitmaps.**

***Keywords*-Compressed bitmaps, WAH algorithm, RLE, CPU-memory-gap**

I. Introduction

Compressed bitmaps play an increasingly important role in efficiently answering multi-dimensional queries in large data sets. Another application is the representation of *positionlists* inside column-stores [1]. We are presently developing a framework with basic components to build column-store applications. Besides *ColumnFile* and *ColumnArray* as basic components, we also identified the *positionlist* as a key component of our framework. A *positionlist* for example is responsible for buffering the data sets that satisfy a condition on a column. This is done by storing a list of tuple-ids. The tuple-ids are sorted and have no duplicates. If the lists are short, tuple-ids can be stored as `INT(4)` values, but in the case of millions of entries in a *positionlist*, the (compressed) bitmap is the more appropriate representation form. After analysing the relevant scientific papers about bitmaps, we identified the well-known WAH algorithm [2] as one of the candidates for implementing our *positionlist*. One drawback of the algorithm was that an efficient compression is only possible when the selectivity is about 0.1% and below. In our operational area, however, also selectivities between 1% and 10% should be handled efficiently. A recent extension of the WAH algorithm is the PLWAH algorithm [3], which has a better compression ratio (up to a factor of 2) by piggybacking trailing words, which look "similar" to the previous fill block. The interesting point here is that the algorithm also is described as faster than the original WAH version under most circumstances even though the compression algorithm is more complex. This leads to the assumption that the CPU memory gap [4] has shifted the algorithm from CPU bound to IO bound in the past years and that the bottleneck of the algorithm is no longer the CPU, but the access to the main memory. In this case, processing time may be reduced by finding a better compression for selectivities between 1% and 10%.

The paper is organised as follows. In the next section, we introduce the main concepts of the WAH algorithm. Afterwards, we present our extension of the WAH algorithm, which introduces a new fill type that cannot only handle identical bits in a fill, but also allows for the existence of a number of pollutions inside. Subsequently, we explain our concept using an example and after that, a number of possible variants will be discussed. Our paper will be completed with a short summary and a longer list of activities once the implementation of our algorithm will be available.

II. Related Work

The WAH algorithm is a compression algorithm for bitmaps. It is based on run length encoding and allows for efficient operations on the compressed versions of the bitmaps. It is very CPU-efficient, because it uses the CPU word size as basic packing unit, which allows very efficient operations on the data. Two types of blocks are distinguished. *Literal* blocks contain uncompressed bits and *fill* blocks contain a number of subsequent identical bit values. In the remaining of the paper, we will focus, without loss of generality, on the 32-bit version of the algorithm. In this case, each *literal* block contains 31 uncompressed arbitrary bits and each *fill* block holds a multiple of 31 bits with the same value. The first bit of a block is used to distinguish a *fill* block from a *literal* block. To separate a *0-fill* from a *1-fill*, the second bit is used. 30 bits are left to indicate the length of a fill. The length is given in multiple of 31 bits, not in individual bits. A value of 2 means a fill with 62 identical bits. Figure 1 shows the compression of a bitmap of 194 bits length (first line). First, the bitmap is divided into equidistant parts of 31 bits (second line) and these parts are further classified as *fill* or *literal*. After that, consecutive fills with the same bit value are combined.

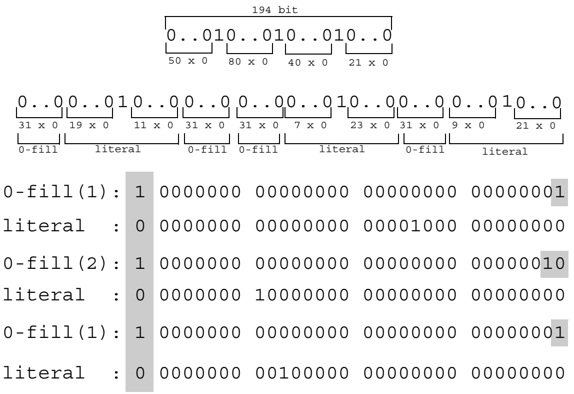
194 bit
0..010..010..010..0
50 x 0
80 x 0
40 x 0
21 x 0
0..00..010..00..00..00..010..00..00..010..0
31 x 0
19 x 0
11 x 0
31 x 0
31 x 0
7 x 0
23 x 0
31 x 0
9 x 0
21 x 0
0-fill
literal
0-fill
0-fill
literal
0-fill
literal
0-fill(1): 1 0000000 00000000 00000000 00000001
literal : 0 0000000 00000000 00001000 00000000
0-fill(2): 1 0000000 00000000 00000000 00000010
literal : 0 0000000 10000000 00000000 00000000
0-fill(1): 1 0000000 00000000 00000000 00000001
literal : 0 0000000 00100000 00000000 00000000

Figure 1. Bitmap compression with WAH

The drawback of this algorithm is that in the case of medium-spare bitmaps, the *fills* are very short and every single "pollution", leads to a full literal block. The switch between a fill and a literal (and back to a fill) block is an expensive job in terms of memory.

III. CONCEPT

The main difference between WAH and our concept is that we support the concept of *draggled fills*, which allows a small number of false bits inside each word of a fill. The intention here is to obtain longer fills, because the switch from a fill to a literal block and back to a fill is an expensive act in terms of memory. The PLWAH (position list word aligned hybrid) method uses a related concept by piggybacking a trailing literal block after a fill, if it differs from the words in the preceding fill by one bit only. For this purpose, the length field in the fill is reduced by some bits, while five of these bits indicate the position of the wrong bit in the trailing literal. With this trick, you can achieve a reduction by a factor of two for certain distributions of data. Otherwise, the maximum length of a fill is reduced by a factor of 2^6, may reach a maximum of 2^{24} instead of 2^{30}.

In contrast to this, our concept does not only allow for one slightly polluted literal at the end of a fill, but it also allows for slightly polluted literals to appear at each position in the fill without reducing the overall length of a fill.

A. Draggled Fill

Our concept requires the introduction of a new block type called *draggled fill*, which can handle the polluted literals inside a fill. In contrast to the other two block types *literal* and *fill*, a *draggled fill* has a variable length depending on the number of pollutions inside. Hence, three different types of blocks (literal, fill, and draggled fill) must be distinguished. We distinguish a *fill* from a *draggled fill* with the third significant bit, so that a 1-fill is identified by the bit combination of 111, while a draggled-1-fill is identified by 110 (0-fill: 101, draggled-0-fill: 100). The indicator of a *literal* remains identical to the WAH algorithm (a 0-bit at the most significant bit), which still allows us to store 31 bits in each *literal*. For every word in a *draggled fill*, we first have to define how polluted it could be to be part of such a fill. For a 32-bit version of the WAH algorithm, different degrees of pollution can be defined, which vary from one wrong bit inside 32, 16, 8, and 4 bit, leading to 1, 2, 4, or 8 wrong bits (called pollution factor) in a complete 32-bit word[1]. Figure 2 presents examples of different pollution factors, each with the maximum number of skipped bits.

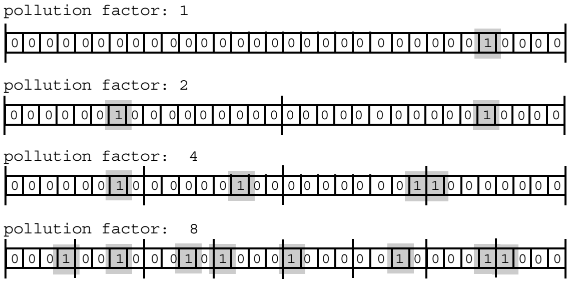

Figure 2. Possible pollution factors for a block

Each polluted 32-bit word needs a fixed number of bits for description. The value of needed bits is dependent on the pollution factor and the maximum length of a fill. In case of a pollution factor of 1, we only need to specify the position of the wrong bit, which could be done with 5 bits ($2^5 = 32$). With a pollution factor of 2, we need 4 bits to specify the position of the wrong bit in the first 16 bits and another 4 bits to specify the wrong bit in the second half of the word. As only one of the two 16-bit words may contain a wrong bit, we need a mask of another 2 bits to specify in which part the skipped bits occur. Table I gives an overview of the memory consumption also for the other pollution factors.

Additional memory is needed to specify the position of the polluted words. The size is dependent on the maximum

[1] Strictly speaking, we do not have 32 bits, but only 31 bits as packing unit. But for the sake of straightforwardness in explaining the concept we talk in this paper about 32 bits. Keep in mind that, without loss of generality, one bit can be ignored, i.e. the leftmost one.

Table I
MEMORY CONSUMPTION OF DIFFERENT POLLUTION FACTORS

Pollution factor	**Memory consumption** (in bit)
1	5
2	10 $(2 * 4 + 2)$
4	16 $(4 * 3 + 4)$
8	24 $(8 * 2 + 8)$

length of a fill. If for example the maximum value is 1024 (2^{10}), 10 additional bits are required to specify the position for each pollutted 32-bit word in the most simple implementation, where the position is specified by its index inside the run. Later in section III-C, we will discuss different possibilities to identify the wrong words.

B. Example

After the introduction of the concept, the effect will now be demonstrated using the example given in Figure 3.

In the middle part of the Figure, seven 32-bit blocks can be seen. Except for the fourth and the sixth block, which contain two and one polluted bit(s) (indicated in grey) all blocks contain 0-bits only. The two polluted blocks are shown in detail in the upper and lower part of the Figure. The pollution factor is set to 2, meaning that we can accept one wrong bit in every 16-bit of the block at the most. So both polluted words can be accepted to be inside a *draggled fill* and the overall length of the fill is 7 words. Besides the overall length, we have to provide additional information for a *draggled fill*. This information includes:

- The number of polluted blocks
- The positions of the polluted blocks inside the fill
- Position of the wrong bits inside a polluted block

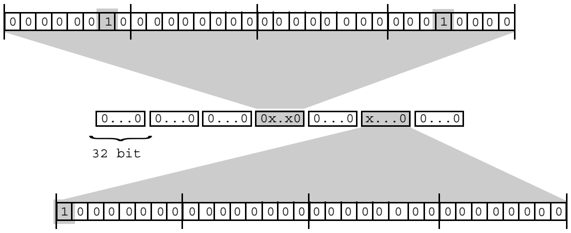

Figure 3. *Draggled fill* with two polluted blocks

The maximum number of polluted blocks depends on the maximum length of a *dragged fill* and the number of bits to specify the number. The same holds for the specification of the position of the polluted blocks. In our example, we choose a maximum length of a *draggled fill* of 64 and a pollution factor of 2. This means that we need 6 bits to specify the size of the fill and another 6 bits to specify the number of polluted blocks inside the fill. For each polluted block, we also have to provide the information on the position of the block inside the fill and the wrong bits inside. Figure 4 shows a possible memory layout for the above example in the upper part. The first three bits are reserved for the block type, then 6 bits for the fill length field, and another 6 bits for the field indicating the number of polluted blocks.

In the lower 16 bits of the first word, the information about the individual polluted words inside a fill is contained. In the defined layout (maximum length: 64, pollution factor: 2), we need exactly 16 bits to specify one pollution word. The first two bits, labeled as “mask”, identify in which of the two 16-bit words a pollution occurs. Possible values are 01, 10, and 11. The next 6 bits specify the position of the polluted word inside the fill. As a maximum of 1 wrong bit can occur inside one 16-bit word, we need 4 more bits to specify the position (0..15) of the wrong bit inside a 16-bit word. As we have two 16-bit words in our polluted block, we need another 4 bits for the second word. In each following 32-bit word, we can now store the information of two more polluted words.

In the lower part of Figure 4, the true values for the example in Figure 3 are presented. First, the block type for a *draggled-0-fill* is specified, followed by the information of a fill length of seven with two polluted words. Then, the ’11’ mask indicates, that there are two skipped bits in the polluted block at position 4 in the fill. The two skipped bits can be found at bit-position 9 (first 16-bit word) and bit-position 4 (second 16-bit word), respectively. In contrast of this the second polluted block only contains one wrong bit in the first 16-bit word (mask ’10’), which can be found at position 15.

The total memory footprint is 64 bits, compared to 160 bits in the original WAH implementation[2] and 128 bits in the PLWAH implementation. Especially in cases of lower selectivity, the proposed concept is superior with regard to memory footprint. The high memory cost of switching from a fill to a literal block and back can be avoided in many cases. And even in the case where no fills can be found, there is no drawback due to the fact that a literal block can handle 31 bits as in the original WAH-algorithm. One little drawback exists in the case of a very high selectivity leading to extremely long fills: Because of the new block type, the proposed concept needs one bit more to indicate a fill block, and so a block can contain a maximum of $2^{29} * 31$ bits instead of $2^{30} * 31$. As our concept has not been yet implemented, we cannot make any statements about the runtime behaviour. However, we plan to run a bunch of experiments with different data distributions concerning runtime and memory behaviour.

C. Variants

In the above concept we divided each 32-bit block into equidistant parts, which can contain 1 wrong bit at the most.

[2] 160 bits = 32 bits (0-fill, length: 3) + 32 bits (literal word) + 32 bits (0-fill. length: 1) + 32 bits (literal) + 32 bits (0-fill, length: 1)

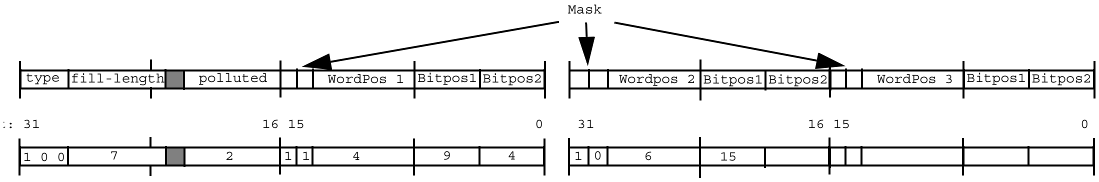

Figure 4. Memory layout of a *draggled-0-fill*

This solution was chosen, because it is easy to implement and also CPU-efficient.

Another, more general solution may be not to divide the block into equidistant parts, but to allow a maximum of n-skipped bits to appear inside a 32-bit block. In this case, the memory consumption is a little bit higher, but it is a more general model, which can lead to longer fills.

Instead of specifying the index position of a polluted block, it is also possible to specify the gaps between polluted blocks (incremental encoding [5]). This leads to a smaller memory footprint for each polluted block, because a lower number of bits can be used to specify the increments. In case the next polluted block is too far away to code the distance with the chosen number of bits, the fill has to terminate. Figure 5 gives an example of this encoding. Each gray square represents a 32-bit block (with unique values, polluted and mixed). The full length of the fill is 21 blocks. As you can see, the values of the increments remain small in contrast to the index encoding in the last line, thus allowing for a lower number of bits to encode the fill.

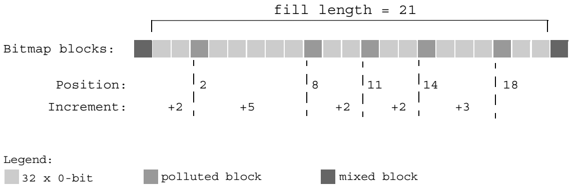

Figure 5. Incremental encoding of “polluted blocks”

All of the above variants require a predefined fixed number of bits to encode the position of the polluted blocks. Another possible solution would be to use a Rice (Golomb) coding [6]. The idea behind this coding scheme is to use a flexible number of bits to encode arbitrarily long integer numbers. Small, but frequently appearing numbers only need a small number of bits, while unfrequent big numbers need more bits as in a normal coding scheme.

IV. Conclusion

We presented an extension of the WAH algorithm, which is currently considered one of the fastest and most CPU-efficient compression techniques for bitmaps. However, in the case of a selectivity of 1% and more, the compression behaviour is unsatisfying. The reason for this behaviour is the blocking factor of 32, which requires packing of a minimum of 31 bits. Even a single skipped bit leads to a literal block, which holds 31 uncompressed bits.

Our contribution handles this problem by allowing so-called polluted blocks to be part of a fill. A polluted block is a block, which has a limited number of wrong bits. The idea is to describe the position of the polluted block and the wrong bits inside it, which takes much less memory than ending a fill, starting a new literal block, and after that starting a new fill.

V. Future Work

Currently, we don’t have an implementation of our concept, but we are working on it. At the time we have a our implementation finished, we plan a number of tests with different selectivity, both synthetical and real world data, comparing both the compression ratio and the execution time of the different operations. Depending on the results we eventually implement different variants of our algorithm, discussed in III-C. Another interesting point would be to look for dependencies between the pollution factor and the maximum fill-length for different data sets.

References

[1] D. J. Abadi, S. R. Madden, and M. Ferreira, “Integrating compression and execution in column-oriented database systems,” in *SIGMOD*, Chicago, IL, USA, 2006, pp. 671–682.

[2] K. Wu, E. J. Otoo, and A. Shoshani, “Compressing bitmap indexes for faster search operations,” in *SSDBM ’02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 99–108.

[3] F. Deliège and T. B. Pedersen, “Position list word aligned hybrid: optimizing space and performance for compressed bitmaps,” in *EDBT ’10: Proceedings of the 13th International Conference on Extending Database Technology*. New York, NY, USA: ACM, 2010, pp. 228–239.

[4] S. Manegold, P. A. Boncz, and M. L. Kersten, “Optimizing database architecture for the new bottleneck: memory access,” *The VLDB Journal*, vol. 9, no. 3, pp. 231–246, 2000.

[5] I. H. Witten, A. Moffat, and T. C. Bell, *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[6] S. W. Golomb, “Run-length encodings,” *IEEE-IT*, vol. IT-12, pp. 399–401, 1966.