A Proposal of a New Compression Scheme of Medium-Sparse Bitmaps

Andreas Schmidt*†, Daniel Kimmig*, and Mirko Beine†
* *Institute for Applied Computer Science*
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Email: {andreas.schmidt, daniel.kimmig}@kit.edu
† *Department of Informatics and Business Information Systems,*
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
Email: andreas.schmidt@hs-karlsruhe.de, bemi0029@hs-karlsruhe.de

***Abstract*—In this paper, we present an extension of the WAH algorithm which is currently considered to be one of the fastest and most CPU-efficient bitmap compression algorithms available. The algorithm is based on run-length encoding (RLE) and its encoding/decoding units are chunks 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 that has a better compression ratio due to piggybacking trailing words, looking "similar" to the previous fill-block. The interesting point here is that the algorithm is also described to be faster than the original WAH version under most circumstances, even though the compression algorithm is more complex. Based on this observation, we extended the concept of the PLWAH algorithm to allow so-called "polluted blocks" to appear not only at the end of a fill-block, but also multiple times inside. This leads to much longer fills 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

One of the main reasons for the work reported here is the increase of the CPU memory gap [1] over the last years. In the context of database applications, processors nowadays are able to process data much faster than the data can be delivered from the main memory to the processor. In many database applications this leads to a situation where the processor(s) spend(s) considerable time waiting for processable data. For this reason, modern processors are equipped with additional cache memory hierarchies which are placed on the processor itself to allow for a much faster access to memory. Accessing a data item that is present in the first-level cache is up to two orders of magnitudes faster than accessing a data item residing in main memory only. Techniques like cache-conscious algorithms [2], [3] or special memory layouts like in *column store* databases allow for further optimisations to minimise the waste of processor time.

A. Column Stores

In a database system, relations (rows in a database table) are typically stored together physically. This is illustrated in Figure 1 at the bottom left. This storage organisation is called a *row store*. A *column store* database system by contrast stores all values of a specific column sequentially. This storage organisation is visualised on the lower right of Figure 1. The information which column value corresponds to which relation is handled by a tuple identifier (TID). The TID could be stored explicitly with the column values, but is generally given implicitly by the position of the value in the column.

The main advantage of a column store is that only data values from columns that took part in a specific query are loaded into the CPU cache and memory[1], in contrast to a row store, where also unnecessary attributes may be loaded, which are irrelevent in the current context. A disadvantage of column stores, on the other hand, is that in case of updates of existing relations or insertions of new relations, lots of write operations at different positions have to be done, which is more expensive compared to writing data at only one physical location.

Based on the splitting of the relations along their columns, complex predicates have to be processed on a column basis instead of row by row. The reason for this approach is the prefetching behaviour of modern processors. If a dataset is requested by the CPU, not only the requested data item, but also the content of the following memory area is copied into the CPU cache[2]. As a consequence of this behaviour, the descision whether a complex condition is fulfilled by a

[1]From secondary storage to main memory and also from main memory to the CPU-cache.

[2]With every access to the main memory, a full cache line (8-128 Bytes) is loaded. This has the advantage that in case of further requests to the main memory, the requested dataset may already be in the CPU cache.

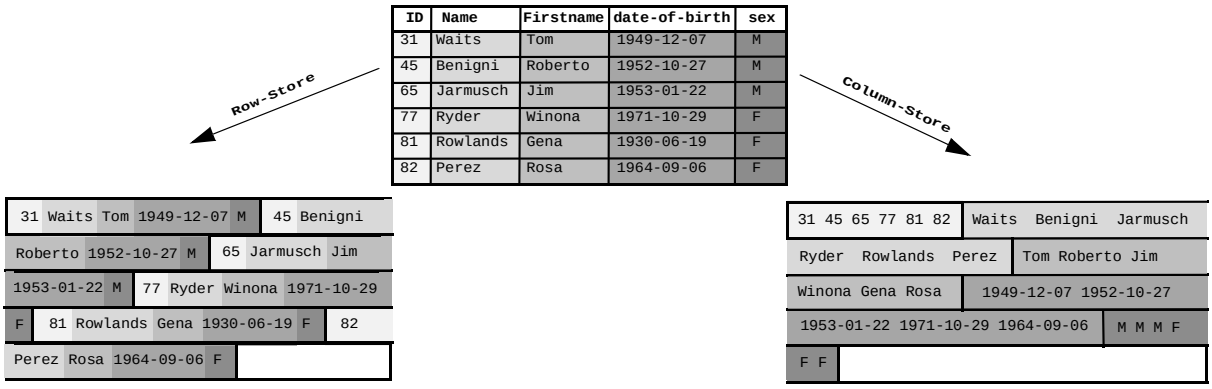


Figure 1. Comparison of memory layouts of a row store and a column store

relation must be delayed up to the examination of the last column.

For this reason, we need an additional datastructure, which remembers the intermediate status of every relation processed so far. This datastructure is called a *positionlist*. A positionlist stores the tuple-ids (TIDs) of the currently qualified relations. The processing of a complex condition generates a positionlist for every single predicate which contains the TIDs of the qualified relations. Afterwards, the positionlists must be combined with *and* - or *or*-semantics. Figure 2 illustrates this behaviour for the following query:

```
select id, name
  from person
 where birthdate < '1960-01-01'
   and sex='F'
```

First, the predicates $birthdate$ <'1960-01-01' and sex ='F' must be evaluated, which results in the positionlists *PL1* and *PL2*. These two evaluations could also be done in parallel. Next, an *and*-operation must be performed on these two positionlists, resulting in the positionlist *PL3*. As we are interested in the names of the persons that fulfil the query conditions, we have to perform another operation, which finally returns the entries for a column, specified by the positionlist *PL3*. Positionlists store the TIDs in ascending order without duplicates. For this reason, the typical *and*/*or* operations can be performed very fast. The complexity for both operations is $O(\|Pl1\| + \|Pl_2\|)$. Furthermore, the two most important operations *and* and *or* can be performed as bitwise logical operations by utilizing the corresponding primitive CPU commands. By exploiting bit-level parallelism, these operations can be performed extremely fast; with every CPU command, 32 or 64 TIDs (depending on the processor architecture) can be processed. If the positionlists are sparse (meaning we have only a small number of bits set), the underlying bitmap could be compressed easily using run-length encoding (RLE) [4]to further reduce the memory that is required to store the data structure. The main advantage of RLE is that compression and decompression can be carried out very fast compared to other compression methods. Furthermore, by using specialised algorithms, the *and*- and *or*-operations can be performed directly on the compressed lists, which improves the performance even further.

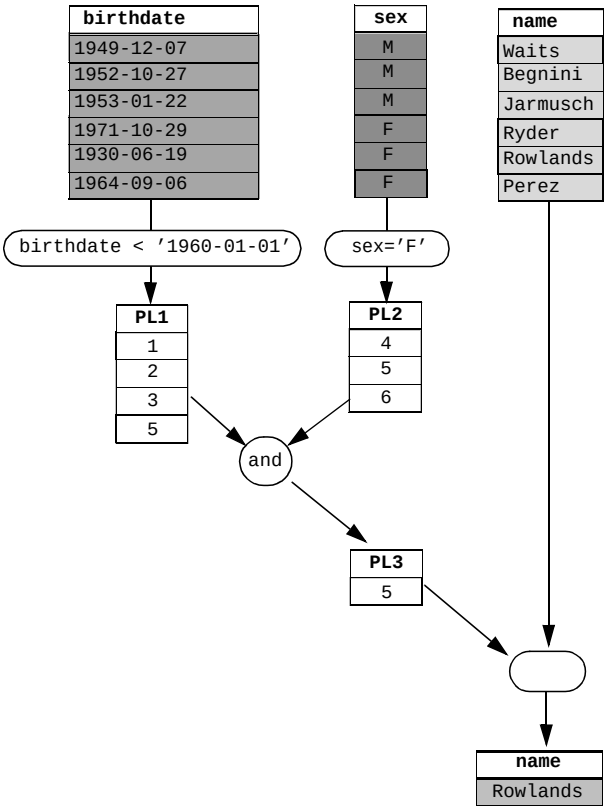


Figure 2. Processing of a query with positionlists

B. Run-Length Encoding

The basic concept of RLE is that for consequtive, identical data elements, only one data element is stored, along with

the number of consecutive occurrences. Such a consecutive sequence of identical data elements is called a *run*, and the number of occurrences is called the *run-length*. RLE is suitable, if many of such runs can be found in a stream of data. Due to these simple basics, compression and decompression can be implemented rather easily compared to more sophisticated techniques like LZ77 [5]. This is an important advantage as because of this simplicity, bitwise logical operations can be carried out without the need to decompress the bitmaps participating in a query.

Figure 3 gives a short example of the application of RLE to a character string.

The rest of the paper is organised as follows. In the next section, we present related work that has been done in the field of bitmap compression. Particularly, we describe the main concepts of the well-known Word-Aligned Hybrid (WAH) [6] algorithm and a recent extension of it, the Position List Word-Aligned Hybrid algorithm [7]. In Section III, we present a new compression scheme that is also based on the WAH algorithm, but introduces a new fill type that is capable of handling "polluted" runs of bits, i.e., runs that contain *mostly* of identical bits. We explain our concept using an example and discuss possible variations of our scheme.

In Section IV, we discuss the results of different experiments performed on synthetic bitmaps to analyse factors that influence the behaviour of our compression scheme.

Our paper will be completed by a short summary and a list of future work that will be tackled once the implementation of our algorithm will be available.

II. Related Work

Apart from their application in the context of positionlists, bitmaps also play an important role in answering multi-dimensional queries in huge datasets. The main idea here is that you have a bitmap for every distinct value of a column [8], [9], [10]. Similar to the example presented earlier, a set bit at position n of a bitmap indicates that a dataset has the specific value at this position. The result set (the TIDs of the relations that match the query criteria) of a multi-dimensional query is then the result of the appropriate *and* or *or* operations on the bitmaps. In this case, the bitmaps represent a special index structure for fast multi-dimensional query processing. This is a very well-known scientific field and a lot of literature as well as implementations of concrete algorithms, i.e., [6] can be found.

The problem in finding a suitable representation form and appropriate algorithms for our positionlist can be mapped onto solutions for multi-dimensional query processing.

The currently fastest algorithm for bitmap operations are based on the *word-aligned hybrid* (WAH) compression algorithm [11]. The main characteristic of this algorithm is that it is very CPU-efficient, but leads to a bad compression factor in case of selectivities of 1% and above.

As the algorithm already is a few years old and the CPU memory gap is still growing, a difference of up to two orders of magnitude exist in accessing the CPU cache instead of the main memory. Our goal is to develop a new algorithm which does a better job in compressing bitmaps, with selectivities between 0.01% and 10%, but is still IO-bound to further benefit from the increasing CPU memory gap as times goes on.

A. The Word-Aligned Hybrid Algorithm

The WAH algorithm is considered to be one of the fastest bitmap compression algorithms when it comes to the performance of bitwise logical operations on compressed bitmaps. The main reasons for its efficiency are the simplicity of the compression scheme and the word alignment requirement [12]: the size of WAH data units is determined by the word size of the underlying computer architecture, making WAH very CPU-efficient.

The scheme distinguishes between two types of blocks: *literal words* and *fill words*. A literal word stores a heterogeneous sequence of bits (i. e. a sequence that contains a mix of set bits and unset bits). A fill word encodes consecutive, homogeneous sequences of bits (i.e., where all bits have the same value). The most significant bit (MSB) of a word is used to distinguish between a fill word(1) and a literal word(0). In case of a CPU word size of 32, a literal word can thus store 31 bits (hereinafter, we will focus, without loss of generality, on the 32-bit version of the algorithm). Fill words can encode sequences of consecutive either set or unset bits, so one more bit is needed to distinguish a 0-fill word from a 1-fill word (also called the *fill bit*). This leaves 30 bits to encode consecutive homogeneous sequences of bits. This number is called the *fill length*. As WAH imposes the word alignment requirement, the fill length does not describe the total length of such a sequence in bits. Instead, the fill length resembles a multiple of 31-bit-sized, consecutive groups that have the same value. For example, a fill word with a fill length of 2 represents two consecutive blocks, and each block, when decompressed, has a size of 31 bits.

Figure 4 shows the compression of a bitmap of 217 bits (first box). First, the uncompressed bitmap is divided into equidistant parts of 31 bits (second box). Each part is classified as *fill* or *literal* (third box). Finally, consecutive fills with the same bit value are combined to single fills with a corresponding fill length (fourth box).

Figure 5 shows the binary representation of the compressed bitmap from Figure 4. The MSB defines the block type: a 1 (fill) at the beginning of the first, third, and fifth word, and a 0 (literal) for all the other words. In case of a fill, the second bit defines the proper fill type; as we have 0-fills only, all the second MSBs in the fill words are also set to 0. The remaining bits of each fill word are used to encode

AAAAAAAAAA BBBBBBBB AAAA BBBBBBBBBBBBBBBBBBBB AAAAAAAAAA

10A 8B 4A 20B 11A

Figure 3. Example of run-length-encoded compression

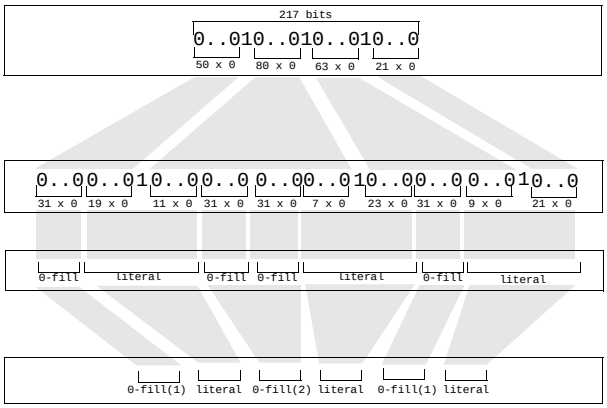


Figure 4. Bitmap compression with WAH

the fill length. Both the header bits and the fill lengths are highlighted in grey.

```
0-fill(1): 10 000000 00000000 00000000 00000001
literal  : 00 000000 00000000 00001000 00000000
0-fill(2): 10 000000 00000000 00000000 00000010
literal  : 00 000000 10000000 00000000 00000000
0-fill(1): 10 000000 00000000 00000000 00000001
literal  : 00 000000 00100000 00000000 00000000
```

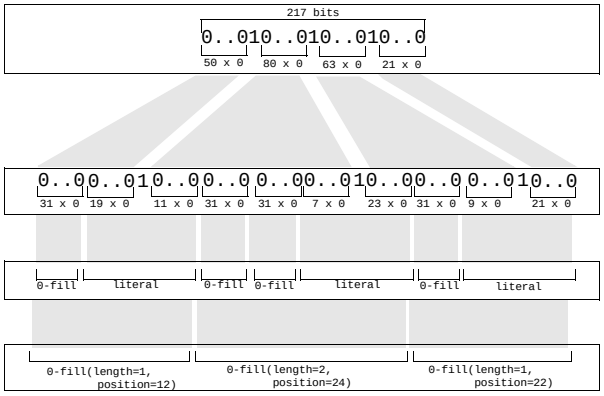
Figure 5. Binary representation of the WAH example in Figure 4

The drawback of this algorithm is that in case of medium-spare bitmaps, the *fills* are very short, as every single "pollution"[3] causes a running fill to terminate and thus leads to a full literal block. The switch from a fill to a literal (and back to a fill) block is an expensive job in terms of memory.

B. Position-List-Word-Aligned-Hybrid Algorithm

PLWAH is a modified WAH compression scheme that exceeds over WAH in terms of both compression efficiency and the performance of bitwise logical operations. It is based on the observation that in many cases, most of the bits that are available to encode the length of a fill are never needed. PLWAH uses some of these otherwise wasted bits to hold a positionlist[4]. This positionlist is then used to encode a slightly polluted word that immediately follows a fill word.

So, for example to hold the information about a 32-bit literal word that differs in only one 1 bit from a fill word, 5 bits are needed to describe the position of this wrong bit ($2^5 = 32$)[5].




```
0-fill(1,12): 10 011000 00000000 00000000 00000001
0-fill(2,24): 10 110000 00000000 00000000 00000010
0-fill(1,22): 10 101100 00000000 00000000 00000001
```

Figure 6. Bitmap compression with PLWAH

Figure 6 shows the compression of the same bit stream used in figure 4 with the PLWAH algorithm. Clearly, all three literals can be piggybacked by the fills that precede them. This leads to an improvement of the compression ratio by about 66% compared to the original WAH algorithm.

With this trick, a reduction by a factor of two can achieved for certain distributions of data. Otherwise, the maximum length of a fill is reduced by a factor of 2^5 and may reach a maximum of 2^{25} instead of 2^{30}. This could lead to a degradation of up to a factor of 5 for very sparse bitmaps[6] compared to WAH.

III. Concept

The main difference between WAH/PLWAH and our concept is that we support the concept of *draggled fills*,

[3] *pollution*n refers to a single wrong bit

[4] This positionlist must not be mixed up with the positionlists in column stores. Here, the positionlist specifies places of skipped bits.

[5] for higher pollutions (i.e. 2, 3, ... wrong bits) 10, 15, ... bits are needed to store the position of the wrong bits

[6] densities of $3 * 10^{-8}$ and below

which allows for 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.

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

In addition to the basic WAH word types, 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 polluted words 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 most 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. To decide whether or not a literal can be part of a draggled fill, we first have to define the maximum number of polluting bits that are allowed to occur in such a polluted word.

For a 32-bit version of the WAH algorithm, different degrees of pollution can be defined, from one wrong bit inside 32, 16, and 8 bits (called pollution factor), to 1, 2, or 4 segments containing wrong bits in a complete 32-bit word[7]. Figure 7 presents examples of different pollution factors, each with the maximum number of skipped bits.

Figure 7. 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 upper half of the word (i.e., the first 16 bits), and another 4 bits to specify the position of the wrong bit in the second half of the word. As there is the possibility that only one of the halves contains a polluting bit, a mask of another 2 bits is needed to specify in which part(s) of the word the pollutions occur. Table I gives an overview of the memory consumption for the remaining pollution factors (pf). The formula for the pollution factor is:

$$mem = log_2(32/pf) + mem\_mask$$

with

$$mem\_mask = \begin{cases} 0 & \text{if } pf = 1 \\ pf & \text{if } pf > 1 \end{cases}$$

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

Table I
MEMORY CONSUMPTION OF DIFFERENT POLLUTION FACTORS

Pollution factor	**Memory consumption** (in bit)
1	5
2	10
4	16

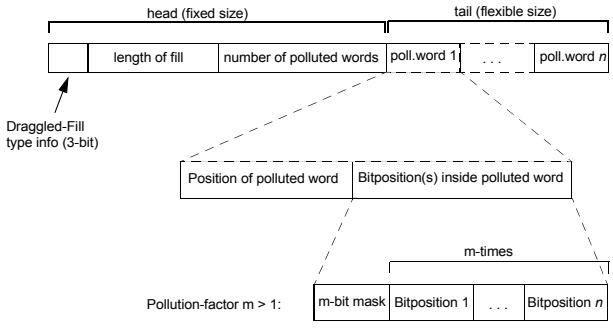


Figure 8. Structure of a *draggled fill* header

Figure 8 shows the structure of a draggled fill. In contrast to literal words and fill words, a draggled fill has a variable size. The fixed-size head contains information about the word type itself, followed by the overall draggled fill length

[7]To be exact, we do not have 32 bits, but only 31 bits as packing unit. But for the sake of straightforwardness the concept is explained based on 32 bit throughout this paper. Keep in mind that, without loss of generality, one bit can be ignored, i.e. the leftmost one.

and the number of polluted words encoded in the draggled fill. The flexible length tail contains information about each polluted word. The concrete size of a polluted word depends on the allowed number of pollutions (as defined by the pollution factor) and the maximum number of polluted words that can be encoded in a single draggled fill. Each polluted word is described by its position inside the draggled fill and the position(s) of the polluting bit(s). In case of a pollution factor > 1, an additional bitmask is required to determine whether or not a bit is set in the specific part of the word[8]. Having defined the layout of a draggled fill, it is necessary in the next step to determine appropriate values for the maximum length of a draggled fill, the maximum number of polluted words, and the number of allowed pollutions in each polluted word (pollution factor). For this purpose, several experiments were conducted on synthetic bitmaps with varying distributions and densities, as will be reported in Section IV. However, to clarify the ideas behind our concept, we will first demonstrate the function of the algorithm by a simple example. In this example, we assume a pollution factor of 2 and both a maximum fill length and a maximum number of polluted words of 64, meaning that a draggled fill could consist entirely of polluted words.

B. Example

After the introduction of the concept, the effect will now be demonstrated using the example given in Figure 9. In the middle part of the Figure, seven 32-bit blocks can be seen. Except for the fourth and the sixth block, in which the former contains two, while the latter contains one polluted bit(s) (indicated in grey), all remaining 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 may be incorporated in 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

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, without loss of generality, a maximum length of a *draggled fill* of 64[9] and a pollution factor of 2. This means that we need 6 bits ($2^6 = 64$) to specify the size of the fill

[8]This extra bit could also be added to the bit position itself, using $0\ldots0$ as a marker that no bit skip occurred in this part of the word.

[9]for this simple example a value of 8 would be enough - but this does not seem to be a realistic number

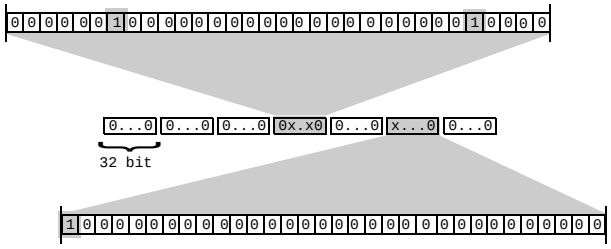


Figure 9. *draggled fill* with two polluted blocks

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 10 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 first polluted word inside the fill is contained. In the defined layout (maximum length: 64, pollution factor: 2), we need exactly 16 bits to specify one polluted word. The first two bits, labelled as "mask", identify in which of the two halves of the word pollutions occur. 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 block, we need 4 more bits to specify the position (0..15) of the wrong bit inside a single block. As a pollution can occur in the upper 16 bits and/or the lower 16 bits of a word, a total of 8 bits is needed to encode the positions of the polluting bits. In each following 32-bit word, we can now store the information of two more polluted words - one in the upper half, and one in the lower half of the word.

In the lower part of Figure 10, the corresponding bit values for the example in Figure 9 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). In contrast to 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[10] and 128 bits in the PLWAH implementation. Especially in cases of lower selectivity, the proposed concept is superior with regard to

[10]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)

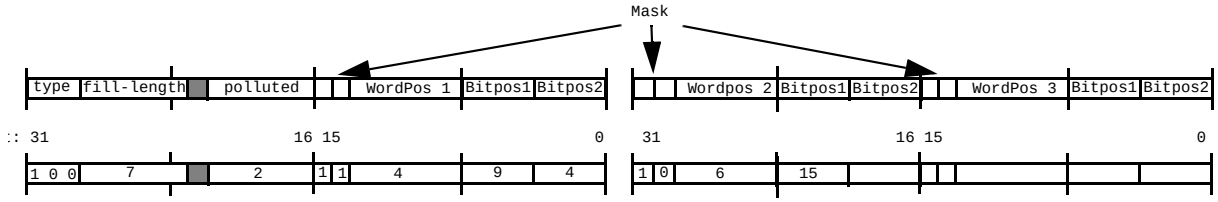


Figure 10. Memory layout of a *draggled-0-fill* with pollution factor 2

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. However, one small 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 $2^{30} * 31$.

C. Variants

In the above concept, we divided each 32-bit block into equidistant parts, which can contain 1 wrong bit at the most. 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 [4]). 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 11 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.

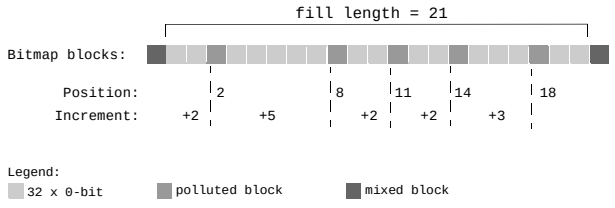


Figure 11. Incremental encoding of "polluted blocks"

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 [13]. 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.

Figure 12 and Figure 13 show a possible encoding of the examples from Figure 4 (WAH) and 6 (PLWAH) for the draggled fill WAH algorithm using incremental encoding of the polluted blocks. As you can see in Figure 13, another reduction of about 33% can be achieved, additionally leaving some bits unused, which would be able to hold another polluted word (if existent). The first three bits indicate the header type (*0-draggled-fill*). The next six bits give the overall length of the fill (7), followed by five bits indicating the number of polluted words inside the fill (3). Depending on the value (n) in this field, $\lceil (n-1)/3 \rceil$ words[11] follow to hold the information about each polluted word. The information about the first pollution can be stored in the lower 16 bits of the *draggled-fill-header*. Here in our example, the first polluted word follows after one *0-fill* word and has its pollution at bit 12. The first ten bits in the second word (flexible part) hold the information that the next polluted word follows after a gap of two *0-fill* words and the polluted position is bit 24. With this encoding (5 bits to store the gap), a maximum gap length between two polluted words can be 32. The 'x' characters indicate unused bits.

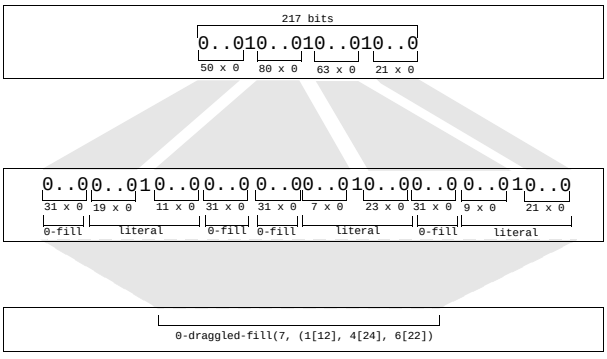


Figure 12. Bitmap compression with DFWAH

[11] $\lceil (3-1)/3 \rceil = 1$

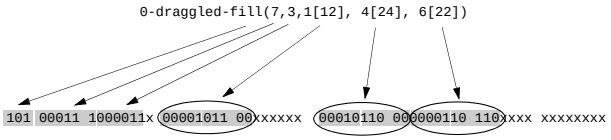


Figure 13. Binary representation of DFWAH from Figure 12

IV. EXPERIMENTS

As the size of a compressed bitmap is one of the most critical factors and we need to define the concrete memory layout for our draggled fill word.

As discussed before, we have some degree of freedom. First of all, we can choose between different pollution factors, we have to define how many bits we allocate to hold the length information or the information for the maximum numbers of pollutions inside a fill which both can restrict the maximum length of a draggled fill. The maximum distance of two pollutions in a fill (see Section III-C) also has a direct impact on the size of a draggled fill.

To obtain a feeling for appropriate settings of these values, we conducted a number of different experiments. In these experiments, we generated long bitmaps with different distributions.

Then, we took these bitmaps and ran our algorithm on it. In contrast to a real implementation, we did not have a maximum size of a fill, a maximum number of pollutions inside a fill or a maximum gap distance between polluted words and so on in this "ideal" implementation. We rather tried to find appropriate values for these parameters.

Input parameters for the bitmap generation are the density and different distributions (see below). The *density* is the fraction of "1" bit inside a bitmap and can be between zero and one. Our experiments focused on densities between 0.005 (0.5%) and 0.1 (10%), representing our medium-sparse bitmaps.

In our experiments we distinguish between different distributions:

- **Uniformly distributed bitmaps:** In a uniform distribution, every possible value (0/1) occurs all the time with the same probability, independently of previous values. In this case, the *density* is the same as the probability that a "1" value occurs. Such bitmaps can be generated easily using the standard random function.
- **Clustered bitmaps:** In a clustered distribution, set bits tend to occur in groups or clusters. Therefore, there is a higher probability, that more consecutive set bits occur than in uniformly distributed bitmaps. Clustered bitmaps typically reflect application data better than uniform distributions [6]. Bitmaps that follow a clustered distribution can be generated easily with a simple two-state *Markov chain*. Figure 14 shows such a finite state machine with probabilities of p and q for changing the state from the prior state. The *density* of such a distribution can be calculated by $d = p * (p + q)$ [7]. Additionally, the cluster factor f is determined by $f = 1/q$. Typically, the input for the generation of a Markov chain-generated bitmap is the density d and the clustering factor f. In this case, the probabilities p and q are calculated by $q = 1/f$ and $p = q * d/(1 - d)$.

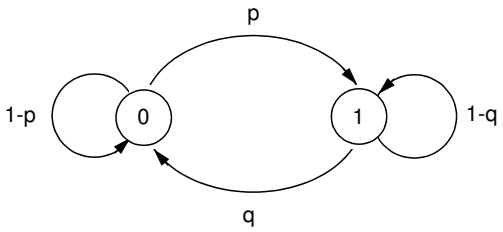


Figure 14. Two-state Markov-chain

A. Length of draggled fills

In a first series of experiments, we examined the potential length of the draggled fills. Figure 15 shows the distribution of draggled fill lengths for different densities between 0.5% and 10% and a pollution factor of 2. As expected, the length of a draggled fill strongly correlates with the density factor. Additionally, an approximate linear correlation between density and average/maximum fill length can be observed. For the interesting densities between 0.005 and 0.1, the average size of a fill is below 20. In Figure 16, the coverage for the different distributions in the previous figure is shown. So, for a density of 0.01, with a maximum fill length of 255 (8 bit memory consumption), 95% of all possible draggled fills discovered in our experiment are covered. Choosing 10 bits for the length field, a coverage of 99.999% of all possible fills can be achieved.

Figure 17 and Figure 18 show the same issue, but with a Markov distribution and a clustering factor of 2. Here, the distribution is more compact compared to the uniform distribution shown before in Figure 15.

Next, we examine a Markov chain-based distribution with higher clustering factors. A higher clustering factor leads to a higher possibility of literal words, but also longer runs of 0 bits appear. Figure 19 shows the distribution for different clustering factors for a distribution with a density 0.01. Obviously, the length increases approximately linearly with the clustering factor.

Resume: The longest runs result from uniformly distributed bitmaps with a low density. For the density range we are interested in, a maximum length between 255 (8 bits) and 1024 (10 bits) for a fill seems to be an appropriate value[12]. For the very small number of possible longer runs, a second run has to be started. For densities of about 0.05 (5%) and above, even a maximum length of 32 (5 bits) is ok.

[12]for a pollution factor of 2.

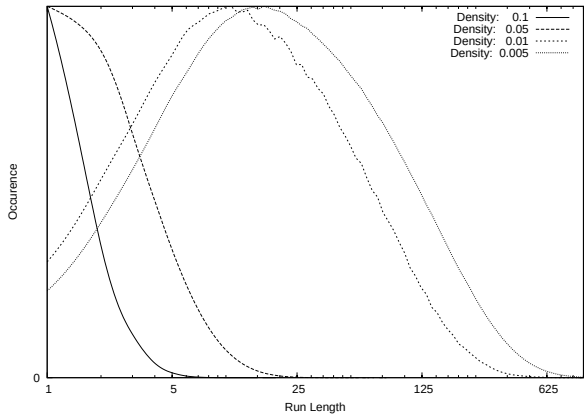


Figure 15. Distribution of draggled fill lengths with uniform distribution (pollution factor 2)

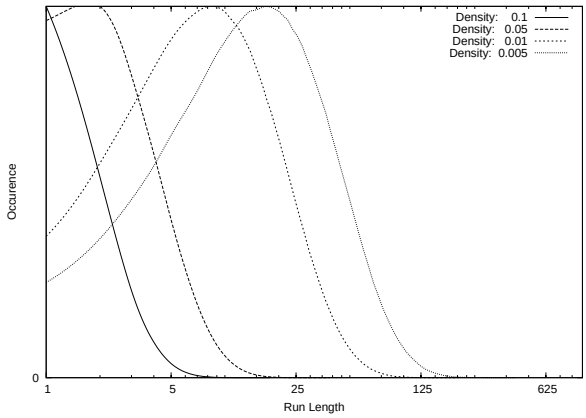


Figure 17. Distribution of draggled fill lengths with a Markov distribution, clustering factor 2 (pollution factor 2)

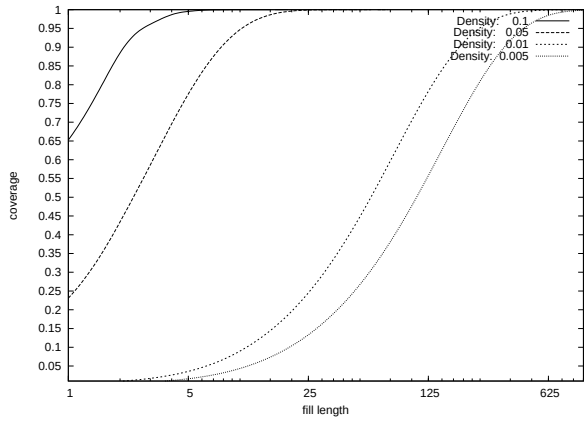


Figure 16. Coverage for different maximum fill lengths from Figure 15

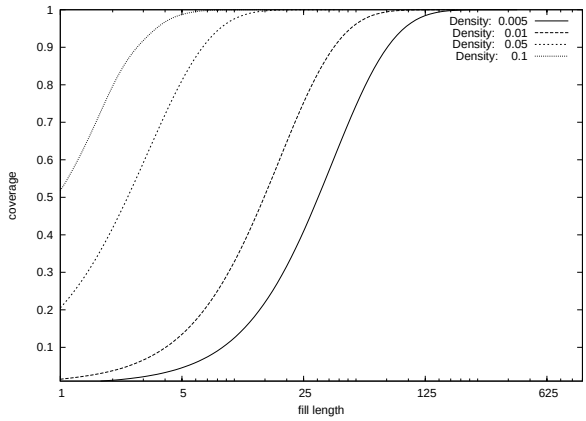


Figure 18. Coverage for different maximum fill lengths from Figure 17

B. *Influence of the pollution factor*

The next series of experiments are performed to obtain an idea of the influence of the different pollution factors. For a density of 5% (uniform distribution), Figure 20 shows the distribution of the length for different pollution factors. As expected, the run-length increases with the increase of the pollution factor. In all cases, the average number of pollutions inside a fill is about 74%-77%, irrespedictive of the pollution factor. In the case of 1% density (not shown), the number of pollutions inside a run is between 24% and 26% of the overall fill length. A comparable behaviour can be observed for the other densities of interest.

Figure 21 by contrast, shows the behaviour for the Markov-chain distribution. As in the previous case, the behaviours for different pollution factors are shown. Compared to the uniform distributions, the advantage of using a higher pollution factor is smaller, due to the characteristic of the distribution and the restriction of the positions of wrong bits inside a polluted word (see Section III-A). As a consequence, there is no advantage in using a pollution factor of 4, compared to a pollution factor of 1 or 2 - even more so as the handling is more CPU-intensive and the memory footprint of a polluted word corresponds to about a factor of 1.6 compared to a pollution factor of 2. The average number of pollutions inside a fill is between 45% and 47%. This lower factor is derived from the fact that in the Markov chain-based distribution used here the '1' bits are more clustered and longer runs of '0' bits appear. An increasing clustering factor for the Markov-chain distribution yields to longer runs compared to lower values (see also the previous experiment in Figure 19), but no further improvements for higher pollution factors.

Resume: Choosing a higher pollution factor yields to longer runs. Because of the restriction in the layout (see Figure 7), a number higher than 4 for the pollution factor does not seem to be meaningful. This is especially true for the more clustered behaviour of Markov chain-generated bitmaps. So the values determined in Section IV-A are still valid for the different pollution factors.

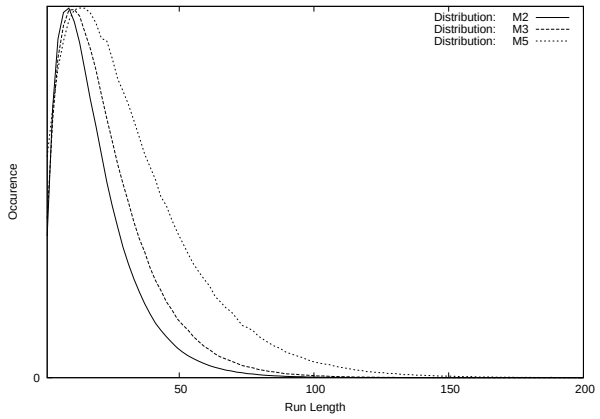


Figure 19. Markov-chain distribution: comparison of the influence of different clustering factors to the run-length (PF: 2, density: 0.01)

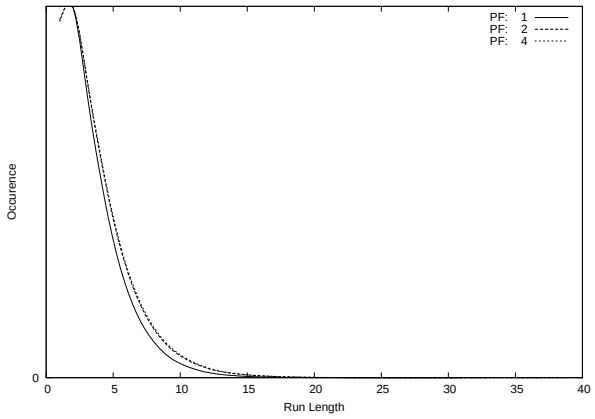


Figure 21. Distribution of run-length for different pollution factors (Markov-chain distribution, clustering factor 2, density 5%, pollution factor 2)

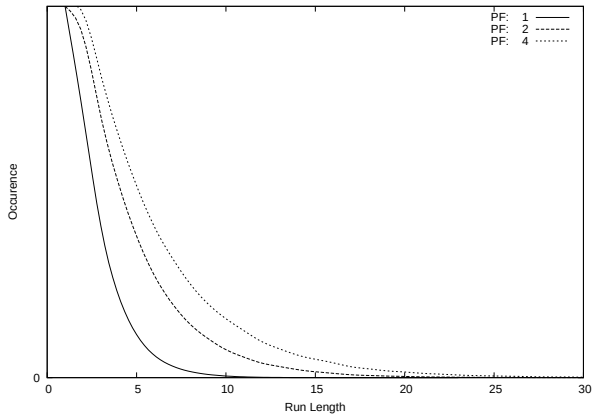


Figure 20. Distribution of run-length for different pollution factors (uniform distribution, density 5%)

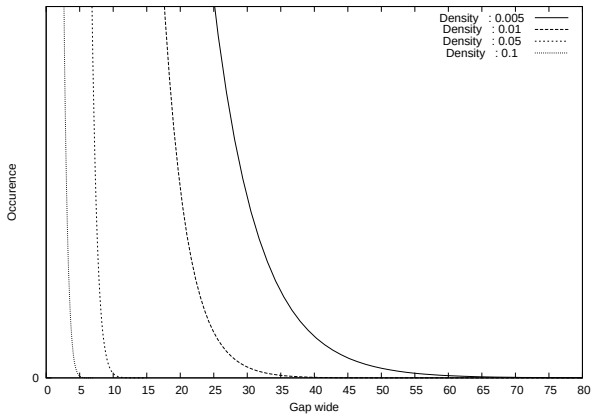


Figure 22. Distribution of gaps between pollutions in a draggled fill (uniform distribution, pollution factor 2)

C. Distance between pollutions in a draggled fill

As explained in Section III-C, the position of a pollution can not only be expressed by its position index inside the fill, but also by the distance to the previous pollution. This has the advantage, that with a growing size of a fill, the memory consumption to express the next pollution position is smaller.

Figure 22 shows the distribution of the gap width for different densities (uniform distribution, pollution factor 2). For the densities we are interested in, a maximum gap width of 16 (covering 95% of all gaps for a 0.01 density) or possibly 32 (covering 99.99%) is sufficient, which means that we need 5 or 6 bits memory consumption for the length field (the maximum gap width for Markov chain-distributed bitmaps is slightly lower).

Figure 23 shows the coverage for different gap lengths and different densities. So i.e., with a maximum gap length of 32 and a density of 0.005, a coverage of 99% can be achieved. For a density of 1%, this can already achieved with a maximum gap length of 16.

More experiments and also some analytical examinations concerning the size of the bitmaps can be found in [14].

V. 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 of WAH is unsatisfying. The reason for this behaviour is the blocking factor of 32, which requires packing of a minimum of 31 bits. Thus, even a single skipped bit leads to a full literal block, which holds 31 uncompressed bits.

Our contribution handles this problem by allowing so-called polluted words to be part of a fill. A polluted word is a block which has a limited number of wrong bits. The idea is to describe the position of the polluted words in the fill and the wrong bits inside it instead of storing the whole

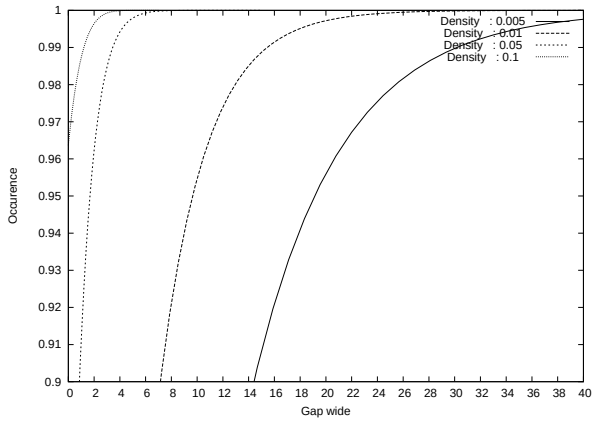


Figure 23. Coverage of gap width from the distribution in Figure 22

literal word, which takes less memory than ending a fill, starting a new literal block, and after that starting a new fill.

After presenting the concept of our improved algorithm, we ran a number of different experiments, to obtain a feeling of the behaviour of our algorithm for different distributions (uniform, Markov with different clustering factors), different pollution factors, and different implementation variants.

VI. Future Work

Currently, final implementation of our concept is not yet finished, but we already have a functional prototype without any optimisations. Our next step will be to integrate the algorithm in the WAH codebase. To do so, we will completely rewrite of our functional prototype. Once we have our implementation finished, we plan a number of tests with different values for selectivity, reflecting both synthetical and real world data in order to compare both the compression ratio and the execution time of the different operations. Depending on the results, we will eventually implement different variants of our algorithm, which we discussed in Section III-C.

References

[1] A. Schmidt and M. Beine, "A Concept for a Compression Scheme of Medium-Sparse Bitmaps," in DBKDA 2011, Proceedings of the Third International Conference on Advances in Databases, Knowledge, and Data Applications, St. Maarten, The Netherlands Antilles, 2011, pp. 192.–195.

[2] 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, 2000, pp. 231–246.

[3] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation. New York, NY, USA: ACM, 1999, pp. 13–24.

[4] 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.

[5] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," IEEE TRANSACTIONS ON INFORMATION THEORY, vol. 23, no. 3, 1977, pp. 337–343.

[6] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," ACM Trans. Database Syst., vol. 31, no. 1, 2006, pp. 1–38.

[7] 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.

[8] P. O'Neil and E. O'Neil, *Database: Principles, Programming, Performance*. San Francisco, CA: Morgan Kaufmann, 2001.

[9] J. W. Hector Garcia-Molina, Jeffrey D. Ullman, Database System Implementation. Prentice-Hall, 2000.

[10] J. Wu, "Annotated references on bitmap index." [Online]. Available: http://www-users.cs.umn.edu/ kewu/annotated.html, retrieved: december, 2011.

[11] 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.

[12] K. Wu, E. J. Otoo, and A. Shoshani, "A performance comparison of bitmap indexes," in CIKM '01: Proceedings of the tenth international conference on Information and knowledge management, ser. New York, NY, USA: ACM, 2001, pp. 559–561.

[13] S. W. Golomb, "Run-length encodings," IEEE-IT, vol. IT-12, 1966, pp. 399–401.

[14] M. Beine, "Implementation and Evaluation of an Efficient Compression Method for Medium-Sparse Bitmap Indexes," Bachelor Thesis, Department of Informatics and Business Information Systems, Karlsruhe University of Applied Sciences, Karlsruhe, Germany, 2011.