8-1-2016

Performance evaluation of word-aligned compression methods for bitmap indices

Gheorghi Guzun
University of Iowa, gheorghi.guzun@sjsu.edu

Guadalupe Canahuate
University of Iowa

Recommended Citation

Gheorghi Guzun and Guadalupe Canahuate. "Performance evaluation of word-aligned compression methods for bitmap indices" *Knowledge and Information Systems* (2016): 277-304. https://doi.org/10.1007/s10115-015-0877-9

Performance evaluation of word-aligned compression methods for bitmap indices

Gheorghi Guzun[1] and Guadalupe Canahuate[2]

[1,2]Department of Electrical and Computer Engineering, The University of Iowa, Iowa City, IA, USA;
[1]gheorghi-guzun@uiowa.edu; [2]guadalupe-canahuate@uiowa.edu

Bitmap indices are a widely used scheme for large read-only repositories in data warehouses and scientific databases. This binary representation allows the use of bit-wise operations for fast query processing and are typically compressed using run-length encoding techniques. Most bitmap compression techniques are aligned using a fixed encoding length (32- or 64-bits) to avoid explicit decompression during query time. They have been proposed to extend or enhance Word-aligned hybrid (WAH) compression.

This paper presents a comparative study of four bitmap compression techniques: WAH, PLWAH, CONCISE, and EWAH. Experiments are targeted to identify the conditions under which each method should be applied and quantify the overhead incurred during query processing. Performance in terms of compression ratio and query time is evaluated over synthetic-generated bitmap indices and results are validated over bitmap indices generated from real data sets. Different query optimisations are explored, query time estimation formulas are defined, and the conditions under which one method should be preferred over another are formalized.

Keywords: Bitmap Indices; Word-aligned compression; Data warehouses; Performance comparison and estimation

1. Introduction

From business analytics to scientific analysis, today's applications are increasingly data-driven, proliferating in data stores that are rapidly growing in density. To support efficient *ad hoc* queries, appropriate indexing mechanisms must be in place. One popular indexing technique for enabling fast query execution over large scale read-only data sets is the bitmap index. Because bitmap indices can leverage fast bit-wise operations supported by hardware, they have been extensively used for selection and aggregation queries in data warehouses and scientific applications.

A bitmap index is used to represent a property or attribute value-range. For a simple bitmap encoding, each bit in the bitmap vector corresponds to an object or record in a table and bit positions are set only

Received Dec 15, 2014
Revised Jul 20, 2015
Accepted Jul 27, 2015

for the objects that satisfy the bitmap property. For categorical attributes, one bitmap vector is created for each attribute value. Continuous attributes are discretized into a set of ranges (or bins) and bitmaps are generated for each bin.

For example, consider a bitmap index for a relation of different objects in a spatial grid. Let us consider three attributes for this table: the object type and the X and Y coordinates where the object resides. Attribute *Type* is a categorical attribute and one bitmap vector is created for each object type. Attributes *X* and *Y* are continuous attributes. The desired resolution for the grid defines how many bins need to be created for *X* and *Y*. A grid with resolution 100×100, would require 100 bitmap vectors for attribute *X* and 100 bitmap vectors for attribute *Y*. A query asking for objects located within the grid cell identified by x_i and y_j, can be answered by ANDing the corresponding bitmap vectors together. Any set bit in the resulting vector indicates an object located within the cell.

If the number of objects is fixed, increasing the resolution of the grid in the running example will increase the number of bitmap vectors producing a larger index but it will also increase the sparsity of the bit-vectors created. For this reason, bitmap indices are often compressed using *run-length* based encoding. A *run* refers to a set of consecutive bits with the same value, i.e., all 0s or all 1s. Sparse bitmap will have long runs of zeros.

Current bitmap compression techniques minimize the compression overhead in query time by allowing operations to work directly over the compressed bitmaps (Antoshenkov, 1995; Wu et al., 2002; Deliege and Pederson, 2010; Wu et al., 2001b). Most of them can be considered as extensions of Word Aligned Hybrid (WAH) encoding (Wu et al., 2001b, 2006). For a computer architecture of w =32-bit or 64-bit words, WAH uses segments of size $w - 1$ to align the bitmap vectors and compress them. A segment with heterogeneous bit values is called a *literal segment*, and a segment where all bits have the same value is referred to as a *fill segment*. Fills and literals are both encoded into words of w bits. The type of word, *literal* or *fill*, is indicated by the most significant bit. A literal word encodes one literal segment. A fill word encodes a number of consecutive fill segments. For fill words, the second most significant bit is used to indicate the homogeneous value of the fill and the remaining $w - 2$ are used to store the number of segments in the *run*. The maximum fill length that can be represented in a single fill word is $(2^{w-2} - 1)(w - 1)$. Clearly short runs could be represented with considerably less number of bits. This has been the motivation for improved schemes over WAH that make better use of the bits in the fill words to compress better (Deliege and Pederson, 2010; Colantonio and Di Pietro, 2010; Fusco et al., 2010) or to improve query execution time (Wu et al., 2002).

The problem when evaluating these techniques is that all of them have the same big-O complexity. However, experiments reported in different studies show considerable performance differences. In order to determine the scenarios under which one method should be preferred over another a more comprehensive study and performance evaluation over the same data-sets with varying dimensionality, cardinality, volume, and data distribution is needed. Also, both point and range queries, the most common supported types of queries should be evaluated over these datasets.

This paper presents a comparative performance study of four recently proposed word-aligned bitmap compression techniques: Word Aligned Hybrid (WAH), Position List WAH (PLWAH), Concise, and Word Bitmap Code (WBC) (aka Enhanced WAH (EWAH)). Estimations derived from experiments using synthetic datasets are validated using real datasets. This study is a necessary step towards building a bitmap compression framework that can estimate performance of the different techniques and is able to select the best compression method and/or guide users in the decision of which compression method to use based on compression ratio, query time, or a combination of both.

The main contributions of this paper can be summarized as follows:

– This is the first work that compares WAH, PLWAH, EWAH, and Concise together over the same datasets and evaluate their performance differences for varying controlled parameters.
– Similar and distinctive characteristics for the compared bitmap encoding schemes are identified and a comprehensive analysis of the encoding and querying algorithms is provided.
– A methodology for estimating the relative query time performance for the different bitmap encodings is

proposed. Our evaluation on both synthetic and real data sets shows that estimation accuracy is within a confidence level of 95% for the relative performance between WAH/PLWAH and WAH/EWAH.

The remainder of this paper is organized as follows: Section 2 presents related work on bitmap compression. Section 3 describes the four compression techniques evaluated in detail. Section 4 goes further into analysing the encoding and querying algorithms, and provides insights for performance estimations. Section 5 introduces the formulas for performance estimation. Section 6 presents our experimental results for a set of synthetic datasets, as well as four real datasets. Finally, Section 7 presents the conclusions of this study.

2. Related Work

Bitmap indices are typically compressed using specialized run-length encoding schemes that allow queries to be executed without requiring explicit decompression. Byte-aligned Bitmap Code (BBC) (Antoshenkov, 1995) was one of the first compression techniques designed for bitmap indices using 8-bits as the alignment length and four different types of words. BBC compresses the bitmaps compactly and query processing is CPU intensive. Word Aligned Hybrid (WAH) (Wu et al., 2002) proposes the use of words instead of bytes to match the computer architecture and make access to the bitmaps more CPU-friendly. WAH reduces the types of words to only two to further simplify query processing. The use of a shorter encoding length enables BBC to compress better than WAH, which in theory would require between 2.5-4 times more space than a two byte implementation of BBC. In practice, however, WAH typically only uses 60% more space than BBC and can execute logical operations 12 times faster (Wu et al., 2002).

Furthermore, reordering has been applied to improve compression and reduce query time of bitmap indices (Lemire et al., 2010; Pinar et al., 2005; Kaser et al., 2008; Apaydin et al., 2008; Lemire et al., 2012). Finding an optimal order, however, has been proven to be NP-Complete (Pinar and Heath, 1999) and different heuristics have been proposed such as lexicographic order, Gray codes (Pinar et al., 2005), and Hamming-distance-based (Malik and Kender, 2007), among others. These reordering algorithms produce longer runs for the first few bit vectors, but deteriorate into shorter runs (and worse, a random distribution) of bits for the higher vectors. Such pattern means that WAH can achieve optimal compression for the first few columns but almost none in the latter columns. This has motivated the use of different encoding lengths to compress the bitmaps: Partitioned WAH (PWAH) (van Schaik and de Moor, 2011) and Variable Length Coding (VLC) (Fabian Corrales and Sawin, 2011). These approaches do not enforce the alignment of the encoding length, which causes significant query performance overhead as the miss-alignments need to be resolved at query time.

Within the past several years many modifications and improvements for WAH have been proposed (Deliege and Pederson, 2010; Wu et al., 2001b; Colantonio and Di Pietro, 2010; Fusco et al., 2010; Wu et al., 2001a; Lemire et al., 2010). All of them try to improve compression ratios or query times. However, depending on bitmap densities and clustering factors, one encoding scheme may perform better or worse than WAH.

Two other works try to analyze and project the running time of different bitmap compression schemes (Wu et al., 2001b), (Wu et al., 2004). However these papers focus on WAH and BBC and do not include newer schemes such as PLWAH (Deliege and Pederson, 2010), COMPAX (Fusco et al., 2010), CONCISE (Colantonio and Di Pietro, 2010) or WBC/EWAH (Wu et al., 2001a; Lemire et al., 2010).

In (Guzun et al., 2014) authors propose a framework that uses different word/segment lengths for encodings and could potentially choose between different encodings depending on their performance over different datasets. In this paper we complement (Guzun et al., 2014) by analyzing the performance of four different bitmap encodings (Deliege and Pederson, 2010; Wu et al., 2001b; Colantonio and Di Pietro, 2010; Fusco et al., 2010). This comprehensive study evaluates the effect of different parameters in the compression ratio and query time of these methods using both synthetic and real data sets. Our goal is

[ht]

128 bits	1,20*0,4*1,78*0,30*1			
31-bit groups	1,20*0,4*1,6*0	62*0	10*0,21*1	9*1
groups in hex	400003C0	00000000 00000000	001FFFFF	000001FF
WAH (hex)	400003C0	80000002	001FFFFF	000001FF

Fig. 1. A WAH bit vector.

to formalize the conditions where one method should be preferred over the others in order to maximize overall performance.

3. Word-aligned Bitmap Compression

Most of the current bitmap compression techniques can be thought as variations of Word Aligned Hybrid (WAH) encoding (Wu et al., 2002). These use w-bit words typically matching the computer architecture (i.e. 32- or 64-bits) to divide the verbatim bitmap and compress it using a mix of literals and fills. WAH only has two types of words: literal words and fill words. Let w denote the number of bits in a word. Assuming the most significant bit indicates the type of word we are dealing with, the lower $(w-1)$ bits of a literal word contain the bit values from the bitmap. If the word is a fill, then the second most significant bit is the fill bit, and the remaining $(w-2)$ bits store the fill length. WAH imposes the word-alignment requirement on the fills. This requirement is key to ensure that logical operations only access words.

The differences between WAH and the other word-aligned methods proposed in the literature, reside mainly in the encoding of the fills and the decoding algorithm used during query execution. Some encodings provide better compression while others are targeted for faster query execution. However, with so many similarities between these techniques, sometimes it is not easy to decide which method to apply. We aim to identify the trade-offs of each method so each encoding can be applied in the cases where it can improve performance the most. The rest of this section describes different word-aligned bitmap compression techniques evaluated in this study with respect to their encoding (compressed size) and query processing (execution time).

As a reference, the terms and notations used further in this paper are defined in Table 1.

3.1. Encoding

Figure 1 shows a WAH bit vector representing 128 bits. In this example, we assume 32 bit words. Under this assumption, each literal word stores 31 bits from the bitmap, and each fill word represents a multiple of 31 bits. The second line in Figure 1 shows how the bitmap is divided into 31-bit groups, and the third line shows the hexadecimal representation of the groups.

```
class bitVector {
  long[] vec; //The words in the bit-vector
  activeWord[] activeWords; //The decoded word(s)
  activeWord active; //The active word
  class activeWord {
    long value;  //The literal value
    long nSegments; //Number of segments in fill
    long fill; //Fill value (allOnes or allZeros)
    boolean isFill; //A flag for the type of word
  }
}
```

The last line shows the values of WAH words. Since the first and third groups do not contain more than 31 single bit values, they are represented as literal words (a 0 followed by the actual bit representation of the group). The fourth group is less than 31 bits and thus is stored as a literal. The second group contains a

Notation	Description
w	Word length used to encode literals and fills
M	Number of words in a bitmap bit-vector
N	Number of bits in a verbatim bitmap
s	Number of bits PLWAH position list can store
p	Number of bits to index PLWAH position
d	The bit density: the fraction of set bits
f	Clustering factor
c	Cardinality of an attribute
m_x	Number of words in the compressed bit-vector X
$C_a(m)$	Time to allocate memory for m
C_1	Time to decode a word
D_l	Time to decode a literal word
D_f	Time to decode a fill word
C_l	Time to append a literal word
C_f	Time to append a fill word
I_w	Number of iterations in the query algorithm
I_l	Number of iterations that append literals
I_f	Number of iterations that append fills
α	Fraction of iterations that generate appendLiterals
t	Time to perform a bitwise logical operation

Table 1. Notation Reference

multiple of 31 0's and therefore is represented as a fill word (a 1, followed by the 0 fill value, followed by the number of fills in the last 30 bits). The first three words are regular words, two literal words and one fill word. The fill word 80000002 indicates a 0-fill that is two-word long (containing 62 consecutive zero bits). The fourth word is the last word, it stores the last few bits that could not be stored in a regular word. Another word with the value nine, not shown, is needed to store the number of useful bits in the last word. Logical operations are performed over the compressed bitmaps resulting in another compressed bitmap.

Listing 1 provides the description of a suitable data structure for representing a word-aligned compressed bit-vector. The bit-vector contains an array of longs that stores the whole encoded bitmap and an activeWord structure that holds the decoded long for query processing purposes.

3.2. Query Processing

Bitmap indices efficiently support point, range, and aggregate queries. For this study we focus on point and range queries. Authors in (Guzun et al., 2014) show that the query algorithms for word-aligned bitmap compressions are very similar and that they can be unified under a single framework. Algorithm 1 implements a generic query algorithm for a logical operation between two bit-vectors encoded with word aligned bitmap compressions. Two word-aligned bit vectors x_w and y_w encoded using word lengths w, are operated together to produce bit vector z_w, which stores the result of the bit-wise logical operation $x_w \circ y_w$, where $\circ$ is a binary logical operator. Algorithm 1 shows the pseudocode for query processing. Each bitmap is still decoded one physical word (*activeWord*) at a time (Line 1-8). The *activeWords* are operated together until exhausted. Two fill words can be operated without explicit decompression (Lines 16-26). If one of the *activeBlocks* is a literal, then the values are operated together and the number of segments in the fill, *nSegments*, is decremented by 1 with each *getLitValue()* call (Lines 27-30). The literals and fills are appended to the resulting bit-vector z which is initialized to the uncompressed size of M words.

We have identified three alternatives for the encoding of the resulting bit-vector:

1. Decompress one of the bit-vectors and perform the bit-wise operation in-place, overwriting the verbatim vector.
2. Have a partially compressed result appending the words to the resulting vector without further encoding, i.e. the fills and literals are appended to the vector but no merging is done with the previous words.
3. Fully compress the result.

We evaluate the performance of these alternative in the experimental results.

In the next subsections we describe several different word-aligned bitmap encoding schemes and emphasize their commonalities and particularities. For this purpose we use the example in Figure 2 to show a a) non-compressed bitmap bit-vector and its encodings using b) WAH, c) PLWAH, d) EWAH, and e) CONCISE.

3.3. Word-aligned Hybrid (WAH) encoding

The bitmap compression schemes that we analyse in this work are derived from WAH (Wu et al., 2002). Some try to improve its compression, while some try to improve on its query execution time. Thus we first introduce the WAH bitmap compression scheme in this section.

3.3.1. Encoding

Figure 2 shows an example of a bitmap B compressed using four different approaches. For this example, $w = 32$. The verbatim bitmap B is first divided into segments of size $s = 31$ bits. Figure 2a) shows the uncompressed bitmap vector with $1,984$ bits divided as a, 61×31 bits run of 0s, 31 bits with a single 1 in position 13 and another run of 0s with 2×31 bits.

Figure 2b) shows bitmap B compressed using WAH encoding with word length $w = 32$. For each word, the first bit (in bold) is the flag bit (1 for fills and 0 for literals). For fill words, the second most significant bit (underlined) is the fill bit and indicates the run symbol. The rest of the bits in the word (30 bits) are used to encode the number of fill segments in the verbatim bitmap (61 for the first fill). A literal word (second one) stores a 31-bit literal segment.

3.3.2. Query Execution

Now let us consider WAH query processing as presented in Algorithm 1. The discussion that follows consider a query executed as the AND of two compressed bitmap vectors.

A bit-wise operation is performed by iterating over the words in the bitmap vectors, similarly to algorithm 1. The current word is decoded into an active structure that identifies the type of word (fill or literal) and the number of segments encoded in the word (Algorithm 2). For literal words it holds the literal value and the number of segments is set to 1; for fill words it holds the fill bit value and the number of segments encoded in the fill. Decoded active words are queried together until the number of segments is exhausted. There are three cases when executing the query. If the two active words are fills, the result

Algorithm 1: General Bit-wise Logical Operation

```
Input: bit-vector $x$, $y$ - Input bit-vectors
Output: bit-vector $z$ - the resulting compressed bit-vector after performing the logical operation $x \circ y$
bitVector $z$=new bitVector($M$);
while $x.vec$ and $y.vec$ are not exhausted do
    if $x.activeWord$ is exhausted and there are more words in $x$ then
        $x.decodeWord()$;
    end
    if $y.activeWord$ is exhausted and there are more words in $y$ then
        $y.decodeWord()$;
    end
    while $x.activeWords$ and $y$.activeWords are not exhausted do
        if $x.activeWord.nSegments == 0$ then
            $x.activeWord = x.nextWord()$;
        end
        if $y.activeWord.nSegments == 0$ then
            $y.activeWord = y.nextWord()$ ;
        end
        if $x.activeWord.isFill$ and $y.activeWord.isFill$ then
            $nSegments$=Min($x.activeWord.nSegments$, $y.activeWord.nSegments$);
            if $x.activeWord.fill \circ y.activeWord.fill == 0$ then
                $z.vec$.add($FillOfZeros + nSegments$);
            end
            else
                $z.vec$.add($FillOfOnes + nSegments$);
            end
            $x.activeWord.nSegments$-=$nSegments$;
            $y.activeWord.nSegments$-=$nSegments$;
        end
        else
            $z.vec$.add($x.activeWord$.getLitValue() $\circ$ $y.activeWord$.getLitValue());
            //GetLitValue decreases nSegments by 1
        end
    end
end
return $z$;
```

Algorithm 2: WAHdecode

```
Input: $value$
Output: $aw$
activeWord aw = new activeWord($value$);
if $value > AllOnes$ then
    aw.fill = (value $\geq$ FillOfZeros ? AllOnes : 0);
    aw.nSegments = value & FillMask ;
    aw.isFill = 1;
end
else
    aw.nSegments = 1; aw.isFill = 0;
end
```

is another fill word with the fill bit value equal to the result of the two fill bit values ANDed together, and the minimum number of segments between the two as the number of segments. If both active words are literals, then the result is a literal word with literal value equal to the two literals ANDed together. Finally, if there is one literal and one fill, the number of segments in the fill word is decreased by one and the result is a literal word with the result of the literal value ANDed with a literal representation of the fill.

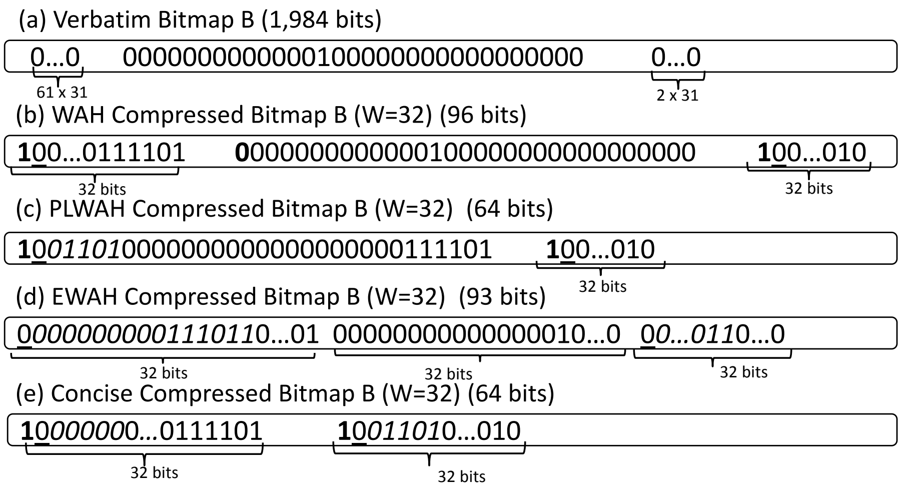


Fig. 2. Example of a) a verbatim bitmap B (1,984 bits) encoded using b) WAH (96 bits), c) PLWAH (64 bits), d) EWAH (93 bits) and e) CONCISE (64 bits).

The result of the logical operation is then appended to the bit-vector representing the final result of the two bit-vectors being operated (Algorithm 3). As can be seen, bitmaps are not explicitly decoded. At no given time is the bitmap fully decoded. Only one word for each bitmap is accessed and decoded at a time.

3.4. Position-list WAH (PLWAH) and CONCISE

PLWAH (Deliege and Pederson, 2010) and CONCISE (Colantonio and Di Pietro, 2010) exploit the fact that not all the bits in a word are used to store the run-length counter when compressing a fill word. Often runs are interrupted by a single set/unset bit, thus they use some of the bits in the fill word to store the position of the set/unset bit. This enables a better compression when compared to WAH.

3.4.1. Encoding

Figure 2c) shows the running bitmap example encoded using PLWAH. PLWAH is able to compress the literal word that follows the first fill as it has only 1 bit set. The five bits shown in *italics* (bits 2-7) have their value set to $(13)_{10} = (1101)_2$, to indicate that the thirteen'th position is set in the literal word following the fill. This literal is not explicitly stored in the compressed bitmap. If the position bits for a fill word are all zeros then no literal word is added during decoding. PLWAH uses $p = \lceil \log_2 w \rceil$ bits to index the position. Where w is the word length. Therefore the maximum run-length that can be encoded with a single fill word is $(2^{w-p-1}s) - 1$. Where s is the maximum number of heterogeneous bits a fill word can store in its list of positions. Longer fills will require two fill words to be encoded. Concise (2e)) is very similar to PLWAH, with the difference that the near-fill literal would be encoded together with the next following fill instead the foregoing fill to the literal. As shown in the figure this does not affect significantly the size of a compressed bitmap and these two techniques shall exhibit similar performance metrics.

Algorithm 3: WAHappendLiteral

```
Input: $w - 1$ literal value in $activeValue$
Output: $vec$ extented by $w - 1$ bits
if $vec.isEmpty()$ then
    $vec$.add ($activeValue$);
end
else if $activeValue$==0 then
    if $vec.back()$==0 then
        $vec$.back()=TwoFillZeros;
    end
    else if ($vec.back() \geq$ FillOfZeros AND $vec.back() <$ FillOfOnes) then
        $vec$.back()++;
    end
    else
        $vec$.add($activeValue$);
    end
end
else if $activeValue$==AllOnes then
    if $vec.back()$== $activeValue$ then
        $vec$.back()=TwoFillZeros;
    end
    else if ($vec.back() \geq$ FillOfZeros) then
        $vec$.back()++;
    end
    else
        $vec$.add($activeValue$);
    end
end
else
    $vec$.add($activeValue$);
end
```

3.4.2. Query Execution

Bit-wise logical operations take most of the time when executing queries over bitmap indices. The same Algorithm 1 can be used to perform logical bit-wise operations over PLWAH compressed bitmaps. The differences between WAH and PLWAH query execution can be limited to the complexity of decoding and encoding algorithms. From Algorithm 1 it is obvious that the number of times the query algorithm iterates is directly proportional to the compressed size of bit-vectors x and y. In fact, it iterates the same number of times for both, WAH and PLWAH, when both use the same hybrid encoding scheme and the same word size. The only case where the number of iterations will differ between the two, is when there are runs longer than $(2^{w-p-1}s) - 1$. These will create additional fill words within PLWAH and will consequently lead to slightly more iterations for PLWAH. The advantage of PLWAH is that it can encode a fill and a sparse literal within the same word. This translates into calling fewer times Algorithm 5 than WAH would call Algorithm 2. On the other hand, the $decode$ and $append$ algorithms for PLWAH are more complex than the ones for WAH. In particular, Algorithm 5, and Algorithm 7 can take more time than their WAH equivalents. Algorithm 5 can take a longer execution time than Algorithm 2 because it has to check if a fill word encodes a sparse literal or not. Algorithm 7 can take a longer running time than 3 because it has to check if a literal is sparse, and whether it can be encoded with the previous fill or not. Algorithms 6 and 4 are almost identical, the only diference being in the values of $FillOfOnes$ and $FillOfZeros$. These values are different for PLWAH because of the position bits.

Ultimately, Algorithm 1 will perform an equal number of iterations for both WAH and PLWAH, however the query time difference between the two depends on how many times PLWAH invokes Algorithm 5

Algorithm 4: WAHappendFill

```
Input: $fillBit$ and $nSegments > 0$
Output: $vec$ extented by $w - 1 \times nSegments$ bits
if $nSegments > 1$ AND NOT vec.isEmpty() then
    if fillBit==0 then
        if (vec.back() $\geq$ FillOfZeros AND vec.back() $<$ FillOfOnes) then
            $vec$.back()+= $nSegments$;
        end
        else
            $vec$.add($FillOfZeros$ + $nSegments$);
        end
    end
    else if vec.back() $\geq$ FillOfOnes then
        $vec$.back()+= $nSegments$;
    end
    else
        $vec$.add($FillOfOnes$ + $nSegments$);
    end
end
else if vec.isEmpty() then
    if fillBit==0 then
        $vec$.add($FillOfZeros$ + $nSegments$
    end
    else
        $vec$.add($FillOfOnes$ + $nSegments$);
    end
end
else
    activeValue = (fillBit ? AllOnes : 0);
    $vec$.add($activeValue$);
end
```

and how many literals are appended to the resulting bit-vector. We further analyse the relative performance of WAH and PLWAH in section 4.

3.5. Word Bitmap Code (WBC) a.k.a. Enhanced WAH (EWAH)

WBC (EWAH) (Wu et al., 2001a; Lemire et al., 2010) is in a way related to BBC (Antoshenkov, 1995). Similarly to BBC, WBC divides a bitmap into runs of fills followed by a tail of literal words.

3.5.1. Encoding

The runs of fills are encoded into marker-words, that store the number of clean words (runs of 0's or 1's) in the leftmost half bits, and the number of following literals in the remaining bits. An EWAH bit-vector always starts with a marker-word. Figure 2 d) shows the verbatim bitmap from Figure 2 a) encoded with EWAH. The first half of a marker-word represents the header. In the case of a 32-bit word length, the first 16 bits are the header. The first bit of the header tells the type of encoded clean words, the next 15 bits represent the number of encoded clean words. The second half of the a marker-word, tells the number of "dirty words" (literals) that follow a marker-word. The literals have the same length as the word length, in our case 32 bits. In most cases, EWAH compression ratio is close to WAH compression ratio given the similarities in encoding and almost the same length of encoded literal words. However, when runs are longer than $(32 \times 2^{15}) - 1$ EWAH needs more than one marker word to encode the run. Thus, as showed

Algorithm 5: PLWAHdecode

```
Input: $value$
Output: $aw$
p=25 if w=32, 54 if w=64 ;
activeWord aw = new activeWord(value);
if $value > AllOnes$ then
    aw.fill = (value $\geq$ FillOfZeros ? AllOnes : 0);
    aw.nSegments = value & FillMask;
    aw.isFill = true;
    $isSparse$ =($value$ & $position_mask$)!=0;
    aw.nwords = ($isSparse$)?2:1;
    $sparse$ = bitmap constructed from the position list
end
else
    aw.nwords = 1;
    aw.nSegments = 1;
    aw.isFill = 0;
end
```

in (Guzun et al., 2014), EWAH compression is less efficient for sorted bitmaps or very sparse non-sorted bitmaps.

3.5.2. Query Execution

EWAH requires a slight modification from the generic query algorithm presented in Algorithm 1. Given that EWAH stores the number of following consecutive literals, there is no need for decoding those words, and thus they can be accessed directly and treated as literals. This means that the decode algorithm is called fewer times and in some cases can make the queries run faster. On the other hand, as shown earlier, the query time is directly proportional with the size of both compressed bit-vectors being queried. For sparse bitmaps, EWAH compressed bitmaps can have larger sizes than WAH or PLWAH, and this may affect query time as well.

Note that the decode and appendFill algorithms for EWAH are very similar to the ones for WAH (Algorithm 2, 4). Appending a literal for EWAH requires usually more time since EWAH has to increment the number of dirty words following the clean words. Algorithm 8 shows the appendLiteral algorithm for EWAH. Algorithm 9 is called by Algorithm 8 to add a fill word. These algorithms (8 and 9) are used in the implementation of EWAH provided by (Lemire et al., 2010).

The **rlw**, in Algorithm 8 is the current (last) running length word. The addEmptyWord method is shown in Algorithm 9.

4. Performance Analysis

In this section we discuss and analyze the encoding and query algorithms for WAH, PLWAH/CONCISE and EWAH. We show the theoretical base for estimating sizes of compressed bitmaps and their query times.

4.1. Word-aligned Hybrid (WAH) encoding

Having already described the basic principles of the WAH bitmap encoding in section 3.3, we proceed with analyzing the probable bitmap sizes and query algorithm complexity of WAH.

Algorithm 6: PLWAH AppendFill

```
Input: $fillBit$ and $nSegments > 0$
Output: $vec$
if $nSegments > 1$ AND NOT vec.isEmpty() then
    if fillBit==0 then
        if (vec.back() $\geq$ FillOfZeros AND vec.back() $<$ FillOfOnes) then
            $vec$.back()+= $nSegments$;
        end
        else
            $vec$.add($FillOfZeros$ + $nSegments$);
        end
    end
    else if vec.back() $\geq$ FillOfOnes then
        $vec$.back()+= $nSegments$;
    end
    else
        $vec$.add($FillOfOnes$ + $nSegments$);
    end
end
else if vec.isEmpty() then
    if fillBit==0 then
        $vec$.add($FillOfZeros$ + $nSegments$
    end
    else
        $vec$.add($FillOfOnes$ + $nSegments$);
    end
end
else
    activeValue = (fillBit ? AllOnes : 0);
    $vec$.add($activeValue$);
end
```

4.1.1. *Size of WAH Compressed Bitmaps*

Let d be the bit density of a uniform random bitmap. The probability of finding an uninterrupted group that is a 1-fill, i.e., $2w-2$ consecutive bits that are one, is d^{2w-2}. Where w is the word length. Similarly, the probability of finding a counting group that is a 0-fill is $(1-d)^{2w-2}$. Thus, the expected size of a WAH compressed bit-vector containing N bits is:

$$m_{\text{WAH}}(d) = \frac{N}{w-1}(1-(1-d)^{2w-2} - d^{2w-2}).$$

For an attribute following a probability distribution, where the i^{th} value has a probability of p_i, the total size of the bitmap index compressed using WAH is (Wu et al., 2002):

$$s_N = \sum_{i=1}^{c} m_R(p_i) \approx \frac{N}{w-1}\big(c - \sum_{i=1}^{c}(1-p_i)^{2w-2} - \sum_{i=1}^{c} p_i^{2w-2}\big),$$

where N is the number of objects or bits in the bitmap, m_R is the expected number of words in a random bitmap, w is the word size, and c is the cardinality of the attribute.

WAH-encoded bitmaps never require more than $4N$ words. If $c < 0.01N$, then the maximum size of the WAH compressed bitmap for the attribute is about $2N$ words. A clustering factor f is computed as the average number of bits in the 1-fill runs. The maximum size of the WAH compressed bitmap for a

Algorithm 7: PLWAH AppendLiteral

```
Input: $vec$ and $literal$
Output: $vec$
last is the last word of $vec$;
if $vec.isEmpty()$ then
    $vec$.add ($activeValue$);
end
else if $literal = 0$ then
    appendFill(0,1) ;
end
else if $last$ is counter and has empty position list then
    if $last$ is a zero fill then
        if $literal$ is sparse then
            Generate position list for a zero fill;
            Place position list into $last$ ;
        end
        else
            Append $literal$ to $vec$ ;
        end
    end
    else if $literal$ is sparse then
        Generate position list for a one fill;
        Place position list into $last$ ;
    end
    else
        Append $literal$ to $vec$ ;
    end
end
else
    Append $literal$ to $vec$ ;
end
```

clustering factor $f > 1$ and $c < 0.1N$ is:

$$s \approx \frac{N}{w-1}\left(1 + \frac{2w-3}{f}\right),$$

which is nearly inversely proportional to the clustering factor f.

A more detailed reasoning and proofs for the above formulas can be found in (Wu et al., 2002).

4.1.2. Query Time Analysis

Let us denote by m_x and m_y, the number of words in the WAH compressed bitmap vectors X and Y, respectively. M is the number of words in the uncompressed bitmap ($M = \lceil \frac{N}{w-1} \rceil + 1$). In the generic algorithm 1, the main loop is executed $max(m_x, m_y) \leq I_w \leq min(m_x + m_y - 1, M)$ times. The *decode* method is called once per word: $m_x + m_y$ times. The *addLiteral* method is called when two literal words or a fill and a literal are operated together: αI_w, where α is the fraction of iterations that generate literals. The *addFill* method is called when two fills are operated together: $(1-\alpha)I_w$ times.

The size of the resulting bitmap is $m_z < min(m_x + m_y, M)$. This space is allocated before entering the loop to avoid dynamic memory allocation. Let us assume that the time to allocate m_z is $C_a(m_x + m_y)$. C_1 is the time to decode a word, C_l the time to execute appendLiteral, C_f the time to execute appendFill, and C_0 is the loop overhead. The time to execute a query using Algorithm 1 is:

$$t = C_a(m_x + m_y) + C_1(m_x + m_y) + C_l \alpha I_w + C_f(1-\alpha)I_w + C_0 I_w \tag{1}$$

Because I_w is always smaller than $m_x + m_y$, it is easy to see from Equation 1 that Algorithm 1 has a

Algorithm 8: EWAHappendLiteral

```
Input: $newdata$
Output: $vec$
if $newdata$ = 0 then
    addEmptyWord(false);
end
else if $newdata$=fillOfOnes then
    addEmptyWord(true);
end
else
    $numbersofar$ = getNumberOfLiteralWords() ;
    if $numbersofar > $ RunningLengthWord.largestliteralcount then
        pushBack(0);
        rlw.position = actualsizeinwords - 1;
        rlw.setNumberOfLiteralWords(1);
        pushBack($newdata$);
    end
    rlw.setNumberOfLiteralWords($numbersofar$ + 1);
    pushBack($newdata$);
end
```

Algorithm 9: addEmptyWord

```
Input: $fillBit$
Output: $vec$
$noliteralword$ = (rlw.getNumberOfLiteralWords() == 0);
$runlen$ = this.rlw.getRunningLength();
if ($noliteralword$) & ($runlen$ == 0) then
    rlw.setRunningBit($fillBit$);
end
if ($noliteralword$) & (rlw.getRunningBit() == $fillBit$) & ($runlen <$ RunningLengthWord.largestrunninglengthcount)
then
    rlw.setRunningLength($runlen$ + 1);
end
pushBack(0);
rlw.position = actualsizeinwords - 1;
rlw.setRunningBit($fillBit$);
rlw.setRunningLength(1);
```

complexity of $\mathcal{O}(m_x + m_y)$, which means that the query time is directly proportional with the size of the bitmap index. In order to estimate the query time however, this information is insufficient. A more precise approximation of I_w is required. Further in the next sections we show that I_w can be approximated with high accuracy, and so can the query time.

4.2. PLWAH and CONCISE

PLWAH and CONCISE exploit the fact that not all the bits in a word are used to store the run-length counter when compressing a fill word. Often runs are interrupted by a single set/unset bit, thus they use some of the bits in the fill word to store a position list of the set/unset bit. This enables a better compression when compared to WAH.

4.2.1. Size of PLWAH/CONCISE Compressed Bitmaps

Let s be the maximum number of heterogeneous bits a fill word can store in its list of positions. The size of the list is $s\log_2(w)$ and the size of the counter is $w - 2 - s\log_2(w)$. A single fill word can thus represent up to $(2^{w-2-s\log_2(w)})$ groups of length $(w - 1)$. All compression schemes have an overhead when representing incompressible bitmaps. For WAH and PLWAH, this overhead is one bit per word, so the compression ratio is $w/(w - 1)$. A bitmap containing no homogeneous groups will not have any fill words and will be incompressible. The upper and lower bounds of the PLWAH compression ratio are respectively: $(2^{w-2-s\log_2(w)}+1)(w-1)/w$ and $(w-1)/w$. As long as the upper bound is not reached, the worst PLWAH compression ratio is bounded by the WAH compression ratio. A more detailed description of PLWAH compressed bitmaps is provided in (Deliege and Pederson, 2010).

We estimate that the number of words in the PLWAH compressed bit-vector containing N bits, and a density d is:

$$m_{\mathrm{PLWAH}}(d) = \frac{N}{w-1}(1 - (1-d)^{2w-2} - d^{2w-2}) - S_{\mathrm{PLWAH}},$$

where S_{PLWAH} is the number of sparse words that can be encoded within the PLWAH fills. The statistics about this variable and the other used as inputs for size estimations have to be collected when scanning the data.

4.2.2. Query Time Analysis

As discussed in Section 3.4.2, PLWAH uses the same generic Algorithm 1 as WAH for performing bit-wise logical operations. Thus a big-O analysis of WAH querying algorithms and PLWAH querying algorithms would lead us to equivalent upper bounds. Thus, performing performance measurements for each of the constants in Equation 1 could be a better way to differentiate between the two encodings. In the next section, we perform an exercise of measuring the performance for the constants defined in Equation 1, and then show that it is possible to predict the relative performance between PLWAH/CONCISE and WAH for different data sets.

4.3. WBC (EWAH)

The WAH and EWAH bitmap encodings have often a similar compression ratio, however for some data distributions the difference in compressed size can be significant between WAH and EWAH. This is also reflected in the EWAH query time.

4.3.1. Size of EWAH Compressed Bitmaps

In general, WAH and WBC (EWAH) encoding schemes are fairly similar. The difference between WBC (EWAH) and WAH is that the first one uses only half of the fill word to encode the 0-fill or the 1-fill, while the second half is used to mark the number of literals following the fill word. Because of this, EWAH may need more than one word to encode a fill, if the bitmaps are very sparse. Thus, when estimating EWAH compressed size of a bitmap, one should consider counting the number of run-lengths longer than EWAH can encode, and adjust the WAH size estimation equations for EWAH. Thereby, the expected size of a EWAH compressed bit-vector containing N bits with density d is:

$$m_R(d) \approx \frac{N}{w}(1 - (1-d)^{2w} - d^{2w}) + R_{ex},$$

where R_{ex} is the number of runs that have a length greater than $(w \times 2^{\frac{w}{2}}) - 1$.

4.3.2. Query Time Analysis

WBC/EWAH compression scheme was designed with the goal of avoiding decoding literal words at query execution time. This reduces the number of memory accesses and can result in faster queries when compared to WAH. However at the same time, when compressing the resulting bit-vector EWAH has to increment the last counter of the following literal words. Thus every time EWAH performs a literal append, it has to access the last marker-word to increment its literal counter, and also decrement the counters of the bitmap columns queried.

5. Performance Estimations

As discussed in the previous sections, there is no best bitmap compression technique that can outperform the rest, for all types of data in both, query time and index size. Furthermore, since the query algorithms for all analysed techniques have the same upper bound complexity, it is difficult to estimate which one will perform better just by considering index sizes and the query big-O analysis.

In this section we estimate query times before actually running the queries for data sets compressed with different bitmap encoding schemes. Generally we want to estimate query times before even compressing a data set. Estimating both the size of the compressed data and the query time can help deciding in choosing the most suitable bitmap index compression technique for the data.

Our goal is to formalise a method to estimate the performance of these different encoding techniques in order to select the best bitmap compression encoding for various data sets. As users have different preferences, "best" could have different meanings, thus it is important to estimate both, the compression size and expected query time. Then it is at the user's discretion to choose between better compression ratio or faster query time. We show that it is possible to estimate within a confidence level of 95% the relative performance between four word-aligned based bitmap compression schemes: WAH, EWAH and PLWAH/CONCISE.

First we compute the size of compressed bitmaps using the procedures from the previous section. As discussed in previous sections, the query time is proportional to the size of the two compressed bitmaps being queried, however the complexity of decoding and appending has a big impact too. Algorithm 1 traverses through bit-vector x and bit-vector y until both of them are exhausted. Being a generic algorithm, Algorithm 1 is applicable for all, WAH, EWAH and PLWAH. Thus Equation 1 from Section 4 is valid for these compression schemes as well. C_a, C_1, C_l and C_f are all constants and can be measured empirically. Analysing the decoding algorithms for WAH (Algorithm 2) and PLWAH (Algorithm 5), it is clear that decoding of a literal word requires less instructions than decoding a fill word. Thus, we split the decoding cost, C_1, into two components: D_l for decoding a literal word, and D_f for decoding a fill word. Therefore, the general query time for Algorithm 1 becomes:

$$t_G = C_a(m_x + m_y) + D_l(m_{x_l} + m_{y_l}) + D_f(m_{x_f} + m_{y_f}) + C_l \alpha I_w + C_f(1_\alpha)I_w + C_0 I_w, \qquad (2)$$

where m_{x_l} and m_{y_l} represent the number of literals contained in the compressed bit-vector x and y respectively. Similarly, m_{x_f} and m_{yf} are the number of fills in x and y. Methodologies for computing the number of literals and fills in a compressed bit-vector have been discussed in Section 4.

Table 2 shows measured values of the constants from Equation 2 for WAH, PLWAH/CONCISE and EWAH. The constants are measured in time units that are relative to the machine the queries are run on. The relative differences between them are expected to be preserved on a faster/slower computer. From Table 2 we can observe that C_l and C_f are not equal, and thus the number of fill appends and literal appends have to be estimated. Another unknown remains the number of times Algorithm 1 executes the main loop, which is equal to the total number of appends ($appendLiterals$ + $appendFills$). The number of iterations for the main loop is completely dependent on the sizes of the queried bit-vectors. More specifically, it depends on the size of the largest compressed bit-vector and the size ratio between the two bit-vectors. For two independent and highly compressed bit-vectors, there is small probability that

Table 2. Measured Constants for query execution

	WAH	EWAH	PLWAH/ CONCISE
Memory alloc(C_a)	x	x	x
Decode Literal(D_l)	10x	10x	16x
Decode Fill (D_f)	20x	20x	22.5x
Append Literal (C_l)	5x	7.5x	7.5x
Append Fill (C_f)	5x	5x	5x

the fill words of one are aligned with the other bit-vector. Thus Algorithm 1 requires two iterations to exhaust one word from that largest bit-vector. On the other hand, for a bit-vector that is not compressed at all, only one iteration is sufficient to consume one word from the largest bit-vector. At the same time, the size ratio between the two bit-vectors being queried determines the final number of iterations. If the better compressed bit-vector is significantly smaller than the other bit-vector, then the probability for two fill words being aligned is higher than when the two bit-vectors are relatively equal. Equation 3 gives the formula for estimating the total number of iterations in the main loop of Algorithm 1:

$$I_w \approx [(1 - \max(CR_x, CR_y)) \times \frac{\min(m_x, m_y)}{\max(m_x, m_y)} + 1] \times \max(m_x, m_y), \tag{3}$$

where CR_x and CR_y are the compression ratios for bit-vectors x and y respectively:

$$CR_x = size(x)_{compressed}/size(x)_{uncompressed}$$

Every iteration of Algorithm 1 results in either a fill append or a literal append. Thus the total number of appends is equal to the number of iterations.

We estimate the number of fill appends (I_f) based on the probability of two fills being aligned in the compressed bit-vectors that are queried. The remaining appends are appends literals (I_l). The probabilities for estimating the number of fill appends differ based on the compression scheme used. In the next three subsections we estimate the query time performance for WAH, EWAH, and PLWAH/CONCISE respectively.

5.1. Performance Estimations for WAH

For WAH, the total number of fill appends is the probability of two fill words being operated at the same time in Algorithm 1, multiplied by the number of iterations in Algorithm 1. The probability of operating two fill words within the same iteration, is at least the product of the probabilities of encountering a fill word in each of the operated bit-vectors:

$$P_{\text{appendFill}} > \frac{\text{Fills}_x}{m_x} \times \frac{\text{Fills}_y}{m_y}.$$

In the equation above we have the “greater” comparison operand because even when it happens for two fill words to be operated at the same time, the probability of them having the same run-length encoded is close to zero. In fact, if the number of fill words would be equal between the two bit-vectors, the number of fill appends would be exactly double the product of the probabilities of encountering a fill word in each of the operated bit-vectors multiplied by the total number of appends. Meaning that, on average, it takes exactly two fill-words from the second bit-vector to consume a fill-word from the first fill-vector, and vice-versa. However, when the number of fill words differs between the two queried bit-vectors, then, on average, it will take more than two fill words from the larger bit-vector to consume one fill-word from the smaller bit-vector. Based on these, we estimate that the number of append-fills in WAH compression scheme can be approximated by:

$$I_f = \frac{\text{Fills}_x}{m_x} \times \frac{\text{Fills}_y}{m_y} \times I_w \times (3 - \frac{\min(m_x, m_y)}{\max(m_x, m_y)}), \tag{4}$$

where Fills_x and Fills_y are the number of fill words in bit-vector x and y respectively, while m_x and m_y are their respective sizes. Plugging I_f and I_l in Equation 2, we obtain:

$$t_{\text{WAH}} = C_a(m_x + m_y) + D_l(m_{x_l} + m_{y_l}) + D_f(m_{x_f} + m_{y_f}) + C_l I_l + C_f I_f + C_0 I_w. \tag{5}$$

The number of literal appends (I_l) can be approximated by subtracting the number of fill appends from the total number off appends $I_l = I_w - I_f$.

5.2. Performance Estimations for EWAH

EWAH query algorithm has some differences from WAH query algorithm. The theoretical advantage of EWAH over WAH is that it does not require any decoding for the literal words. On the other hand, maintaining the counter of the literal words when appending literals at query time adds cost to EWAH query processing time. Furthermore, some EWAH compressed bitmaps may have a larger size, due to its smaller fill encoding capacity, and thus EWAH may require more decodes for fills than WAH in these cases. The Equation 4 is applicable for EWAH as well, considering EWAH number of literals and fills in the compressed bit-vector.

Even if EWAH does not explicitly decode of the literal words, it still has to access them once, and it also has to decrement the counter that tells how many literals are still following. This means that decoding literals for EWAH may be cheaper, however, it still comes at a cost. In general, the expected EWAH total query time would be:

$$t_{\text{EWAH}} = C_a(m_x + m_y) + \frac{CR_x + CR_y}{2} \times D_l(m_{x_l} + m_{y_l}) + D_f(m_{x_f} + m_{y_f}) + C_l I_l + C_f I_f + C_0 I_w. \tag{6}$$

In Table 2, we set the Decode Fill time for EWAH the same as the one for WAH. However this is only true for the case when there are no fill words. As EWAH compresses better the bitmaps, the cost of decode literal decreases.

5.3. Performance Estimations for PLWAH/CONCISE

As discussed earlier, Algorithm 1 performs the same number of iterations for both, PLWAH and WAH. Because these two schemes use the same hybrid encoding, the number of fill appends and literal appends between the two schemes should be equal when performing the same query. Thus, the estimation results from Equation 4 are also valid for PLWAH. There are several differences between these two schemes. One of them is that PLWAH utilises less memory to store the compressed bitmap, and makes fewer memory accesses when decoding the bitmap. On the other hand, decoding a PLWAH word requires more time than decoding a WAH word due to its checks for sparse literals encoded within fills. Finally, when compressing the resulting bit-vector on the fly, at query time, every literal append should check whether or not the last word is a fill and if this is a sparse literal that can be encoded together with the fill.

These variations result in query time differences when comparing these two schemes. The ideal case for PLWAH is when it has a sparse literal between each fill word. In this case PLWAH can exploit the better memory utilization and fewer bitmap accesses at maximum. The worst case is when there are no sparse literals encoded within the fills. In this case PLWAH still has a more complex query algorithm and append literal algorithm than WAH, while it does not benefit from the memory advantage.

In the next section we use the equations derived in section 5 to estimate query times for EWAH, PLWAH/CONCISE and WAH compressed bitmaps over synthetic and real data sets. Ignoring the loop overhead C_0 does not introduce any significant amount of error, and the estimations are within a 5% error, as it is shown in the experimental results.

6. Performance Evaluation

In this section we evaluate the different word-aligned compression methods using both synthetic and real data sets. We first describe the experimental setup and the data sets used. Then, we conduct a performance study that evaluates the effect of cardinality (bitmap density), clustering, query dimensionality, and query cardinality of point range queries for four different word-aligned compression methods: WAH, PLWAH, CONCISE and EWAH.

6.1. Experimental Setup

Synthetic data sets were generated following two different distributions: `uniform` and `zipf`. These two distributions are representative of real-world bitmaps because continuous attributes and even high cardinality attributes are transformed using discretization or binning before creating the bitmap indices. The distribution of the data into bitmap indices depends on the method used for discretization. For example, if equi-populated bins (bins containing roughly the same number of objects) are used, the distribution of the bitmaps is uniform. However, if the quantization method is based on clustering or on the density of the data, then most likely the bins would follow a skewed distribution. The zipf distribution generator, assigns each bit a probability of

$$p(k,n,f) = \frac{1/k^f}{\sum_{i=1}^{n}(1/i^f)}$$

where n is the number of elements determined by cardinality, k is their rank, and the coefficient f creates an exponentially skewed distribution. We generated data sets for $f = 0$ through $f = 5$.

Four real data sets were used in our experiments and their attributes were discretized using the least squares quantization method (Lloyd, 1982):

- `KddCup`[1]. This data set was used for The Third International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD-99. The data set contains 4,898,431 rows and 42 attributes.
- `Berkeley Earth`[2]. The Berkeley Earth Surface Temperature Study has created a preliminary merged data set by combining 1.6 billion temperature reports from 16 preexisting data archives. For our experiments we use a subset containing 14,786,160 rows and 7 attributes.
- `Linkage`[3]. It contains element-wise comparison of records with personal data from a record linkage setting. The source of this data set are records stem from the epidemiological cancer registry of the German state of North Rhine-Westphalia. The data set contains 5,749,132 rows and 12 attributes.
- `Skin`[4]. This data set was collected by randomly sampling B,G,R values from face images of various age groups (young, middle, and old), race groups (white, black, and asian), and genders obtained from FERET database and PAL database. This data set is of the dimension $2,450,570 \times 4$ where first three columns are B,G,R (x1,x2, and x3 features) values and fourth column is of the class labels.

Experiments were executed over a machine with an Intel Core i7-2600 processor (8MB Cache, 3.20

[1] http://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data
[2] http://berkeleyearth.org/dataset/
[3] http://archive.ics.uci.edu/ml/datasets/Record+Linkage+Comparison+Patterns
[4] https://archive.ics.uci.edu/ml/datasets/Skin+Segmentation

GHz) and 8 GB of memory, running Windows 7 Enterprise. In addition, to validate our query time estimation algorithms, we repeated the estimation experiments on a Linux Suse 12.2 with Intel Core i7-3770 processor and 16 GB of memory.

For single-point queries, a set of queries over two randomly selected columns (from different attributes) was generated and queries consist on `AND`ing the corresponding bitmaps. For query time, we do not take into account the time required to load the bitmaps into memory. Since all bitmaps fit into main memory, loading time can be done once and amortized over a large number of queries. The query set was executed six times and the result for the first run was discarded to prevent Java's *just-in-time* compilation to mislead the results. The time from the other five runs were averaged together and the results are reported. For size comparisons, compression ratio is computed as $\frac{size_{compressed}}{size_{verbatim}}$. Smaller compression ratios means that the bitmaps compress better when compared to verbatim bitmaps.

We do not include the I/O times in this paper, and only focus on in-memory usage of bitmap indices. Many databases use temporary in-memory bitmap indices to speed-up queries. Also more complex database queries require multiple rounds of bit-wise operations, and the intermediate results produced can be efficiently reused in further operations to answer them. In these cases our research can be helpful in determining on the best compression method.

6.2. Algorithm Implementation

In order to allow a fair comparison with the already available implementations of WBC/EWAH and Concise in Java, and described in (Lemire et al., 2010) and (Colantonio and Di Pietro, 2010) respectively. We also implemented WAH and PLWAH in Java following the details and optimisations described in (Wu et al., 2002) and (Deliege and Pederson, 2010), respectively. For EWAH we used version 0.9.0 available, as published by the authors, and for CONCISE we used the version published in 2010 by the authors. We tried to have the same data structures and memory allocation policies where possible, for all four compression schemes. The Java version used for these experiments is 7.0.67 and the experiments were run with the server mode enabled.

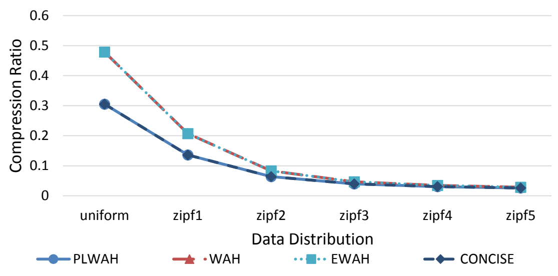
All four compression schemes considered in this section have the 32-bit implementation. For all experiments $w = 32$ bits. For PLWAH, we only consider 5 bits for the position list, storing sparse literals with only one heterogeneous bit, i.e. $s = 1$.

Although our implementation is in Java, we are confident that if implemented in another programming language the relative performance of the four methods will be similar, as in our estimations, we took into consideration the complexion of the lower level algorithms such as *decode* and *append*, and provide the methods to determine the number of calls for each method.

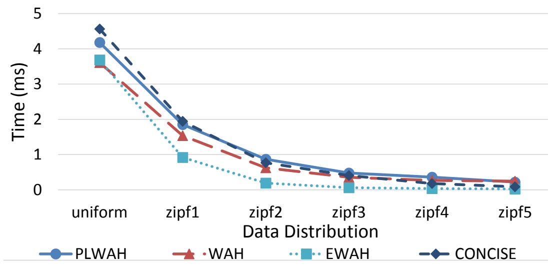
6.3. Effect of skewness or data distribution

In this section we evaluate the effect of the skewness of the data by varying the exponent in a Zipfian distribution from 0 (uniform) to 5. Evaluation is done in terms of compression ratio and query time of a 2-dimensional point query for two attributes with cardinality 100. Each data set contains 10 million rows and the query time represents the average time over one thousand AND operations in milliseconds. Figure 3(a) shows the compression ratio for WAH, PLWAH/CONCISE and EWAH over synthetic data sets. By increasing the skewness, compression ratio improves for all encoding schemes. As per our earlier analysis, PLWAH and CONCISE have exactly the same compression ratio, while WAH and EWAH have very similar compression ratios. PLWAH and CONCISE show better compression ratios when compared to WAH for low skewness values. In these cases the position bits are used often to compress a near-fill literal word. As the data becomes skewer, the runs become longer, and the usage of position bits is less significant in terms of compression ratio.

Figure 3(b) shows the query times for these compression techniques over data sets with different

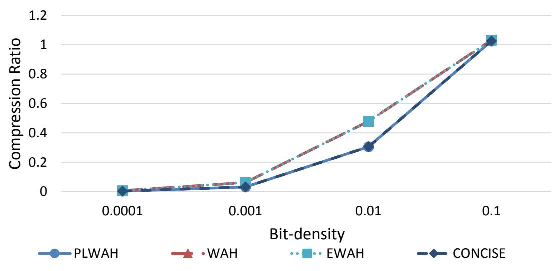


(a) Compression ratio function of data distribution

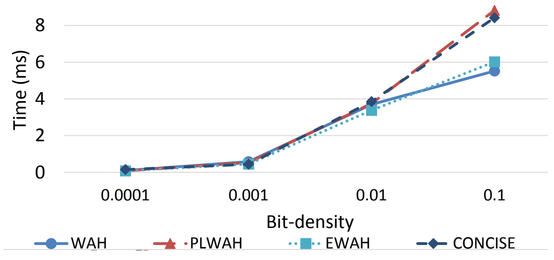


(b) Query time function of data distribution

Fig. 3. Effects of data distribution on compression and query. time (Data rows: 10 Million)



(a) Compression ratio function of bit density



(b) Query time function of bit density

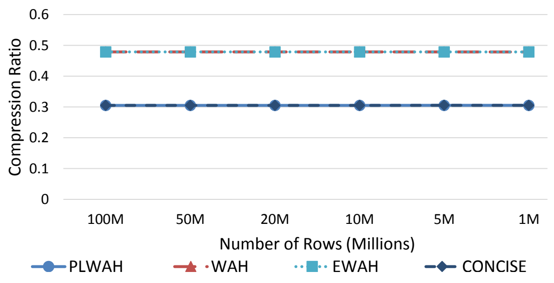
Fig. 4. Effects of bit density on compression and query time. (Data rows: 1 Million)

skewness levels. All encoding schemes perform better over skewed data, as they perform logical operations directly over fragments of compressed data. PLWAH and CONCISE are slower than WAH and EWAH over less skewed data, as they require more decoding. Over more skewed data, however, PLWAH and CONCISE perform better than WAH. This is due to a better compression that shortens the main loop from Algorithm 1, while having an insignificant overhead on decoding.

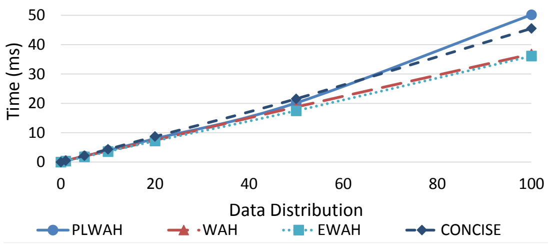
6.4. Effect of bit density

In this section we evaluate the effect of bit density, i.e. the number of set-bits in a bitmap simulated by using uniform random data and varying the cardinality of the attributes ($bit\ density = \frac{1}{attrib\ cardinality}$). Evaluation is done in terms of compression ratio and query time of a 2-dimensional point query. The data sets are randomly generated with one million rows and an uniform distribution. Figure 4(a) shows the compression ratios of the four encoding schemes while increasing the bit density. Here PLWAH/CONCISE compress better for densities of 0.01 and 0.001, where the position bits are actually used. For the data set with a set-bit density of 0.0001, the compression ratio is very high and thus the use of position bits is not significant, while the data set with bit density 0.1, is not compressed at all as the average run length is 10.

Figure 4(b) shows the query times over data sets with different set-bit densities. The time is measured in milliseconds and represents the cumulative time of 1000 2-dimensional point queries. Given an uniform distribution, for higher densities there is almost no compression. This gives an advantage to EWAH, since it doesn't need to decode the literals. On the other hand, PLWAH/CONCISE have a more expensive decoding algorithm and thus perform worse for high bit density data.



(a) Compression ratio function of number of rows



(b) Query time function of number of rows

Fig. 5. Effects of partition size on compression and query time. (Data rows: 1-100 Million)

Table 3. Query time per 1 million rows function of partition size (ms)

	1M	5M	10M	20M	50M	100M
PLWAH	0.4103	0.4039	**0.3949**	0.3980	0.4028	0.5016
CONCISE	0.4403	0.4351	**0.4344**	0.4349	0.4354	0.4550
WAH	0.4203	0.4092	**0.3713**	0.3790	0.3804	0.3876
EWAH	0.4303	0.3554	0.3547	**0.3496**	0.3592	0.3612

6.5. Partition size

Bitmap indices are amenable for parallel execution and they can be partitioned horizontally. Horizontal partitioning occurs when there is a limit on the number of rows that can go into a partition. Assuming that all queried partitions are on the same machine, the next set of experiments tries to determine what would be the optimal horizontal partition size, i.e. how many rows would provide the best trade-off between query time and compression. Note that this experiment does not take into account the network overhead in the case of building a fully distributed system. Nor does it account for disc access. Figure 5(a) and Figure 5(b) show the compression size and query time for a uniformly distributed data set with 100 million rows and varying partition size of 100K, 1M, 10M, 50M, 100M rows. The attribute cardinality for this data set is 100. The compression ratio is constant for all partition sizes, thus only query time plays a significant role in choosing the optimal partition size. Figure 5(b) shows the average query times over 100 2-dimensional point queries for different partition sizes.

A good indicator of how the partition size affects the query time would be the query time per row (or per 1 million rows). Table 6.5 shows the query times per 1 million rows for different partition sizes.

As the partition size grows from 1 million rows to 5 millions, the query time per row actually decreases for all encodings. The same pattern is observed when increasing the partition size from 5 million to 10 million. When increasing further to 20 million however, for most of the encodings the query time per row increases. This trend becomes more obvious as the partition size increases even more to 100 million. Table 3 shows query times measured in milliseconds per 1 million rows. The best times for each method are highlighted in bold. As can be seen, the optimal partition size on our machine, for all compression schemes, is somewhere between 10 and 20 million rows. The optimal partition size is hardware dependent, and the CPU cache size is the most important limitation for the number of rows that should be allocated per partition.

6.6. Range evaluation

As discussed in Section 3, there are several possibilities for the encoding of the resulting bit-vector when evaluating range queries: 1) Compress the resulting bitmap on the fly. 2) In-Place - store the resulting

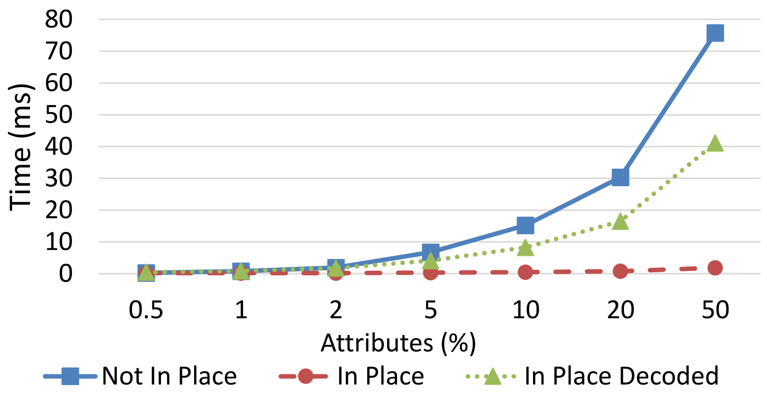


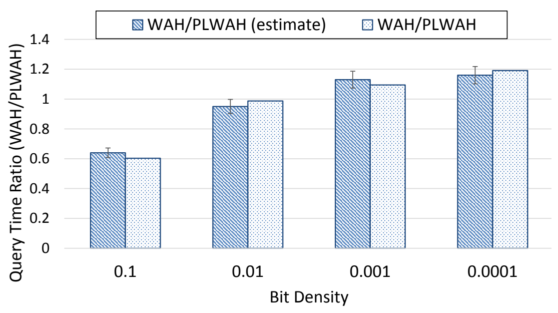
Fig. 6. Range Query Evaluation - Caption percent of Range Queried. (Data rows: 1 Million)

bitmap in place of one of the operands. 3) Decode both operands and perform an in-place operation. Figure 6, shows query times for the three described alternatives using WAH, when the range queries are increasing the number of columns per attribute. We stop at 50% as it represents the worse case for a range query. If the range queried involves more than half of the columns for the attribute then the non-queried columns are ORed together and the result is complemented. Effectively, a range query never queries more than half of the columns for a given attribute. As can be seen in Figure 6, as the percentage of attribute columns queried increases in range queries, the in-place query execution exhibits the best performance. In-place queries show significant improvement over the two other methods because it does not require memory allocation for the intermediate results and the resulting bitmap vector. The data set used for this experiment has a uniform distribution, 1 million rows and cardinality 1000. A set of 100 queries has been used for querying the data set. The query times reported are the average times over 100 range queries.

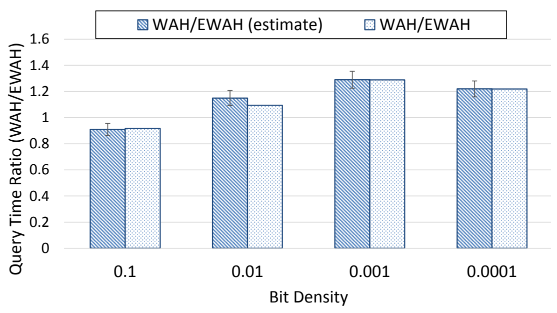
6.7. Query Time Estimation Over Synthetic Data

In Sections 4 and 5 we analysed the complexity of the querying algorithms for the different compression schemes, and argued that we can estimate the relative performance between EWAH, WAH and PLWAH. Our ultimate goal is to estimate which of the analysed bitmap encoding schemes has the potential for faster queries, and by what margin. In this section we estimate query times for the synthetic data sets used in section 6.4, and four real data sets. Then we compare the estimations with the actual measured values for one thousand random selected single point AND queries.

The data sets and the configurations of the two machines used for measuring query times were described in Section 6.1. Figures 7(a) and 7(b) show the estimated versus recorded query time ratios for WAH/PLWAH and WAH/EWAH respectively. These are the estimations for the data sets used in section 6.4. Because the density of the data set is the factor that affects the most the compression ratio, the relative performance of the different encoding schemes changes considerably when varying the bit-density from 0.1 to 0.0001. We chose these data sets to show the robustness of our estimations methods as the relative performance between the three encoding schemes varies. Figure 7(a) shows that PLWAH becomes faster than WAH for bit-densities smaller than 0.01. Using the equations for PLWAH/CONCISE from Section 5, we were able to estimate the relative performance between WAH and PLWAH with a sub-5% error. Figure 7(b) shows the performance estimations for EWAH against WAH. EWAH is slower than WAH only for high bit-density data i.e. $d = 0.1$. This is mostly because of the high cost of appending literals and the following remaining literal counter maintenance.

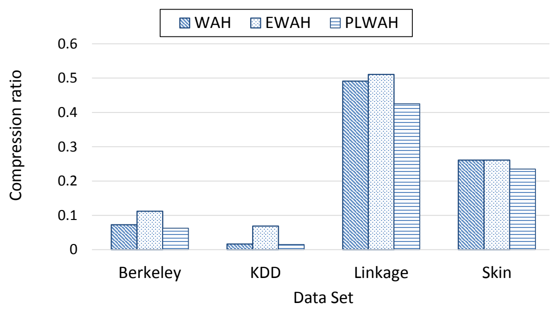


(a) Estimated vs recorded query time ratio WAH/PLWAH over synthetic data

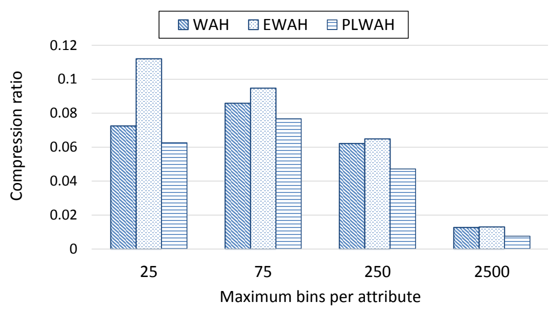


(b) Estimated vs recorded query time ratio WAH/EWAH over synthetic data

Fig. 7. Query time estimation over synthetic data. (Data rows: 1 Million)



(a) Compressed ratios of the real data sets



(b) Compression ratio for the Berkeley data set with different quantization resolutions

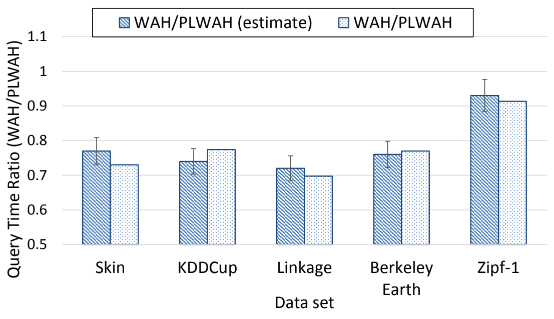
Fig. 8. Data compression ratios

6.8. Performance Over Real Data

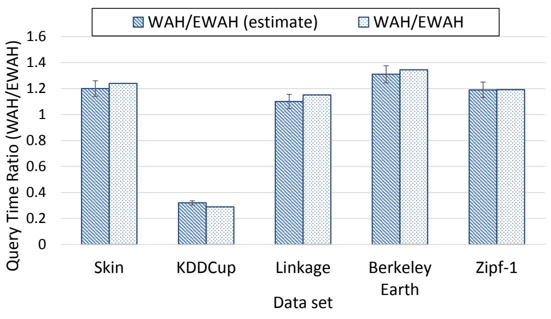
Figure 8(a) shows the compression ratios of the four real data sets described earlier. Each attribute was discretized into 25 bins or less. As can be seen, PLWAH produces the more compact bitmaps while EWAH is the least effective for compressing the bitmap indices. For these datasets, WAH performance is very close to PLWAH. The reason is that a small number of bins have been used to discretize the attributes and the number of near-fill literals is not significant.

Query time was measured and the performance estimation was computed for the real datasets. Figures 9(a) and 9(b) show the expected vs. measured query time ratios between WAH/PLWAH and WAH/EWAH, respectively. In Figures 9(a) and 9(b) we also added the zipf-1 data set used in Section 6.3, given that zipf-1 is the distribution that can be observed in many real-world data sets (Zipf, 1949). One thing worth noting in Figure 9(b) is the poor performance of EWAH for the `KDDCup` data set (EWAH 70% worse that WAH). The explanation can be found by examining Figure 8(a) and realizing that EWAH compressed bitmaps are over 3.5x larger that WAH compressed bitmaps. The reason is that this data set has many sparse columns and the 16 bits used by EWAH to encode the fills are not sufficient to encode most of the runs. Since the compression ratio is considerably worse for EWAH, Algorithm 1 needs more iterations to process EWAH-compressed columns. Because EWAH size is included in our query time estimation, this non-ordinary behaviour is accurately predicted.

In Figure 10 we repeated the same estimation experiment as in Figure 9, however here we only used

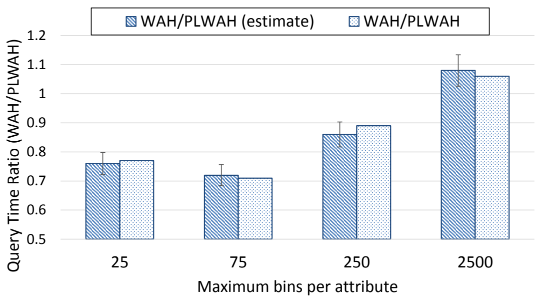


(a) Estimated vs recorded query time ratio WAH/PLWAH over real data

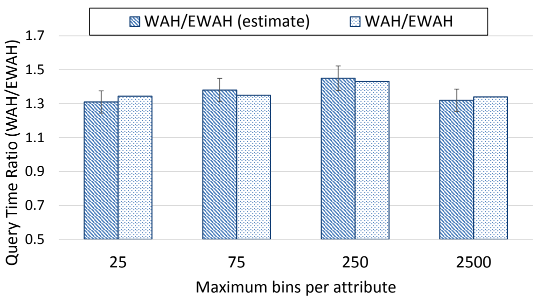


(b) Estimated vs recorded query time ratio WAH/EWAH over real data

Fig. 9. Query time estimation over real data



(a) Estimated vs recorded query time ratio WAH/PLWAH for the Berkeley-Earth data-set when varying the quantization resolution



(b) Estimated vs recorded query time ratio WAH/EWAH for the Berkeley-Earth data-set when varying the quantization resolution

Fig. 10. Query time estimation for the Berkeley-Earth data-set when varying the quantization resolution.

the `berkeley-earth` data set, and varied the attribute quantization from 25 to 2500 bins. As expected, as the number of bins (cardinality) increases, the bitmap vectors become sparser, and PLWAH outperforms WAH. EWAH was faster than WAH in all four cases, however the difference decreased when we had 2500 bins per attribute. The compression ratios of the three compression methods for different quantization resolutions are shown in Figure 8(b).

For all data sets used in this paper, the relative performance of WAH/PLWAH and WAH/EWAH was estimated within the confidence level of 95%. The experiments were reproduced on the Linux machine as well and the same relative performance was observed.

7. Conclusions and Future Work

In this paper we have evaluated several word-aligned bitmap compression techniques in terms of compression ratio but more importantly query time for point and range queries. As expected, no encoding is better than all others in all cases. However, these results evidence that there are specific scenarios in which one method should be preferred over the others. Depending on one's needs, the preference could be towards a more compact bitmap index or faster queries. In this study we provide insights as to which encoding should be preferred depending on the data set. For example, EWAH could be preferred for bitmaps that are hard-to-compress, that have short fills and mostly literals; PLWAH could be preferred for very sparse bitmaps where it compresses significantly better than WAH and may have faster query times.

Furthermore, we formalise a method for estimating relative query performance between two encodings before actually compressing the bitmaps. This method only requires a single traversal of the bitmaps for providing these estimations, and has proven to be accurate. We evaluated our estimation method on synthetic data, as well as on four real data sets, and predicted the relative performance of WAH versus PLWAH and WAH versu EWAH within a confidence level of 95%. For future work we aim to integrate these estimation methods into a framework, that automatically sets the encoding method and the word length, as proposed in Guzun et al. (2014), and further extend them to integrate more word-aligned compression methods as well as different encoding lengths.

References

Antoshenkov, G. (1995). Byte-aligned bitmap compression. In *DCC '95: Proceedings of the Conference on Data Compression*, page 476, Washington, DC, USA. IEEE Computer Society.

Apaydin, T., Tosun, A. c., and Ferhatosmanoglu, H. (2008). Analysis of basic data reordering techniques. In *International Conference on Scientific and Statistical Database Management*, pages 517–524.

Colantonio, A. and Di Pietro, R. (2010). Concise: Compressed 'n' composable integer set. *Information Processing Letters*, 110(16):644–650.

Deliege, F. and Pederson, T. (2010). Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps. In *Proceedings of the 2010 International Conference on Extending Database Technology (EDBT'10)*, pages 228–239.

Fabian Corrales, D. C. and Sawin, J. (2011). Variable length compresssion for bitmap indices. In *ACM International Conference on Database and Expert Systems Applications*, pages 381–395.

Fusco, F., Stoecklin, M. P., and Vlachos, M. (2010). Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. *Proceedings of the VLDB Endowment*, 3(2):1382–1393.

Guzun, G., Canahuate, G., Chiu, D., and Sawin, J. (2014). A tunable compression framework for bitmap indices. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 484–495. IEEE.

Kaser, O., Lemire, D., and Aouiche, K. (2008). Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes. In *ACM 11th International Workshop on Data Warehousing and OLAP*, pages 1–8.

Lemire, D., Kaser, O., and Aouiche, K. (2010). Sorting improves word-aligned bitmap indexes. *Data and Knowledge Engineering*, 69:3–28.

Lemire, D., Kaser, O., and Gutarra, E. (2012). Reordering rows for better compression: Beyond the lexicographic order. *ACM Transactions on Database Systems*, 37(3):20:1–20:29.

Lloyd, S. P. (1982). Least squares quantization in pcm. In *IEEE Transactions on Information Theory*.

Malik, H. H. and Kender, J. R. (2007). Optimizing frequency queries for data mining applications. In *International Conference on Data Mining*, pages 595–600.

Pinar, A. and Heath, M. T. (1999). Improving performance of sparse matrix-vector multiplication. In *Proceedings of Supercomputing*.

Pinar, A., T.Tao, and Ferhatosmanoglu, H. (2005). Compressing bitmap indices by data reorganization. In *Proceedings of the 2005 International Conference on Data Engineering (ICDE'05)*, pages 310–321.

van Schaik, S. J. and de Moor, O. (2011). A memory efficient reachability data structure through bit vector compression. In *ACM SIGMOD International Conference on Management of Data*, pages 913–924.

Wu, K., Otoo, E., and Shoshani, A. (2004). On the performance of bitmap indices for high cardinality attributes. *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, 30:24–35.

Wu, K., Otoo, E. J., and A.Shoshani (2002). Compressing bitmap indexes for faster search operations. In *Proceedings of the 2002 International Conference on Scientific and Statistical Database Management Conference (SSDBM'02)*, pages 99–108.

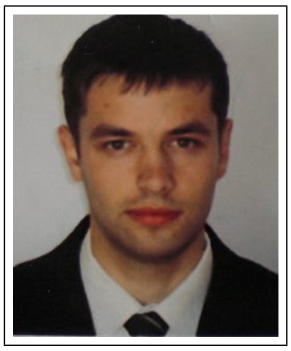
Wu, K., Otoo, E. J., and Shoshani, A. (2001a). A performance comparison of bitmap indexes. In *CIKM 2001*, pages 559–561.

Wu, K., Otoo, E. J., and Shoshani, A. (2006). Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38.

Wu, K., Otoo, E. J., Shoshani, A., and Nordberg, H. (2001b). Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory.

Zipf, G. (1949). Human behaviour and the principle of least-effort. Addison-Wesley, Cambridge, MA.

Author Biographies



Gheorghi Guzun is currently pursuing a Ph.D. degree in Electrical and Computer Engineering at The University of Iowa, IA, USA. He received a B.S. degree in Electrical Engineering from the Technical University of Moldova in 2010. His current work focuses on indexing solutions for data warehouses, hardware-driven software design, query algorithms, and distributed databases.

Guadalupe M. Canahuate joined the Electrical and Computer Engineering department of the University of Iowa as an Assistant Professor in 2011. She earned her Ph.D. in Computer Science and Engineering from The Ohio State University in 2009 and her M.S. degree in Computer and Infornation Science in 2003. She graduated with a B.S.E. in Computer Science from Pontificia Universidad Catlica Madre y Maestra, Dominican Republic in 2000. Her research focus on high-dimensional big data management, indexing, and analysis. .

Correspondence and offprint requests to: Gheorghi Guzun, Department of Electrical and Computer Engineering, The University of Iowa, Iowa City, IA 52242, USA. Email: gheorghi-guzun@uiowa.edu