Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps

François Deliège
Department of Computer Science
Aalborg University, Denmark
fdeliege@gmail.com

Torben Bach Pedersen
Department of Computer Science
Aalborg University, Denmark
tbp@cs.aau.dk

ABSTRACT

Compressed bitmap indexes are increasingly used for efficiently querying very large and complex databases. The Word Aligned Hybrid (WAH) bitmap compression scheme is commonly recognized as the most efficient compression scheme in terms of CPU efficiency. However, WAH compressed bitmaps use a lot of storage space. This paper presents the Position List Word Aligned Hybrid (PLWAH) compression scheme that improves significantly over WAH compression by better utilizing the available bits and new CPU instructions. For typical bit distributions, PLWAH compressed bitmaps are often half the size of WAH bitmaps and, at the same time, offer an even better CPU efficiency. The results are verified by theoretical estimates and extensive experiments on large amounts of both synthetic and real-world data.

1. INTRODUCTION

Compressed bitmap indexes are increasingly used to support efficient querying of large and complex databases. Example applications areas include very large scientific databases and multimedia applications, where the datasets typically consist of feature sets with high dimensionality. The present work was motivated by the need to perform (one- or multi-dimensional) range queries in large multimedia databases for music (Music Warehouses). Here, music snippets are analyzed on various cultural, acoustical, editorial, and physical aspects, and the extracted features are high-dimensional, and range over very wide intervals. However, the exact same characteristics apply to a wide range of applications within multimedia and scientific databases. While bitmap indexes are commonly accepted as an efficient solution for performing search on low-cardinality attributes, their sizes increase dramatically for high-cardinality attributes. Therefore, the central challenge is to develop an efficient compression technique for sparse bitmaps and that has a low overhead for low-cardinality attributes.

The first use of a bitmap index in a DBMS dates back to 1987 [10]. The index was made from uncompressed bitmaps and suffered from tremendous storage requirements proportional to the cardinality of the indexed attribute, causing it to become too large to fit in memory and leading to a deterioration of the performance [18]. Since then, various approaches have been studied to improve bitmap indexes for high-cardinality attributes. A deeper review of bitmap index extensions, compression techniques, and technologies used in commercial DBMSes is presented in [15].

The *binning* technique partitions the values of the index attributes into ranges [13, 17]. Each bitmap captures a *range* of values rather than a single value. Binning techniques prove to be useful when the attribute values can be partitioned into sets. In [8], a binning technique using partitioning based on query patterns, their frequencies, and the data distribution is presented and further improves the index performance. *Bit-slicing* techniques rely on an ordered list of bitmaps [11]. If every value of an attribute can be represented using n bits, then the indexed attribute is represented with an ordered list of n bitmaps, where for example, the first bitmap represents the first bits of the values of the indexed attribute. Dedicated arithmetic has been developed to operate directly on the bitmaps in order to, for example, perform ranges queries [12]. The Attribute-Value-Decomposition (AVD) is another bit-slicing technique designed to encode both *range-encoded* and *equality-encoded* bitmap indexes [5]. Both lossy and lossless bitmap compression schemes have been applied to bitmap indexes. The Approximate Encoding, (AE), is an example of lossy bitmap compression scheme [4]. An AE compressed bitmap index returns false-positives but is guarantied not to return false-negatives. The accuracy of an AE compressed bitmap index ranges from 90% to 100%.

The Byte-aligned Bitmap Compression (BBC) [3] and the Word Aligned Hybrid (WAH) [16] are both lossless compression schemes based on run-length encoding. In run-length encoding, continuous sequences of bits are represented by one bit of the same value and the length of the sequence. The WAH compression scheme is currently regarded the most CPU efficient scheme, and is faster than, e.g., BBC. The performance gain is due to the enforced alignment with the CPU word size. This yields more CPU friendly bitwise operations between bitmaps. However, WAH suffers from a significant storage overhead, an average of 60% storage overhead over BBC was reported [16]. More recently, for attributes with cardinalities up to 10,000, hybrid techniques based on combined Huffman run-length encoding have shown their superiority from a size point of view [14]. However, both storage and performance of such compressed bitmap indexes decrease for bigger cardinalities and performing bitwise operations (OR/AND) on such bitmaps is very expensive since the bitmaps must first be de-compressed and later recompressed. General-purpose data compression algorithms, such as PFOR and PFOR-DELTA [19], often offer very efficient decompression but are not well-suited for bitmap compression as they necessitate a preliminary decompression to operate on the values.

This paper improves upon existing work by offering a lossless bitmap compression technique that outperforms WAH, currently seen as the leading bitmap compression scheme for high-cardinality attributes, from both a storage and a performance perspective. Other extensions to the bitmap index (bit-slicing, etc.) will thus also benefit from the proposed compression scheme to represent their underlying bitmaps.

Specifically, the paper presents Position List Word Aligned Hybrid (PLWAH), a new bitmap compression scheme. PLWAH is based on the observation that many of the bits used for the *run-length counter* in the WAH compression scheme are in fact never used, since runs never become long enough to need all the bits. Instead, these bits are used to hold a "position list" of the set/unset bits that follow a 0/1 run. This enables a significantly more efficient storage use than WAH. In fact, PLWAH compressed bitmaps are often only half the size of WAH compressed ones, and at the same time, PLWAH is faster than WAH, thus PLWAH provides "the best of both worlds." PLWAH guarantees a maximum bitmap size in words that is at most the number of set bits. Furthermore, PLWAH has a very low overhead on non-compressible bitmaps (1/64 or 1/32 depending on word length).

Additional contributions include the removal of the active word and bitmap length that significantly reduce the size of compressed bitmap indexes composed of very short compressed bitmaps, the reduction of the conditional branches required to decompress a word, and the creation of an adaptive counter that allows extremely sparse bitmaps to be compressed efficiently without necessitating longer words that increase the size of the bitmaps.

This paper shows that for uniformly distributed bitmaps, the hardest bitmaps to compress for compression schemes based on run length encoding, PLWAH uses half the space of WAH. On clustered bitmaps, PLWAH also uses less storage than WAH. These results are shown in a detailed theoretical analysis of the two schemes. The paper also presents algorithms that perform efficient bitwise operations on PLWAH compressed bitmaps and analyzes their complexity. Again, the analysis shows PLWAH to be faster than WAH. The theoretical results are verified by extensive experimentation on both synthetic and real-world (music) data. The experiments confirm that PLWAH uses significantly less storage than WAH, while also outperforming WAH in terms of query speed. While developed in the specific context of Music Warehouses, the presented compression scheme is applicable to any kind of bitmaps indexes, thus making the contribution generally applicable.

The paper is structured as follows. Section 2 describes PLWAH compression scheme. In Section 3, we establish the upper and lower bounds of the compression ratio, provide size estimates for both uniformly distributed and clustered bitmaps, and discuss the impact of the size of the position list. The algorithms for performing bitwise operations on the compressed bitmaps are presented in Section 4, along with an analysis of the time complexity of the presented procedures. In Section 5, the theoretical size and time complexity estimates are verified through extensive experiments on both synthetic and real data sets. Finally, in Section 6, we conclude and present future research directions.

2. PLWAH COMPRESSION

The PLWAH compression is composed of four steps; they are explained with an example to ensure clarity. Assume an uncompressed bitmap composed of 175 bits and a 32-bit CPU architecture. The PLWAH compression steps are detailed below and are illustrated in Figure 1.

Step 1. The uncompressed bitmap is divided into groups of equal size, corresponding to the word length of the CPU architecture minus one. In our example, the first four groups have a size of 31 bits. The last group is appended with 11 zeros to reach a size of 31 bits.

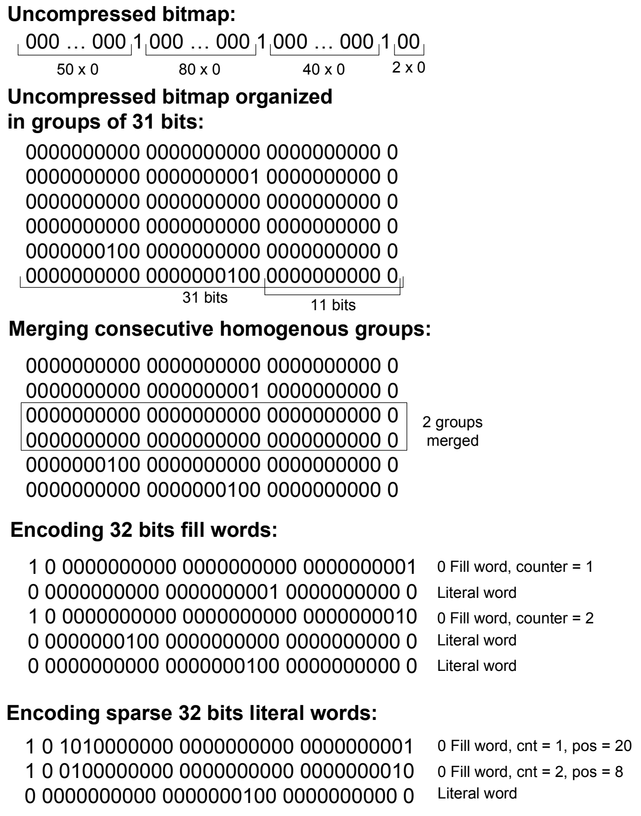


Figure 1: Example of PLWAH32 compression

We thus have six 31 bit long groups.

Step 2. Identical adjacent homogeneous groups, i.e., groups composed of either 31 set bits or 31 unset bits, are marked as candidates for a merge. In our example, the first group is a candidate for a merge since it is exclusively composed of homogeneous bits. Similarly, in the third and fourth groups, all bits are unset, so the two groups are candidates for a merge. The second, fifth, and sixth groups are not homogeneous and therefore are not candidates.

Step 3. An additional bit is appended to the groups at the position of their Most Significant bit (MSB). A set bit represents a group composed of *homogeneous* bits. Those 32 bit long words starting with their MSB set are referred to as *fill words*. Fill words can be of two types, *zero fill words* and *one fill words*; they are distinguished by their second MSB. Candidate groups for a merge are transformed into fill words. The last 25 Least Significant Bits (LSBs) are used to represent the number of merged groups each fill word contains. In our example, the first group becomes a fill word. Similarly, the third and fourth groups become a fill word with its counter is set to two; this corresponds to the number of merged groups. An extra unset bit is added as MSB to *heterogeneous groups*. Encoded words having their MSB unset are referred to as *literal words*. In our example, the second, the fifth, and the sixth word are transformed into literal words; each starts with an unset bit. The first and second words are fill words, their MSBs are set.

Step 4. Literal words immediately following and "nearly identical"[1] to a fill word are identified. The positions of the heterogeneous bits are calculated and are placed in the preceding fill word. The unused bits located between the fill word type bit and the counter bits are used for this purpose. In our example, 25 bits are

[1]The maximum number of bits differing between a literal word and a fill word to be considered as "nearly identical" will later be defined by a threshold parameter.

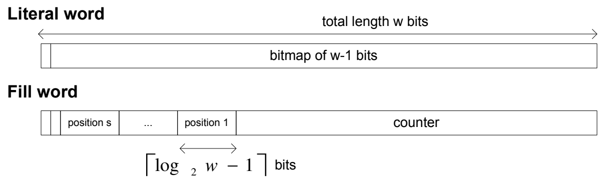


Figure 2: Structure of PLWAH literal and fill words

used for the counter, and 2 bits are used for the representing the word type. We thus have 5 bits remaining, namely the 3rd to the 7th MSB. A heterogeneous bit requires 5 bits to ensure all the possible bit positions in a literal word can be captured. In the example, the second and fourth words are literal words with only one bit different from their preceding fill word; they can be piggybacked. The position of the heterogeneous bit is placed in the position list of the fill word and the literal word is removed from the bitmap. The last word is treated as any other word; in the example, it cannot be piggybacked into its predecessor and is left as it is.

As mentioned above, the PLWAH compressed words are of two kinds: fill words and literal words; their structures are presented in Figure 2. A literal word is identified by an unset MSB, followed by an uncompressed group. Fill words are identified by a set MSB, followed by a type bit to distinguish zero fills from one fills. The remaining bits are used for the optional storage of the list of positions of the heterogeneous bits in the tailing group, and for storing a counter that captures the total number of groups the current fill word represents.

The position list can be empty in three cases: first, if the following word is a fill word of a different type; second, if the following word is a literal but is not "nearly identical" to the current fill word; and third, if the multiple fill words have to be repeated because the counter has reached its maximum value. An "empty" element in the position list is represented with all its value set to zero. The empty position list is represented with all position values set to zero.

3. PLWAH SIZE ESTIMATES

In this section, we compare the space complexity of the WAH and PLWAH compression schemes.

Let w be the number of bits in the CPU word, so that $w = 32$ on a 32-bit CPU and $w = 64$ on a 64-bit CPU. Given an uncompressed bitmap as input, we described in Section 2 that the PLWAH compression scheme divides the uncompressed bitmap into groups of $(w-1)$ bits, referred to as literal groups. For a bitmap with N bits, there are $\lfloor N/(w-1) \rfloor$ such groups plus an optional incomplete word of size $N mod(w-1)$. Unset bits are appended to the incomplete word so that the bitmap is divided into $M = \lceil N/(w-1) \rceil$ groups each containing $(w-1)$ bits. The total length in bits of the uncompressed bitmap with its tailing unset bits is $L = M(w-1)$. A literal word is created by prepending an unset bit to a group. When all groups of a bitmap are represented with literal words, the bitmap is said to be in its *uncompressed form*.

3.1 Compression upper and lower bounds

The maximum compression is obtained when all words are fill words. This happens when all groups but the last are homogeneous and identical, and the last group can be represented using the list of positions of the preceding fill word. The upper bound is thus determined by the maximum number of groups a fill word can contain, plus one for the last literal group. 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 $\left(2^{w-2-s\log_2 w} + 1\right)$ groups of length $w-1$. A very sparse bitmap composed of a long sequence of unset bits and ending with a set bit, 00000000...00001, is an example of a bitmap were the maximum compression can be reached. Such bitmap can be represented with only one fill

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 bitmap 01010101010101...0101010 is an example of an incompressible bitmap for both the WAH and PLWAH compression schemes.

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.

The WAH and PLWAH compression ratio for different bitmap distributions that fall *within* the compression limits are described in Sections 3.2, 3.3, and 3.5. The (rare) bitmap distributions *not* within these boundaries, e.g., bitmaps that would cause counter overflows, are discussed in Section 3.6.

3.2 Compression of sparse and uniformly distributed bitmaps

In a uniformly distributed bitmaps, the probability of having a set bit is independent from the bit position. Such bitmaps can be characterized by their bit density d. The bit density is the fraction of bits that are set compared to the total number of bits. On a uniformly distributed bitmap of density d, the probability of having exactly k or less bit set in a sequence of w bits is given by the binomial distribution $P_u(k, w, d) = C_k^w d^k (1-d)^{w-k}$, where $C_k^w = \frac{w!}{k!(w-k)!}$ is the binomial coefficient and represents the total number of combinations, that k bits can be picked out of a set of w bits. The probability of having no bit set in a $w-1$ bit long word is $P_u(0, w-1, d) = (1-d)^{w-1}$. The probability of having all bits set is: $P_u(w-1, w-1, d) = d^{w-1}$. The probability of having two successive $w-1$ bit long words filled with unset bits is: $P_u(0, 2w-2, d) = P_u(0, w-1, d)P_u(0, w-1, d) = (1-d)^{2(w-1)}$. Similarly, the probability of having two $w-1$ bit long words filled with set bits is: $d^{2(w-1)}$.

The number of words W in a compressed bitmap is $W = M - G$, where M is the total number of groups, and G is the number of pairs of blocks that can be collapsed [16]. For the WAH scheme, G_{WAH} is the number of pairs of adjacent blocks containing only unset bits plus the number of adjacent blocks containing only set bits. The total number of adjacent blocks is $M-1$, and the expected value of G_{WAH} is $\overline{G}_{WAH} = (M-1)P_{col}$, where P_{col} is the probability of collapsing two adjacent blocks. The expected total number of words using WAH is:

$$\begin{aligned}\overline{W}_{WAH} &= M - \overline{G}_{WAH} \\ &= M - (M-1)\Big[P_u\big(0, 2(w-1), d\big) \\ &\quad + P_u\big(2(w-1), 2(w-1), d\big)\Big] \\ &= M\Big[1 - (1-d)^{2(w-1)} - d^{2(w-1)}\Big] \\ &\quad + (1-d)^{2(w-1)} + d^{2(w-1)}\end{aligned} \tag{1}$$

Let L be the total length in bits of the uncompressed bitmap as defined in Section 3.1. On a sparse bitmap, i.e., $d \to 0$, by applying a binomial decomposition, we have $1-(1-d)^{2(w-1)} - d^{2(w-1)} \to 2(w-1)d$. As explained in [16], considering large values of L (and thereby M) and small values of d, the expected number of words can be approximated as follows:

$$\overline{W}_{WAH} \approx M\Big[1-(1-2(w-1)d)\Big] = 2Ld = 2h \qquad (2)$$

where h denotes the number of set bits. Using the definition of bit density, $h = dL$, the number of words in a sparse bitmap can be expressed in terms of set bits as shown in Equation 2. In such a sparse bitmap, all literal words contain only a single bit that is set, and each literal word is separated from the next by a fill word of zeros. On the average, two words are thus used for each bit that is set.

We now calculate the expected total number of words using the PLWAH compression scheme. The probability of having 0 to s set bits in a $w-1$ bit long group is: $\sum_{k=0}^{s} P_u(k, w-1, d)$. The probability of having a $w-1$ bit long group with $w-1$ unset bits followed by a $w-1$ bit long group with 0 to s set bits is: $P_u(0, w-1, d)\sum_{k=0}^{s} P_u(k, w-1, d)$. Similarly, the probability of having a $w-1$ bit long group with all its bits set followed by a group with 0 to s unset bits is: $P_u(w-1, w-1, d)\sum_{k=0}^{s} P_u(w-1-k, w-1, d)$. The expected total number of words is:

$$\begin{aligned}
&\overline{W}_{PLWAH}\\
&= M-(M-1)\Big[P_u(0,w-1,d)\sum_{k=0}^{s}P_u(k,w-1,d)\\
&\quad + P_u(w-1,w-1,d)\sum_{k=0}^{s}P_u(w-1-k,w-1,d)\Big]\\
&= M-(M-1)\Big[(1-d)^{w-1}\sum_{k=0}^{s}C_k^{w-1}d^k(1-d)^{w-1-k}\\
&\quad + d^{w-1}\sum_{k=0}^{s}C_k^{w-1}d^{w-1-k}(1-d)^k\Big]
\end{aligned} \qquad (3)$$

For small values of d and large values of M, we can use a binomial decomposition. The expected number of words can then be approximated as follows.

$$\begin{aligned}
\overline{W}_{PLWAH} &\approx M\Big[1-(1-d)^{2(w-1)}-(w-1)d(1-d)^{2w-3}\Big]\\
&\approx M(w-1)\Big[2d-d(1-(2w-3)d)\Big]\\
&\approx Ld = h
\end{aligned} \qquad (4)$$

Looking at the expected number of words relative to the expected number of set bits, we have:

$$\frac{\overline{W}_{WAH}}{h} \approx 2 \qquad \text{and} \qquad \frac{\overline{W}_{PLWAH}}{h} \approx 1 \qquad (5)$$

The compressed size of a sparse bitmap is thus directly proportional to the number of bits set. In such a sparse bitmap, all words are fill words of unset bits with one position set in the position list of the fill word. The compression ratio of PLWAH is asymptotically, within the limits of compressibility detailed in Section 3.1, twice the compression ratio of WAH for a uniformly distributed bitmap as the bit density goes to zero.

Figure 3 shows the expected number of words per set bit for WAH and PLWAH depending on the bit density. The behavior of the curves for low densities is explained by the previous approximations detailed in Equation 5. The error due to the approximation in Equation 5 is less than a 1% for bit densities $d < 0.001$. For higher densities, the number of words per set bit drops linearly to the bit density: if the density is too high to have fill words, the bitmaps are filled with literal words, they have become incompressible and have reached their maximum size as explained by the lower bound of the compression ratio established in Section 3.1.

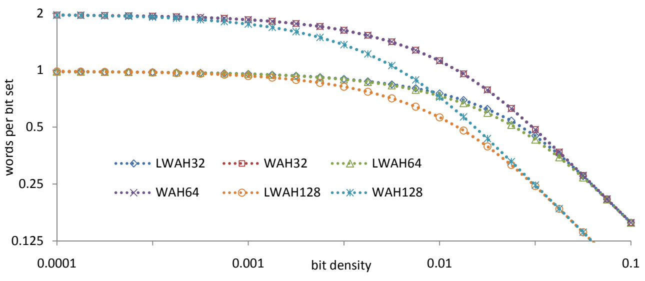


Figure 3: Word count estimates on uniformly distributed bitmaps

3.3 Compression of clustered bitmaps

We now study the compression of a bitmap constructed by a random walk modeled by the following Markov process. A bit has a probability p to be set if its preceding bit is unset, and a probability q to be unset if its preceding bit is set. The bit density is: $d = (1-d)p + d(1-q) = p/(p+q)$. The clustered bitmap can thus be fully described by the density d and its clustering factor $f = 1/q$.

Let $P_c(k, w, d, f|b_0)$ denote the probability of having k set bits in a block of w bits when the bit preceding the block is b_0, the clustering factor is f and the bit density is d. It is straightforward to calculate $P_c(0, w, d, f|0) = (1-p)^w = (1-d/f(1-d))^w$. Thus, the probability to have two consecutive groups exclusively composed of set bits or unset bits is respectively $d(1-q)^{2(w-1)-1}$ and $(1-d)(1-p)^{2(w-1)-1}$. The expected number of compressed words in a WAH bitmap can thus be expressed as follows.

$$\begin{aligned}
\overline{W}_{WAH} &= M-(M-1)\Big[(1-d)P_c(0,2w-3,d,f|0)\\
&\qquad + dP_c(2w-3,2w-3,d,f|1)\Big]\\
&= M\Big[1-(1-d)(1-p)^{2w-3}-d(1-q)^{2w-3}\Big]\\
&\qquad + (1-d)(1-p)^{2w-3}+d(1-q)^{2w-3}
\end{aligned} \qquad (6)$$

For sparse bitmaps with $d << 1$, an upper bound for the number of words per set bit is to consider that almost all homogeneous groups are composed of unset bits. Furthermore, for $p = dq/(1-d) \to 0$ we can use a binomial decomposition; this is always true for $d << 1$ as $q <= 1$. Therefore, as stated in [16], for large values of M, the expected number of compressed words is as follows.

$$\begin{aligned}
\overline{W}_{WAH} &\approx M\Big[1-(1-d)\big(1-dq/(1-d)\big)^{2w-3}\Big]\\
&\approx M\Big[1-(1-d)\big(1-(2w-3)dq/(1-d)\big)\Big]\\
&= \frac{Ld}{w-1}\Big[1+(2w-3)q\Big]
\end{aligned} \qquad (7)$$

For bitmaps with a moderate clustering factor, $f < 5$, we have $\overline{W}_{WAH} \approx 2h/f$. Similarly, the expected number of words on a PLWAH bitmap is as follows.

$$\begin{aligned}
&\overline{W}_{PLWAH}\\
&= M-(M-1)\\
&\quad \times\Big[(1-d)(1-p)^{w-2}\sum_{k=0}^{s}P_c(k,w-1,d,f|0)\\
&\qquad + d(1-q)^{w-2}\sum_{k=0}^{s}P_c(w-1-k,w-1,d,f|1)\Big]
\end{aligned} \qquad (8)$$

Unlike when $k = 0$, for $1 < k < w - 1$, a recursive calculation is needed to calculate $P_c(k, w, d, f|0)$. Nonetheless, on a clustered bitmap, a lower bound for $P_c(k, w, d, f|0)$ is the probability to have k set bits forming an uninterrupted sequence of set bits. The probability of having a group full of unset bits followed by a group with k set bits in sequence is: $(1-d)p(1-q)^{k-1}(1-p)^{2w-4-k}\big[(w-k-1)q+(1-p)\big]$. Therefore, an upper bound for the expected number of words in an PLWAH bitmap is to consider only uninterrupted sequences of set bits. The upper bound for $d << 1$ can be expressed as follows.

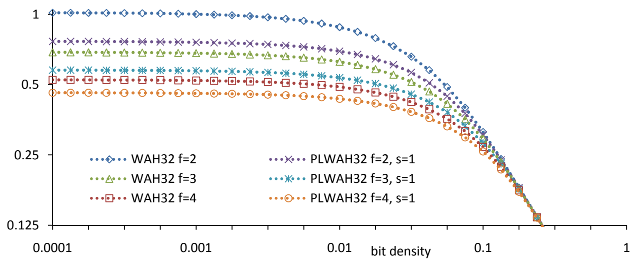
$$\begin{aligned}
&\lceil \overline{W}_{PLWAH} \rceil \\
&= M - (M-1) \\
&\quad \times \Big[(1-d)(1-p)^{2w-3} + d(1-q)^{2w-3} \\
&\qquad + \sum_{k=1}^{s} \Big((1-d)p(1-q)^{k-1}(1-p)^{2w-4-k} \\
&\qquad\qquad \times \big[(w-k-1)q + (1-p)\big] \Big) \\
&\qquad + \sum_{k=1}^{s} \Big(dq(1-p)^{k-1}(1-q)^{2w-4-k} \\
&\qquad\qquad \times \big[(w-k-1)p + (1-q)\big] \Big) \Big]
\end{aligned} \tag{9}$$

Furthermore, we can make the same assumption used in Equation 7; on a sparse bitmap with $d << 1$, almost all homogeneous groups are composed of unset bits. For $s = 1$, the expected number of words a PLWAH bitmap can be approximated as follows.

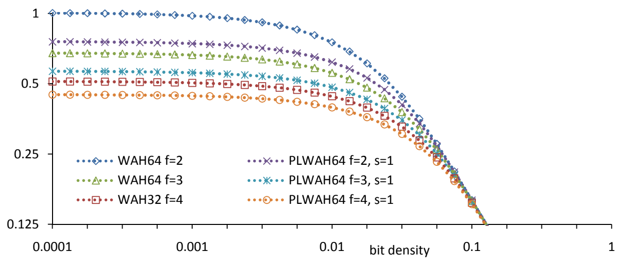
$$\begin{aligned}
W_{PLWAH} &\overset{s=1}{\approx} M\Big[1 - (1-d)(1-p)^{2w-3} \\
&\quad - (1-d)p(1-p)^{2w-5}\big((w-2)q + (1-p)\big)\Big] \\
&\overset{f<5}{\approx} Ld(2q - q^2) = \left(2 - \frac{1}{f}\right)\frac{h}{f}
\end{aligned} \tag{10}$$

Equation 10 represents the size in words of a PLWAH bitmap where the maximum number of sparse bit positions a fill word can contain is constrained to one. The number of sparse bits that can be "piggybacked" into a fill word depends on w and the length of the counter as explained in Section 3.1.

Figure 4 presents the average number of compressed words per set bit required by the WAH and PLWAH depending on the bit density for different values of the clustering factor. As established in Equation 10, the number of words in a PLWAH bitmap decreases as q decreases, i.e., as the clustering factor increases. We also observe that PLWAH compression always requires less words per set bit than WAH compression. In Figure 4(a), the ratio between the number of words in a sparse WAH bitmap and the corresponding PLWAH bitmap ranges from 1.32 to 1.13 for clustering factors varying from 2 to 4. The compression ratio on 32 bit and 64 bit word are identical. At higher bit densities, the bitmaps become incompressible and adopt a linear behavior. Finally, for very sparse bitmaps, the word length has little impact on the number of WAH and PLWAH words per set bit as shown by comparing the graphs from Figures 4(a) and 4(b). However, as the bitmaps get denser, they become incompressible and the number of words per set bit is thus inversely proportional to the word length. As the density approaches 1, the number of words becomes proportional to the number of unset bits, while the number of words per set bit tends to 0. For example, an uncompressed bitmaps of 1,000,000 bits with all bits set would be compressed to one word. Therefore, the ratio of words / bit set is: 1 / 1,000,000, which is close to 0.



(a) Clustered bitmap on 32 bit long words



(b) Clustered bitmap on 64 bit long words

Figure 4: Word count estimates on clustered bitmaps

3.4 The size of the position list in fill words

The size of the position list depends on the number of bits available in a fill word. A fill word contains two bits, a position list, and a counter. Increasing the size of the position list causes the total number of bits assigned to the counter to decrease. There is therefore a trade-off between the maximum number of groups that can be represented in a fill word, and the maximum number of heterogeneous bits in group that can be stored within its preceding fill word.

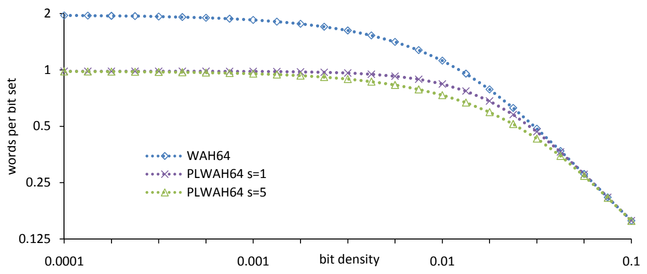
Each heterogeneous bit requires $\lceil \log_2 w \rceil$ bits to be stored in the position list. Let l be the number of bits taken by the position list and r the number of bits taken by the counter. The total number of bits in a fill word is : $w = 2 + l + r$. The maximum number of heterogeneous bits that can be stored in a fill word is: $s = \lfloor \frac{l}{\lceil \log_2 w \rceil} \rfloor = \lfloor \frac{w-2-r}{\lceil \log_2 w \rceil} \rfloor$.

In Equation 3, the cumulative distribution increases for increasing s. Similarly, all the terms in the $\sum^{s}$ in Equation 9 are positive. Thus, for both uniformly distributed and clustered bitmaps, W_{PLWAH} decreases when increasing s. Therefore, maximizing s minimizes W_{PLWAH}.

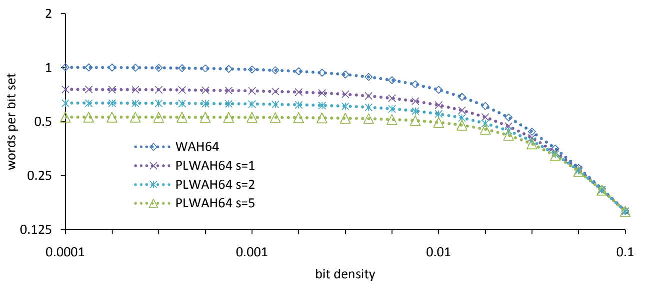
Figure 5 shows the effect of varying s for uniformly distributed and clustered bitmaps. There is very little benefit of increasing s on sparse uniformly distributed bitmaps as the probability of having multiple heterogeneous bits in a single group drops for low bit densities. However, bitmaps whose bit density is between the two linear zones contain fill words and literal words with a few set bits. For those bitmaps, increasing the value of s increases the probability of a fill word to be able to carry its following literal word. Thus, as s increases, the number of words per set bit decreases.

On a clustered bitmap, however, increasing s increases the compression ratio, such that $W_{PLWAH} = 0.5 * W_{WAH}$, i.e., all literal words with heterogeneous bits can be stored in their preceding fill word.

The length of an uncompressed bitmap in a bitmap index corresponds to the number of indexed elements. For $s = 1$, on a 32-bit CPU, PLWAH can represent $31 * 2^{25} > 1,000,000,000$ elements in a single fill word. For $s = 5$, on a 64-bit CPU, PLWAH can represent $63 * 2^{32} > 270,000,000,000$ elements in a single fill word. Almost any imaginable database application will thus only require a single fill word to capture homogeneous bit sequences.



(a) Uniformly distributed bitmap on 64 bit long words



(b) Clustered bitmap on 64 bit long words, f = 2

Figure 5: Word count estimates on clustered bitmaps

3.5 Compression of high-cardinality attributes

In this section, we discuss the compression of a attribute whose *random value* follows a uniform distribution, and the compression of a *clustered attribute* whose probabilities to change from one value to another depends on a Markov process as described in Section 3.3.

Let c be the cardinality of the random attribute. The attribute can thus be represented with c bitmaps. Since all the values have an equal probability of appearing, each bitmap is uniformly distributed with bit density: $d = 1/c$. For indexing attributes with high-cardinality the total estimated size in bits of the c WAH compressed bitmaps can be approximated as follows.

$$\text{size}_{WAH} \approx Lwc/(w-1) \times \left[1 - (1-1/c)^{2(w-1)} - 1/c^{2(w-1)}\right] \tag{11}$$

Similarly, the total estimated size in bits of the c PLWAH compressed bitmap is approximated as follows.

$$\text{size}_{PLWAH} \approx Lwc/(w-1) \times \left[1 - (1-1/c)^{2(w-1)} - (w-1)\frac{1}{c}(1-1/c)^{2w-3}\right] \tag{12}$$

When c is large, $(1/c)^{2w-2} \to 0$ and $(1-1/c)^{2w-2} \to (1-(2w-2)/c)$. The total size of the c WAH compressed bitmaps that compose the bitmap index for a uniformly distributed random attribute has the following asymptotic formula.

$$\text{size}_{WAH} = 2Lw \text{ bits} = 2L \text{ words} \tag{13}$$

Similarly, the total estimated size of the c PLWAH compressed bitmaps is as follows.

$$\text{size}_{PLWAH} = Lw \text{ bits} = L \text{ words} \tag{14}$$

The same reasoning holds for a clustered attribute where the probabilities are allowed to depend on another value. One such example is a simple uniform c-state Markov process: from any state, the Markov process has the same transition probability q to other states, and it selects one of the $c-1$ states with an equal probability. The total expected size in bits of the WAH compressed bitmaps necessary to index a clustered attribute is given by rewriting Equation 7 as follows.

$$\text{size}_{WAH} \approx \frac{Lwc}{w-1}\left(1-(1-1/c)\left(1-q/c(1-1/c)\right)^{2w-3}\right) \overset{f<5}{\approx} \frac{2h}{f} \text{ words} \tag{15}$$

Using Equation 9, the total expected size of a corresponding bitmap index compressed with PLWAH is as follows.

$$\text{size}_{PLWAH} \overset{s=1,f<5}{\approx} L(2q-q^2) = \left(2-\frac{1}{f}\right)\frac{hw}{f} \text{ words} \tag{16}$$

The bit density in each bitmap of a bitmap index of a uniformly distributed attribute is inversely proportional to the attribute cardinality. Thus, for high-cardinality attributes, the bit density is low for each bitmap of the bitmap index. The total size is proportional to the number of set bits. Similarly, for a high-cardinality attribute following a clustered distribution, the bitmaps of the bitmap index are sparse. The total size is proportional to the number of set bits divided by the clustering factor, i.e., a clustered attribute take less storage space. For example, an attribute following a distribution with a clustering factor $f = 2$ takes half the storage of a uniformly distributed bitmap when compressed with PLWAH on 64 bit long words and a position list of size 5. Additionally, equations 13, 14, 15, and 16 show that for both WAH and PLWAH, and for all attribute distributions, the total size of the bitmap is proportional to the size of the alignment, i.e., the CPU word length. For example, a bitmap index of a uniformly distributed attribute compressed using PLWAH with 32 bit alignment takes half the space of a WAH compressed one with 32 bit alignment, which in turn is the same size as a PLWAH compressed one with 64 bit alignment, which again is half the size of PAWAH compressed one with 64 bit alignment.

3.6 Adaptive Counter

A potential problem with the PLWAH compression technique for 32 bit words occurs when the number of elements significantly exceeds 10^9. In this case, the very long runs cannot be represented with a single fill word, as not enough bits are available to represent the counter value. Instead, for a run with a length of 10^{11}, a chain of approximately 100 fill words is needed. This issue could be resolved by using 64 bit words, but this would double the size of each word in the index (and the total index size), in order to solve a problem that only occurs very infrequently.

Instead, in such scenario, we propose to use an *adaptive counter* that basically uses 32 bit words when this is enough, and then *adapts* to 64 bit words *when necessary*, meaning that the longer words are only used for very long runs. This is achieved without changing the basic encoding scheme. A very long run is encoded with an empty position list in the (first) 32 bit (fill) word. The next word is also a fill word, and the total length of the run (using a 50 bit counter) will be encoded in two parts: the 25 LSBs of the length are put into the first word's counter part, while the 25 MSBs of the length are put into the second word's counter part. For the (extremely rare) case of 0-run followed immediately by a 1-run (or vice versa), a literal word is inserted in the middle.

Figure 6 proposes an example of PLWAH bitmap using the adaptive counter. The counter value is $2^{24-1} + 2^{17-1+25} = 2^{23} + 2^{31}$ and the position of the set bit in the following word is 4. In its uncompressed form, the bitmap is thus composed of $(2^{23} + 2^{31}) * 32 + 3$ unset bits followed by a set bit.

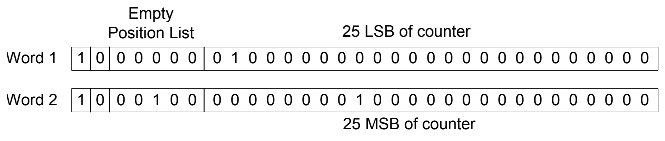


Figure 6: Example of PLWAH adaptive counter

4. BITWISE OPERATIONS

Logical bitwise operations are crucial operations for swiftly carrying out range queries and queries with multiple predicates using bitmap indexes. For this reason, we next examine the complexity of performing bitwise logical operations on PLWAH compressed bitmaps. In this section, we show that the time for completing an arbitrary logical operation between two PLWAH compressed bitmaps is proportional to the total size of the two compressed bitmaps, while when the two operands are in their uncompressed form and assuming all the bitmaps have the same size, the time required is constant. Finally, we show that the time to perform a logical operation between a decompressed bitmap and a PLWAH compressed one is proportional to the size of the compressed bitmap.

4.1 Operations on 2 compressed bitmaps

A bitwise operation on two compressed bitmaps is performed by sequentially scanning each input bitmap, compressed word by compressed word. The complexity of bitwise AND/OR operations are quite similar, but ANDs are somewhat faster since the resulting bitmap has less set bits than the operands (as opposed to ORs, where the opposite is true). Thus, for simplicity, we focus on the logical OR operation on two compressed bitmaps (the hardest of the two). However, the methods are very easily adapted to logical ANDs. Details on the OR implementation are found in Algorithm 4. Additionally, three auxiliary procedures are shown: *ReadWord*, *AppendFill*, and *AppendLit*.

The *ReadWord* procedure (Algorithm 1) is called each time a compressed word has to be decompressed. The decompression requires very few CPU cycles and has only one branch[2]. If the word is a fill word, it loads the corresponding fill group, adjusts the counter, and transforms the position list into a bitmap. Both the load of the data corresponding to the fill word type and the transformation from a position list to a bitmap are done by multiplication and bit shift operations and do not require any branching. If the word is a literal word, the data is loaded by masking out the word's MSB.

Algorithm 1: Reads a compressed word and updates the run

```
    ReadWord ( Compressed word $W$ )
 1: if $W$ is a fill word then
 2:     $data \leftarrow ((W >> 62)\&1) * all\_ones$
 3:     $nWords \leftarrow W\&counter\_mask$
 4:     $isFill \leftarrow true$
 5:     $isSparse \leftarrow (W\&position\_mask)! = 0$
 6:     $sparse \leftarrow$ bitmap constructed from the position list
 7: else
 8:     $data \leftarrow W\&first\_bit\_unset$          $\triangleright$ MSB of $W$ is unset
 9:     $nWords \leftarrow 1$
10:     $isFill \leftarrow false$
11:     $isSparse \leftarrow false$
```

The *AppendFill* procedure (Algorithm 2) generates a fill word corresponding to the provided fill word type and counter. It has one conditional branch.

Finally, the *AppendLit* procedure (Algorithm 3) transforms a literal group into a PLWAH encoded word. It has up to 4 conditional branches. It performs two interesting operations, namely the count of the number of set bits and the transformation of a bitmap into a position list. Both operations can be efficiently performed using the new instructions at hand on recent CPU architectures. Counting the number of set bits can be performed directly using the *population count* instruction, part of the SSE4a instruction set available, for example, on the AMD "10h" [1] and the Intel "Core i7" [7] processor families. For older architectures, the most efficient alternative is probably to count the number of times the LSB of the bitmap can be removed before obtaining a bitmap with all bits unset. In practice, one can limit the count to a threshold corresponding to the maximum size the position list can reach. Generating the list of the positions of the set bits in a bitmap is performed by locating the position of the LSB, removing the LSB, and repeating the process until all bits are unset. Many techniques to find the position of the LSB in a word exist in the literature [9]. However, the *bit scan forward* instruction, available for 32 and 64-bit words on modern CPU architectures, e.g., "Pentium 4", "AMD K8", and above, is by far the fastest approach of tackling the task.

Algorithm 2: Appends a fill word to a compressed bitmap

```
    AppendFill(bitmap $C$, word $fill$, int $count$)
    $last$ is the last word of the bitmap
 1: if $last$ is same type as $fill$ and $last$ position list is empty then
 2:     $count$ is added to the counter of $last$
 3: else
 4:     Append $fill$ to $bitmap$
```

Algorithm 3: Appends a literal word to a compressed bitmap

```
    AppendLit(Compressed bitmap $C$, literal word $L$)
    $last$ is the last word of $C$
 1: if $L = 0$ then
 2:     $AppendFill(C, 0, 1)$
 3: else if $last$ is counter and has empty position list then
 4:     if $last$ is a zero fill then
 5:         if $L$ is sparse then                          $\triangleright$ $popcount$ CPU
 6:             Generate position list for a zero fill   $\triangleright$ $bitscan$ CPU
 7:             Place position list into $last$
 8:         else
 9:             Append $L$ to $C$
10:     else if $L$ is sparse then                         $\triangleright$ $popcount$ CPU
11:         Generate position list for a one fill        $\triangleright$ $bitscan$ CPU
12:         Place position list into $last$
13:     else
14:         Append $L$ to $C$
15: else
16:     Append $L$ to $C$
```

The *CCOR*, procedure (Algorithm 4) performs a bitwise OR on two compressed bitmaps. Its total execution time is dominated by the number of iterations through the main loop. Each loop iteration consumes a fill word or a literal word from either one or both bitmap operands. Let W_x and W_y, respectively, be the number of words of each operand, and let M_x and M_y denote the number of words in their decompressed form, i.e., the number of groups. If each iteration consumes a word from both operands, $\min(W_x; W_y)$ iterations will be required. If each iteration consumes only one word from either operand, $W_x + W_y$ iterations may be required. Since each iteration produces at least one group and the result contains at most $\min(M_x, M_y)$ groups, the main loop requires at most $min(M_x, M_y)$ iterations[3]. The number of iterations through the

[2]Reducing branching is important for keeping the CPU instruction cache full and avoiding stalls.

[3]Our implementation allows to perform bitwise operations using bitmaps of different uncompressed sizes. In the case of an OR, the remaining part of the longest bitmap is appended to the result.

Algorithm 4: Performs a bitwise OR on 2 compressed bitmaps

```
    CCOR(Compressed bitmap $X$, Compressed bitmap $Y$)
    $xrun$ holds the current run of $X$
    $yrun$ holds the current run of $Y$
    $Z$ is the resulting compressed bitmap
 1: Allocate memory for $Z$
 2: while $xrun.it < size(X)$ and $yrun.it < size(Y)$ do
 3:     if $xrun.nWords = 0$ then $Readword(X, xrun.it)$
 4:     if $yrun.nWords = 0$ then $Readword(Y, yrun.it)$
 5:     if $xrun.isFill$ and $yrun.isFill$ then
 6:         $nWords \leftarrow min(xrun.nWords, yrun.nWords)$
 7:         $AppendFill(Z, xrun.data|yrun.data, nWords)$
 8:         decrease $xrun.nWords$ by $nWords$
 9:         decrease $yrun.nWords$ by $nWords$
10:     else
11:         $AppendLit(Z, xrun.data|yrun.data)$
        ▷ If bitmaps position lists are empty, ignore the remaining
12:     if ($xrun.nWords = 0$ and $xrun.isSparse$) or
                ($yrun.nWords = 0$ and $yrun.isSparse$) then
13:         if $xrun.nWords = 0$ and $xrun.isSparse$ then
14:             Load $xrun$ sparse data
15:         if $yrun.nWords = 0$ and $yrun.isSparse$ then
16:             Load $yrun$ sparse data
17:         if $xrun.nWords > 0$ and $yrun.nWords > 0$ then
18:             $AppendLit(Z, xrun.data|yrun.data)$
19:             Decrement $xrun.nWord$
20:             Decrement $yrun.nWord$
21:             if $xrun.nWords = 0$ and $xrun.isSparse$ then
22:                 Load $xrun$ sparse data
23:             else if $yrun.nWords = 0$ and $yrun.isSparse$
    then
24:                 Load $yrun$ sparse data
25:     if $xrun.nWords = 0$ then Increment $xrun.it$
26:     if $yrun.nWords = 0$ then Increment $yrun.it$
27: Append the remaining of the longest bitmap to $Z$
28: Return $Z$
```

loop I satisfies the following conditions:

$$\min(W_x; W_y) < I < \min\left((W_x + W_y), \min(M_x, M_y)\right) \quad (17)$$

When the operands are two sparse bitmaps, each word is a fill word with a non-empty position list. In that case, each operation only consumes one word from one of the operands. Therefore, it takes $W_x + W_y$ iterations to complete the bitwise OR, as each loop iteration executes *AppendFill* and *AppendLit* once. Compared to a WAH bitmap where each set bit requires on average two words, half the number of loop iterations are thus required. In the case of WAH bitmaps, every two iterations, the *AppendFill* procedure is called followed by, in the next iteration, a call to the *AppendLit* procedure. Thus, the total number of calls to the *AppendFill* and *AppendLit* procedures to perform an OR operation on two PLWAH bitmaps and two WAH bitmaps are equal.

The time complexity for performing a logical OR on two compressed bitmaps, T_{CC}, mainly depends on four terms: the time to perform one memory allocation T_a, the time to perform I decompressions T_d, the time to perform I *AppendFill* T_f, and the time to perform I *AppendLit* T_l. Let C_d, C_f, and C_l respectively be the time to invoke each of these operations. As established in Equation 17, $(W_x + W_y)$ is an upper bound for I. In common memory management libraries, the time for memory allocation *a*nd initialization is less than proportional to the size of the memory allocated [6]. Let C_a be the time to allocate one word, we thus have $T_a < C_a(W_x+W_y)$. An upper bound for the total time complexity is as follows.

$$\begin{aligned} T_{CC} &= T_a + T_l + T_f + T_d \\ &< (C_a + C_l + C_f + C_d)\,(W_x + W_y) \end{aligned} \quad (18)$$

The complexity of performing an OR operation on two compressed bitmap is $O(W_x + W_y)$.

4.2 In-place operations

The most time expensive operations to perform a logical OR between two compressed bitmaps are in decreasing order: the memory allocation, the addition of a literal word, the addition of a fill word, and the decompression of a word. If an OR operation is executed on many bitmaps, the memory management dominates the total execution time. However, logical bitmap operations such as OR or AND are frequently used on a large number of sparse bitmaps in order, for example, to answer range queries or combine predicates. A common approach to reduce the memory management cost is to use an "in-place" operator that recycles the memory allocated for one of the operands to store the result, thus eliminating expensive memory allocation. For example, instead of allocating memory for z, and performing $z = x$ OR y, the "in-place OR" does $x = x$ OR y.

The PLWAH in-place OR takes one uncompressed and a compressed bitmap as operand. Using an uncompressed operand ensures that there is never more input than output words, so that no result word is overwriting a future input word. The uncompressed bitmap can thus be used both as input and output. In addition to avoiding repeated allocation of new memory for the intermediate results, it also removes the need to compress temporary results. The time complexity is thus mainly dependent on the time for initially allocating memory for the storage of the uncompressed bitmap and to perform I decompressions. The in-place OR operation is performed by the UCOR function whose details are supplied in Algorithm 5. UCOR is composed of a main loop that iterates through each word of the compressed operand.

Algorithm 5: Performs a bitwise *OR* on an uncompressed bitmap and a compressed bitmap

```
    UCOR(Uncompressed bitmap $U$, Compressed bitmap $C$)
 1: for all $words$ in $C$ do
 2:     if $run.data$ = 0 then
 3:         move forward of $run.nWords$ words in $U$
 4:     else
 5:         while $nWords$ do
 6:             $nWords \leftarrow nWords - 1$
 7:             Next word in $U \leftarrow |run.data$    ▷ "|" is a bitwise
    OR
 8:     if $run.isSparse$ then
 9:         Next word in $U \leftarrow |run.sparse$ ▷ "|" is a bitwise OR
```

Let x and y, respectively, be the uncompressed and compressed operand bitmaps. The number of words in x is M_x, the number of compressed words in y is W_y. Let I be the number of iteration in the main loop, each iteration of the loop treats one compressed word, therefore there are $I = W_y$ iterations. The time to allocate memory for storing x, T_x, is at least proportional to M_x. Thus, $T_a < C_aM_x$. Let C_i be the time to process one iteration, the time to process the whole loop is proportional to C_iI. An upper bound for the total time spent to perform an in-place OR T_{UC}, is therefore:

$$T_{UC} = C_aM_x + C_iW_y \quad (19)$$

The UCOR has three conditional branches including one conditional branch for decompressing a word. In comparison, the in-place OR algorithm has a total of two conditional branches per compressed word, but the number of compressed words in a WAH bitmap is always higher than the number of words found in the corresponding PLWAH bitmap. As explained in Section 3, the number of words in a PLWAH bitmap tends to be half the number of words in a WAH bitmap. Executing the UCOR thus represents a net performance gain for sparse bitmaps.

Equations 18 and 19 show that the fastest procedure depends on the size of the uncompressed bitmap and the size of the compressed bitmap, i.e., the compression ratio. Let r be the number of logical bitwise OR operations performed to answer a range query. The total time to perform the logical OR operations on compressed bitmaps to answer a range query is as follows.

$$\begin{aligned} T_{CC,r} &\approx r\,(C_a + C_l + C_f + C_d)\,(W_x + W_y) \\ T_{UC,r} &\approx C_a M_x + r(C_i W_y) \end{aligned} \qquad (20)$$

The in-place benefits from a lower number of iterations in its main loop. In addition, the complexity of the main loop is significantly reduced compared to the OR operation on two compressed bitmaps. However, the in-place OR procedure suffers from a higher startup cost due to the initial memory allocation corresponding to the size of an uncompressed bitmap. Since, in a bitmap index, the size of the uncompressed bitmap is proportional to the size of the dataset, the fastest procedure to perform a range query depends on the size of the dataset and the range of the query.

5. EXPERIMENTS

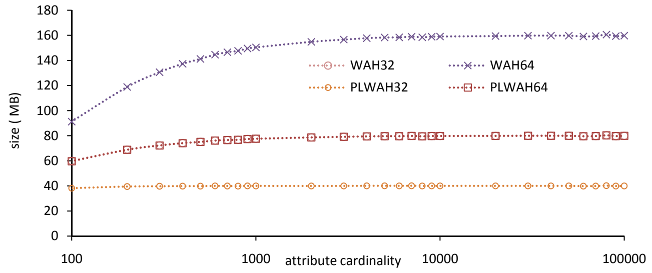
In this section, we present the timing results from a Dell Dimension 9300 equipped with a Intel "Core 2 6700 2.66GHz" CPU running FreeBSD 7.1 64-bit OS which relies on the "jemalloc" [6] memory allocation library. All the presented experiments are performed within a PostgreSQL 8.3 DBMS. The implementation is in C and runs as a loadable module for PostgreSQL, thus making the code usable by a wide audience.

The experiments are conducted on both synthetic and real data sets. The generated data is composed of a set of *(key, attribute)* pairs. The attribute follows either a uniform distribution, or a clustered distribution as presented in Section 3.5. For each, a comparative study of the influence of the distribution parameters on the size and performance of the WAH and PLWAH indexes is conducted. The present study shows that PLWAH compression is superior to WAH compression. Furthermore, PLWAH performs "in-place OR" operations faster than WAH. For long range queries, the speedup is up to 20%. As the performance-wise superiority of WAH over BBC is already shown in previous work [16], we only compare PLWAH with WAH. The indexes are implicitly cached in memory by Postgresql as previous research has demonstrated that compressed bitmap indexes were CPU bounded [16].

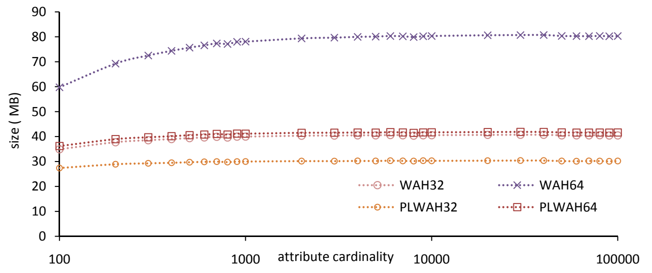
Finally, experiments are conducted on a real data set composed of 15,000,000 music segments. Each music segment is described using 15 attributes, each with a granularity of 100,000. The index keys are composed of the attribute identifier and the attribute value and are in turn indexed using the B-tree index available in PostgreSQL. The index thus has $15 \times 100,000$ entries and compressed bitmaps. Each compressed bitmap is less than 2KB in size and thus fits in L1 cache and benefits from CPU optimizations related to cache locality.

5.1 Bitmap index size

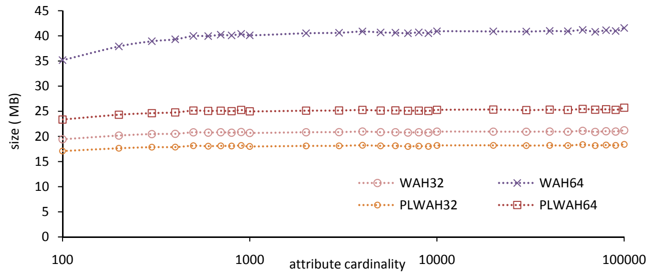
The sizes of the bitmap indexes for the various data distributions discussed earlier are presented in Figure 7. Figure 7(a) confirms that for uniformly distributed high-cardinality attributes, the size of PLWAH tends to be half the size of WAH for a given word length,



(a) Size of bitmap index for a uniformly distributed attribute



(b) Size of bitmap index for a clustered attribute, $f = 2$



(c) Size of bitmap index for a clustered attribute, $f = 4$

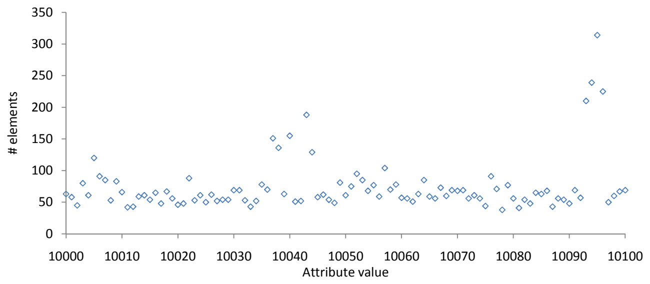
Figure 7: Size comparison between WAH and PLWAH bitmap indexes (indexed elements: 10,000,000)

in accordance with the estimates in Equations 13 and 14. Hence, the curves of PLWAH64 and WAH32 overlap (the WAH32 curve is actually hidden below the PLWAH64 curve).

The size of bitmap indexes on clustered attributes are shown for different clustering factors in Figures 7(b) and 7(c). Again, PLWAH outperforms WAH on compression ratio. PLWAH64 provides significant benefits due to its longer position list. For short position lists, a higher clustering factor decreases the probability for PLWAH to "piggyback" the next literal word in the current fill word.

The histograms of the music data set vary greatly from value to value. Low values tend to be very frequent, e.g., 30% of the elements have 0 as value for the first extracted feature. For frequent values, the four compression schemes converge; the bitmaps are too dense to be compressed and the bitmaps are mainly composed of literal words. For less frequent values, the bitmaps can be compressed. As illustrated in Figures 8(a) and 8(b), PLWAH32 tends to be half the size of WAH32, PLWAH64 half the size of WAH64 and very similar to the size of WAH32. A comparison of the total index size between each compression scheme for the reviewed data distribution is presented in Table 1. All the bitmap indexes have a size proportional to the number of values.

Finally, evidences of the influence of the position list size are uncovered in Figures 9(a) and 9(b). For uniformly distributed attributes, the probabilities of having multiple set bits in a single literal word are very low and the position list tends to contain only one set bit position. The gain of increasing the size of the posi-


(a) Histogram for values from 10,000 to 10,100 of the first music feature attribute

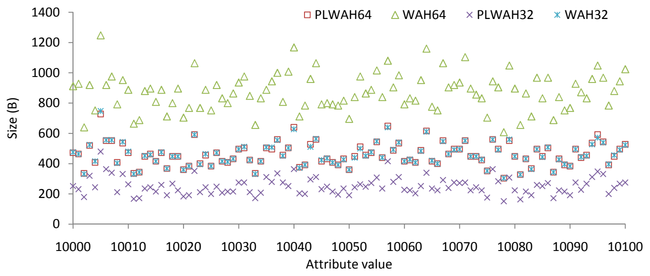
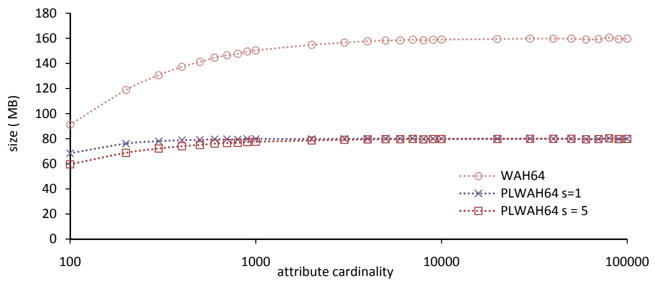

(b) Size of bitmaps corresponding to values 10,000 to 10,100 of the first music feature attribute

Figure 8: Bitmap sizes of the first music feature attribute


(a) Size of bitmap index for a uniformly distributed attribute

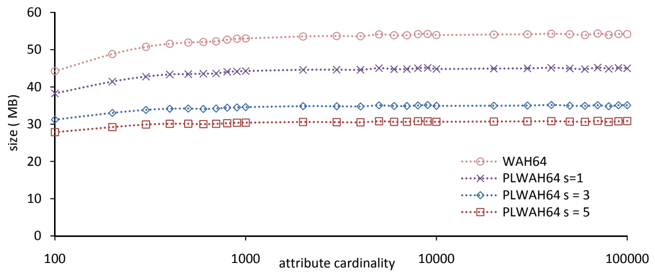

(b) Size of bitmap index for a clustered attribute, $f = 3$

Figure 9: Size comparison between PLWAH bitmap indexes with different position list sizes (indexed elements: 10,000,000)

tion list is thus marginal. The position list size proves to have an important impact on the compression ratio for clustered attributes. Indeed, for clustered attributes, the probability of having more than one set bit in a literal word increase with the clustering factor. Extending the position list allows more literal words to be encoded within their preceding fill word.

5.2 Performance study

In the following, we study the performance of the OR operator for answering range queries. A similar study can be conducted using other bitwise operators, however, (1) OR operations are more complicated to handle for the compression scheme as the bitmaps become denser, this is not always the case, e.g., with the AND operator; (2) long series of OR operations are often required to perform range queries, while the number of AND operations, for example required to treat a multi-dimensional range query, tends to be much smaller. Range queries are generated by choosing a range of a given size randomly within the interval $[0; 100,000]$.

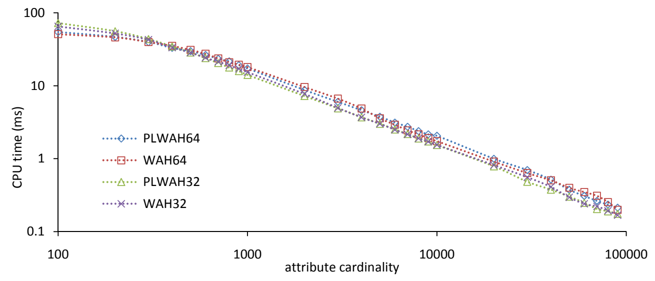
The total CPU time of performing range queries depending on the range is presented next. In the following experiments, the bitmaps are aggregated sequentially since this is the most revealing for the basic performance difference between the compression schemas. Other aggregation schemes are of course possible, for example, the aggregation could be performed by arranging the operation as a balanced tree. However, such schemes require more intermediate memory allocation, etc., something which blurs the basic performance difference between compression schemes. For a balanced tree-based scheme, there would be $\log r$ operations of complexity $O(r)$, thus leading to an overall $O(r \log r)$ complexity. Even in more complex scenarios, involving series of different logical operation on the bitmaps, the optimal ordering of the operations involved in the aggregation can be easily calculated, as discussed in [2]. The complexity of performing a single operation and the aggregation scheme are thus orthogonal, so our choice of sequential aggregation does not distort the results.

The performance of range queries on short random intervals comprised of 10 values is shown in Figures 10(a) and 10(b). Here, since the number of operation is small, the aggregation scheme has little influence on the query performance. When the attribute cardinality increases, i.e., the density decreases, the compression ratio increases, and the number of words per bitmap decreases. Hence, the CPU time required to process a constant number of bitmaps decreases. For both algorithms, the query time is very similar for PLWAH32 versus WAH32, and for PLWAH64 versus WAH64. However, for the UCOR algorithm, the 64 bit compression schemes perform up to 20% faster than their 32 bit counterparts.

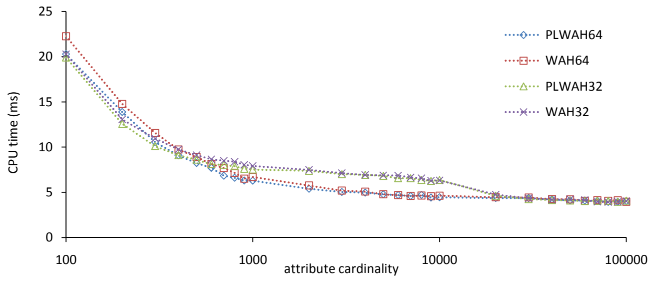
The query performance for different range lengths using a sequential aggregation scheme is studied in the following. In a bitmap index, each possible value of the indexed attribute is represented with a bitmap; exactly one bit is set at any given position across all the bitmaps. Thus, when performing an OR operation, the number of set bits in the resulting bitmap is the total number of set bits in each operand bitmap. On a uniformly distributed attribute, the bit density per bitmap is constant and since all bitmaps have the same uncompressed length, the expected number of set bits in each bitmap h is constant. Each OR operation thus increases the number of set bits approximately by h. After i OR operations, the result has ih set bits. Therefore, as explained by Equations 13 and 14, on very sparse bitmaps the length of the resulting WAH and PLWAH bitmaps after i OR operations are, respectively, $2ih$ and ih words long. Since, on sparse PLWAH bitmaps, each loop iteration produces one fill word with a non-empty position list, the total number of iteration to produce the i^{th} result is ih. Similarly, on WAH bitmaps, each loop iteration produces one word, thus, in total, $2ih$ iterations are required to produce the i^{th} result. To answer a range query, $r - 1$ OR operations are performed, thus $\sum_i^r ih = hr(r-1)/2$ and $\sum_i^r ih = hr(r-1)$ loop iterations are run. The complexity of performing a range query using the CCOR procedures on two compressed bitmaps is thus quadratic in the range while aggregating the bitmaps in a sequential fashion.

Data set	PLWAH64	WAH64	PLWAH32	WAH32
Uniform	86 MB	177 MB	43 MB	86 MB
Clustered, $f = 2$	48 MB	88 MB	36 MB	46 MB
Clustered, $f = 3$	37 MB	60 MB	28 MB	33 MB
Clustered, $f = 4$	31 MB	47 MB	24 MB	27 MB
15 music att.	926 MB	1617 MB	565 MB	926 MB

Table 1: The size of the bitmap indexes for a common attribute cardinality of 100,000

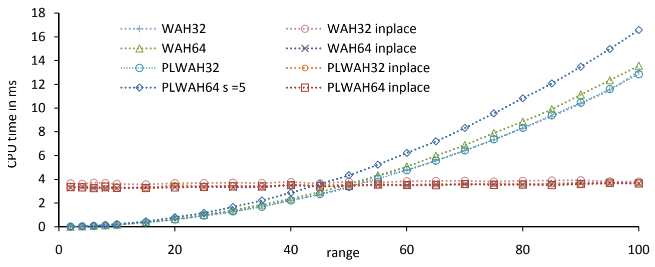


(a) CPU time to perform CCOR on 10 bitmaps

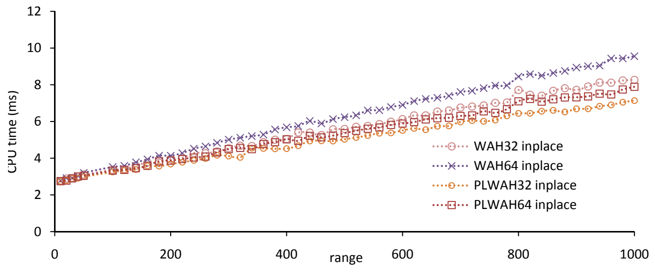


(b) CPU time to perform UCOR on 10 bitmaps

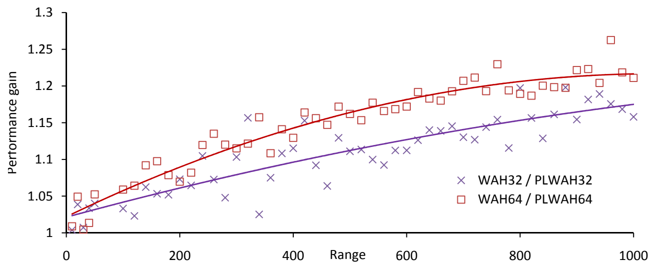
Figure 10: Performance impact of the attribute cardinality (10,000,000 elements, uniformly distributed attribute)



(a) Total query time using CCOR



(b) Total query time using UCOR



(c) Query time ratio WAH / PLWAH

Figure 11: Performance on uniformly distributed attribute (elements: 10,000,000, cardinality: 100,000)

However, as discussed earlier, this simple aggregation scheme allows a straightforward comparison the WAH and PLWAH.

The CCOR procedure offers good response time for queries on short ranges as show in Figure 11(a). Experiments show no significant performance drop for ranges covering less than 20 attribute values. In fact, PLWAH32 and WAH32 performance are so similar that their performance curves overlap. For larger ranges, the performance of both WAH and PLWAH drops. For those large ranges, the WAH performance tends to be slightly better than the PLWAH due to the more complex AppendLit procedure present in PLWAH. However, these observations for larger range queries are not relevant, as for both algorithms, the “in-place OR” procedures prove to be much faster.

As illustrated in Figure 11(b), large range queries are very well handled by the UCOR procedure whose time complexity is linear in the size of the range. However, the UCOR procedure suffers from a high start-up cost mainly due to the initial memory allocation. For short range queries, the CCOR procedure is thus preferred. The maximum size of the position list does not change the complexity of the decompression or the management of a sparse literal; for clarity purposes, Figure 11(b) only shows PLWAH64 with a position list of size 5, and PLWAH32 with a position list of size 1. Both the WAH and PLWAH complexities depend on the size of the compressed operand bitmap. The small additional complexity of the decompression and the accounting of sparse literal words do not handicap the performance of PLWAH. On the contrary, PLWAH is more efficient for executing the UCOR procedure as only half the number of loop iterations are required. As shown in Figure 11(c), the ratios between the CPU time of WAH and PLWAH increase up to 20% as the range increases. The chaotic start is due to the high influence of the initial memory allocation. Better memory management, planned as future work, would further improve the ratio.

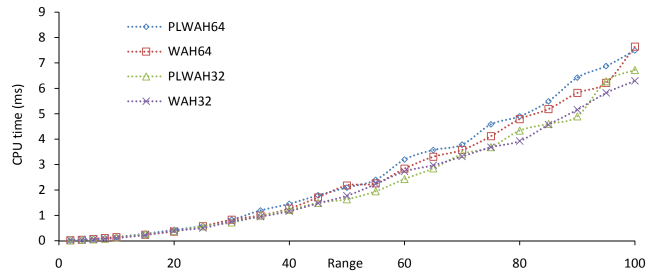
Figures 12(a) and 12(b) show similar results on the music data set. The noteworthy higher starting cost of the in-place OR operator on the music data set is due to the larger number of elements: the uncompressed bitmaps are 50% longer due to the 50% increase in the number of elements. However, the flatter slope after the startup overhead is due to the smaller size of the compressed bitmaps; bitmaps of uniformly distributed attributes are the hardest to compress, hence better compression ratios are obtained on the music data. For the same reason, a flatter slope is also noticeable for the OR on compressed bitmaps.

Finally, the impact of the position list size on the performance of the CCOR operator is shown in Figure 13. The computation of the position list is directly proportional to its size. As the size of the position list increases, a small overhead can be observed for long range queries for which better performance is achieved using the UCOR algorithm. For short range queries, no significant overhead is observed. The decompression time is independent of the size of the position list. Hence, the performance of the UCOR operation is not directly influenced by of the size of the position list.

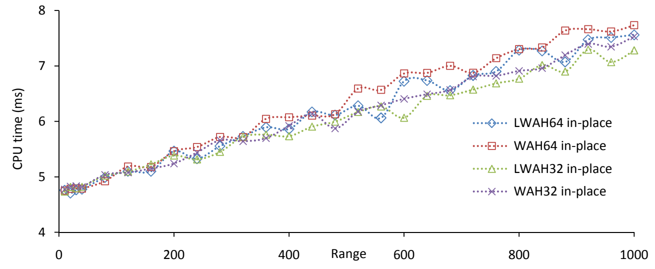
In summary, the experiments confirm the analytical estimates in Equations 13 and 14: a uniformly distributed attribute indexed with PLWAH bitmaps takes half the size it would require using WAH. Furthermore, we have measured the performance impact of three factors, namely the attribute cardinality, the range, and the size of the position list. For long range queries PLWAH shows a significant performance improvement. For short range queries, PLWAH and WAH efficiencies are equivalent.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we present PLWAH, a bitmap compression scheme that compresses sparse bitmaps better and answers queries on long ranges faster than WAH, which was so far recognized as the most efficient bitmap compression scheme for high-cardinality attributes. The results are verified through detailed analytical and experimental approaches. The storage gain essentially varies depending on



(a) Total CPU time using CCOR on the music attributes



(b) Total CPU time using UCOR on the music attributes

Figure 12: Performance comparison between the WAH and PLWAH OR operators on the music attributes

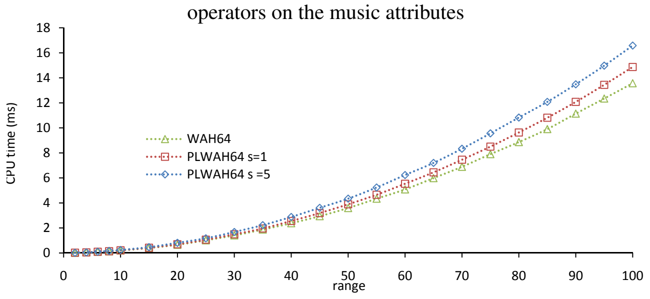


Figure 13: Performance impact of the size of the position list on a uniformly distributed attribute

the following parameters: the size of the data set, the attribute distribution, the attribute cardinality, and the word length.

For uniformly distributed high-cardinality attributes, we both prove and observe that the compression ratio is twice as good as for WAH. For real data, the size of PLWAH compressed bitmaps varies between 57% and 61% of the size of WAH compressed bitmaps. In terms of performance, PLWAH and WAH are comparable for short range queries. However, for long range queries, PLWAH is up 20% faster than WAH, depending on the data distribution.

Future work encompasses studying the performance impact of PLWAH used with complementary bitmap indexing strategies, and collecting empirical storage and performance results from different data sets and CPU instruction sets. Furthermore, the current implementation would benefit from making use of multi-core CPU architecture and dividing long bitmaps into chunks to ensure they fit into the L1 cache. Additionally, new techniques, e.g., posting list and batched updates, need to be developed in order to improve the update performance of the index. Promising research directions include the development of a dedicated bitmap materialization engine that would build aggregated bitmaps based on query patterns and frequencies, and data distribution. The index would then be able to select the most efficient aggregation path, the required bitwise operations, and the type of bitwise algorithm to use.

7. REFERENCES

[1] Advanced Micro Devices. *Software Optimization Guide for AMD Family 10h Processors*, November 2008.

[2] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, pages 329–338, 2000.

[3] G. Antoshenkov and M. Ziauddin. Query processing and optimization in ORACLE RDB. *VLDB Journal*, 5(4):229–237, 1996.

[4] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A. S. Tosun. Approximate Encoding for direct access and query processing over compressed bitmaps. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, pages 846–857, 2006.

[5] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *SIGMOD Rec.*, 27(2):355–366, 1998.

[6] J. Evans. *A Scalable Concurrent malloc Implementation for FreeBSD*. FreeBSD, April 2006.

[7] Intel. *Intel SSE4 Programming Reference*, July 2007.

[8] N. Koudas. Space efficient bitmap indexing. In *Proc. of ACM Conf. on Information and Knowledge Management (CIKM)*, 2000.

[9] C. E. Leiserson, H. Prokop, and K. H. Randall. Using de Bruijn sequences to index a 1 in a computer word, 1998. Available at http://supertech.csail.mit.edu/papers.html.

[10] P. O'Neil. Model 204 architecture and performance. In *Proc. of Workshop in High Performance Transaction Systems*, 1987.

[11] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. of ACM Int. Conf. on Management of Data (SIGMOD)*, pages 38–49, May 1997.

[12] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. *SIGMOD Rec.*, 30(2):47–57, 2001.

[13] D. Rotem, K. Stockinger, and K. Wu. Optimizing candidate check costs for bitmap indices. In *Proc. of ACM Conf. on Information and Knowledge Management (CIKM)*, 2005.

[14] M. Stabno and R. Wrembel. RLH: Bitmap compression technique based on run-length and Huffman encoding. *To appear in Information Systems*, 2009.

[15] K. Stockinger and K. Wu. *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, chapter Bitmap Indices for Data Warehouses, pages 157–178. 2007.

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

[17] K. Wu and P. Yu. Range-based bitmap indexing for high cardinality attributes with skew. In *Proc. of Int. Computer Software and Applications Conf. (COMPSAC)*, pages 61 – 67, 1998.

[18] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for Data Warehouses. In *Proc. of the Int. Conf. on Data Engineering, (ICDE)*, pages 220–230, 1998.

[19] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. of the Int. Conf. on Data Engineering, (ICDE)*, pages 59–71, 2006.

Acknowledgments

This project is funded by the Danish Research Council for Technology and Production Sciences, through the framework project "Intelligent Sound", grant no. 26-04-0092.