Information Systems 62 (2016) 155–168

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

SBH: Super byte-aligned hybrid bitmap compression

Sangchul Kim, Junhee Lee, Srinivasa Rao Satti*, Bongki Moon*

Department of Computer Science and Engineering, Seoul National University, Seoul 08826, Republic of Korea

ARTICLE INFO

Article history:
Received 9 February 2016
Received in revised form
24 June 2016
Accepted 8 July 2016
Recommended by: D. Shasha
Available online 16 July 2016

Keywords:
Database Indexing
Bitmap Index
Bitmap compression
Byte-based Bitmap Code
Word-Aligned Hybrid

ABSTRACT

Bitmap indexes are commonly used in data warehousing applications such as on-line analytic processing (OLAP). Storing the bitmaps in compressed form has been shown to be effective not only for low cardinality attributes, as conventional wisdom would suggest, but also for high cardinality attributes. Compressed bitmap indexes, such as *Byte-aligned Bitmap Compression* (BBC), *Word-Aligned Hybrid* (WAH) and several of their variants have been shown to be efficient in terms of both time and space, compared to traditional database indexes. In this paper, we propose a new technique for compressed bitmap indexing, called *Super Byte-aligned Hybrid* (SBH) bitmap compression, which improves upon the current state-of-the-art compression schemes. In our empirical evaluation, the query processing time of SBH was about five times faster than that of WAH, while the size of its compressed bitmap indexes was retained nearly close to that of BBC.

1. Introduction

Many enterprise applications generate a massive volume of data through logging transactions or collecting sensed measurements, which brings up the need for effective indexing methods for efficient storage and retrieval of data. To support some of the retrieval queries efficiently, database management systems make use of indexes such as B-trees and bitmap indexes [14,18]. The index schemes based on B-tree are efficient for a wide variety of data, since they support both searches and updates with nearly the same complexity. However, for most data warehousing applications, search operations are more frequent than updates. For such applications, bitmap indexes may improve the overall performance. Besides, when the number of unique values of an attribute is small (e.g., gender), one can achieve much better performance than B-trees by using bitmap indexes.

A bitmap index is a collection of bit vectors created for each distinct value in the indexed column. The number of bit vectors in a bitmap index for a given indexed column, also referred to as *cardinality*, is equal to the number of distinct values that appear in the indexed column. For a column with cardinality m, the ith bit vector corresponds to the ith distinct value (in some order) and the jth bit in the ith bit vector is set to 1 if and only if the value of the jth element in the column is equal to i. Thus, if an indexed column of n elements has cardinality m, then its bitmap index contains m bit vectors of length n each, and hence uses a total of mn bits.

Fig. 1 shows a set of bitmaps for an attribute `City`. The cardinality of `City` is four. Each bitmap represents whether the value of `City` is one of the four available values. Suppose we process an SQL query below.

```
SELECT*
FROM T
WHERE City = Seoul OR City = London
```

The main operation performed in the bitmaps is a bitwise operation $B_{Seoul} \vee B_{London}$, where B_c is a bitmap corresponding to city c. Standard selection queries on bitmap indexes can be

* Corresponding authors.
E-mail addresses: stdio@snu.ac.kr (S. Kim), jvl@tcs.snu.ac.kr (J. Lee), ssrao@snu.ac.kr (S.R. Satti), bkmoon@snu.ac.kr (B. Moon).

http://dx.doi.org/10.1016/j.is.2016.07.004
0306-4379/© 2016 Elsevier Ltd. All rights reserved.

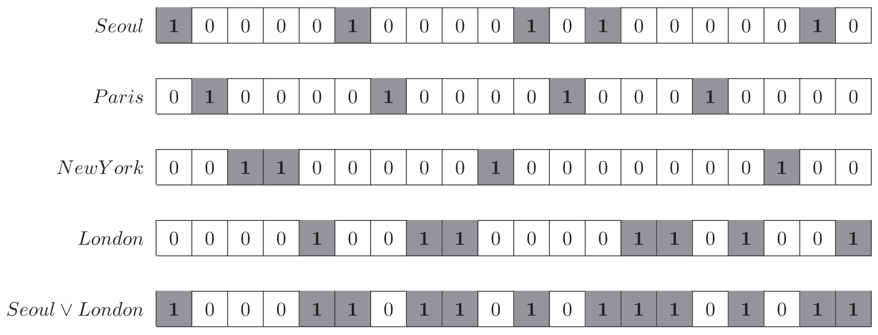

Seoul 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 0
Paris 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0
NewYork 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0
London 0 0 0 0 1 0 0 1 1 0 0 0 0 1 1 0 1 0 0 1
Seoul ∨ London 1 0 0 0 1 1 0 1 1 0 1 0 1 1 1 0 1 0 1 1

Fig. 1. A set of bitmaps.

supported by performing bitwise operations on the bit vectors. Since modern computers are optimized to perform bitwise operations, dealing with bitmap indexes is easy and efficient [4,16].

One of the main drawbacks of bitmap indexes is that their space usage is high when compared to the raw data or a B-tree index, especially when the cardinality is high. To overcome this, compressed bitmap indexes have been proposed. The simplest way to compress a bitmap index is to compress each individual bit vector using a standard text compression algorithm, such as LZ77 [27]. This improves the space usage significantly, but performing logical operations on LZ77-compressed bit strings is usually much slower than uncompressed bit strings, because compressed bitmaps have to be fully decompressed.

As a result, compression schemes that apply some form of Run-Length Encoding (RLE) are suggested to be used for compressing bitmaps. RLE is a fundamentally lossless compression scheme that encodes a string by splitting it into sequences of runs, and then encoding the runs efficiently. Since a typical bitmap index used in database applications is assumed to have sparsely distributed set bits (i.e., most bits are set to zero), forming long sequences of zeros, RLE-based schemes are expected to achieve good compression and are commonly used to compress bitmaps.

Two of the most popular compression schemes, based on run-length encoding, are Byte-aligned Bitmap Code (BBC) [1,2] and Word-Aligned Hybrid (WAH) bitmap compression [5]. Both the schemes divide the original bit vectors to be compressed into blocks of a specific unit size. The main difference between these two schemes is that BBC uses a unit size of 8 bits, while WAH uses 32 bits as unit size. In terms of performance, BBC typically consumes less space, while WAH supports faster query processing.

In this paper, we propose an improved version of BBC, called Super Byte-aligned Hybrid (SBH) bitmap compression. It uses a new feature that enhances time performance in processing logical operations so that its byte-aligned encoding can lead to better compressibility in comparison with a word-aligned scheme. Specifically, the superiority of our scheme in compressibility becomes more pronounced when the cardinality of an indexed column grows larger than 50. The query processing time of SBH was about five times faster than that of WAH, while the size of compressed bitmap indexes was retained nearly close to that of BBC. Thus, our scheme improves upon both these schemes.

The rest of the paper is organized as follows. We first describe other bitmap compression schemes that were proposed in literature and how they were implemented in Section 2. In Section 3, we describe some algorithmic and implementation details of our new scheme. Section 4 discusses experimental results and comparison with the existing schemes. Finally, Section 5 concludes the paper.

2. Related work

Several compression schemes based on run-length encoding have been proposed in the literature. The main merit of these schemes is that logical operations could be done without decompressing the whole bitmaps. One of the earliest such schemes that have been successful is the Byte-aligned Bitmap Code [2], or BBC for short. BBC is very effective in compressing bit sequences, and in particular for representing bitmap indexes. To improve runtime performance of BBC, a word-based bitmap compression scheme, called the Word-Aligned Hybrid (WAH) scheme has been introduced [24], which takes advantage of the word-level bitwise operations. Subsequently, many other compressed bitmap indexing schemes have been proposed, such as EWAH [15], PLWAH [9], PWAH [22], COMPAX [10], VAL-WAH [12], RLH [20], UCB [3], PLWAH+ [6], and SECOMPAX [23]. Some of the schemes are discussed in Chen et al. [7]. Major schemes have been mathematically analyzed by Guzun and Canahuate [11] and Wu et al. [26]. Most of these schemes are variants of WAH, and perform better than WAH in terms of either time or space or sometimes both. They also typically achieve better time performance than BBC, but not in terms of space.

2.1. Byte-aligned bitmap code (BBC)

Byte-aligned Bitmap Code (BBC) [2] is a byte-based compression scheme that encodes (compresses) a sequence of bytes in the uncompressed bit vector by a sequence of one or more bytes. More specifically, a *run* of bits (i.e., sequence of bits having the same value) followed by a small number of bytes are “compressed” into a *header byte*, optionally followed by one or more *counter bytes* followed by zero or more *literal bytes*, as explained below.

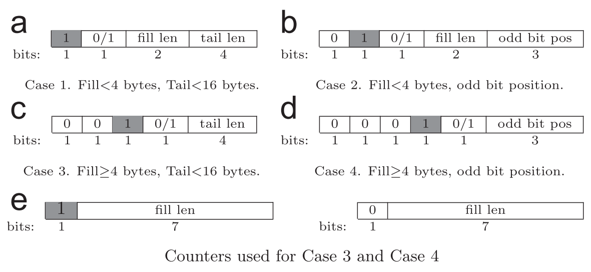

a
1 | 0/1 | fill len | tail len
bits: 1 1 2 4
Case 1. Fill<4 bytes, Tail<16 bytes.
b
0 | 1 | 0/1 | fill len | odd bit pos
bits: 1 1 1 2 3
Case 2. Fill<4 bytes, odd bit position.
c
0 | 0 | 1 | 0/1 | tail len
bits: 1 1 1 1 4
Case 3. Fill≥4 bytes, Tail<16 bytes.
d
0 | 0 | 0 | 1 | 0/1 | odd bit pos
bits: 1 1 1 1 1 3
Case 4. Fill≥4 bytes, odd bit position.
e
1 | fill len
bits: 1 7
0 | fill len
bits: 1 7
Counters used for Case 3 and Case 4

Fig. 2. Four cases of BBC header.

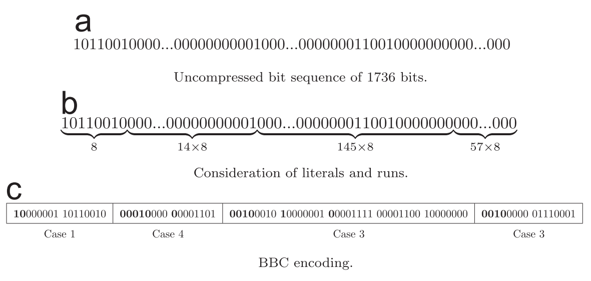


Fig. 3. A bit vector with BBC compression process.

Two variants of BBC scheme have been proposed: a one-sided version that only compresses runs of zeros, and a two-sided version that compresses runs of both zeros and ones [25]. In this paper, we only describe the two-sided version since it can compress almost as good as the one-sided version [13].

In the two-sided version of the BBC scheme, there are four types of distinguishable runs. These four types are distinguishable by their header bytes, which are shown in Fig. 2.

Case 1: In this case, a run of length at most 3 bytes followed by a tail of length up to 15 bytes is encoded by a *header byte* followed by a sequence of *tail bytes*. The *Most Significant Bit* (MSB) of the header byte is set to 1 to indicate this case. The bit following the MSB stores a *fill-bit*, which is either 0 or 1 depending on whether the run it encodes is a sequence of zeros or ones. This is followed by a two-bit representation storing the length of the run (in bytes), which in turn is followed by a four-bit representation indicating the length of the tail (again, in bytes). If k is a value stored in the last four bits of this header byte (where $0 \le k \le 15$), the k bytes following the header byte store the next k bytes following the run that is encoded by the header byte.

Case 2: In this case, a run of at most 3 bytes, followed by a single *odd byte* is encoded by a single header byte. The odd byte has a property that all bits, except one, in the byte are set to the same bit as the run preceding it. The first two most-significant bits in the header byte are set to 01 to indicate this case. The next bit stores the fill bit, and the two bits following it store the length of fill. The remaining three bits store the *odd position*, i.e., position that has a different bit in the odd byte.

Case 3: This case is similar to Case 1, except that the length of fill is at least 4. Instead of putting the fill length in the header byte, it uses one or more *counter bytes* to encode the fill length. The MSB of a counter byte is set to 1 if there are more counter bytes, and is set to 0 otherwise. The remaining seven bits in the counter byte store the bit sequence corresponding to the fill length (spread over all the counter bytes).

Case 4: This case is similar to Case 2, except that the length of the fill is at least 4. Again, counter bytes are used to store the fill length (as in Case 3).

An example bit sequence and its result of compression is shown in Fig. 3.

The BBC scheme compresses nearly as well as *gzip* [13]. However, in the worst case, the total time required to perform a logical operation on two BBC compressed bitmaps can still be longer than that on two uncompressed bitmaps.

2.2. Word-aligned hybrid (WAH)

Another compression scheme based on run-length encoding is *Word-Aligned Hybrid* (WAH) scheme [24]. This scheme is similar to BBC, except that unit of compression is a *word* (of length 32 bits) instead of a byte – hence its name. The scheme is much simpler compared to the two-sided BBC described earlier. The input bitmap is divided into blocks of length 31 bits each. Each word in the compressed bitmap encodes one or more of these blocks. If a block contains both zeros and ones, then it is encoded as a *literal block* by setting the MSB to 0, and writing the 31 bits of the block as remaining bits in the word. For $k \le 2^{30} - 1$, if there is a run of k blocks of zeros (or ones), then this run is encoded by setting the MSB to 1, the bit following the MSB to 0 (or 1), and the remaining bits to the value of k in binary. Runs that are at least 2^{30} blocks are encoded by splitting them into shorter runs that can be encoded in one word in a greedy manner.

An example bit sequence and its result of compression is shown in Fig. 4.

2.3. Enhanced word-aligned hybrid (EWAH)

Enhanced Word-Aligned Hybrid (EWAH) [15] is a variant of WAH. It improves the WAH scheme but the most important difference from WAH is that EWAH never generates a compressed bitmap larger than (within 0.1% of) the uncompressed bitmap. Fig. 5 shows an example of EWAH encoding.

The EWAH compression algorithm is as follows. The given bitmap is divided into 32-bit groups. These groups are classified as two types: *clean* word and *dirty* word. A clean word contains all equal bits, either all zeros or all ones. On the other hand, a dirty word cannot be compressed since it consists of heterogeneous bits (0 and 1).

A marker word is a kind of header indicating a type of next words. The marker word consists of three parts. The first part (one bit) is type of clean words (zero or one), half of a word (16 bits) store the number of clean words and the rest of bits (15 bits) store the number of dirty words. Thus, the maximum number of clean words and dirty words that can be compressed using a single marker word are $2^{16} - 1$ and $2^{15} - 1$, respectively.

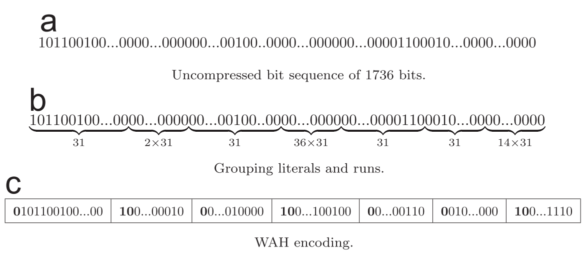
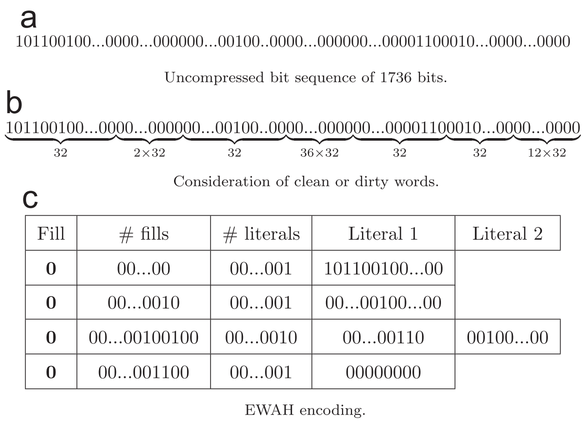


Fig. 4. A bit vector with its WAH compressed form.



Fill	# fills	# literals	Literal 1	Literal 2
0	00...00	00...001	101100100...00	
0	00...0010	00...001	00...00100...00	
0	00...00100100	00...0010	00...00110	00100...00
0	00...001100	00...001	00000000	

Fig. 5. A bit vector with its EWAH compressed form.

2.4. *Position list WAH (PLWAH)*

Prior WAH-based schemes consider a word with only a single odd bit as a literal, which makes the word incompressible. *Position List WAH* (*PLWAH*) [9] tries to alleviate this, by storing location of an odd bit explicitly in a fill word using $s\log_2 w$ bits, where w is the size of a single word and s is the number of odd bits.

PLWAH compresses a bitmap by four steps: (i) divide the original bitmap into $w-1$ bit groups; (ii) identify groups that are either 0-fills or 1-fills which can be merged; (iii) for non-mergeable groups, append a 0 to MSB to indicate that they are literals. Otherwise, put the length of a merge to $w-2-s\log_2 w$ *Least Significant Bit* (*LSB*)s, put 1 to MSB and put either 0 (for 0-fill) or 1 (for 1-fill) to the bit next to the MSB to represent the type of a merge; (iv) identify a nearly identical literal following the merge, calculate the location of the odd bit and put it in a fill word.

Fig. 6 explains these steps of PLWAH compression with an example. Firstly, a bitmap is grouped into 31-bit sequence. After this, depending on the content, a non-set bit is appended to the MSB if a group is a literal, or a set bit otherwise. Finally, if a literal word next to the fill word contains only a single odd bit, then the location of it is put in the 3rd to 7th significant bits. For instance, the second literal word contains a set bit only in the 27th location, thus 11011 is piggybacked in the first fill word, as underlined in the figure. If more than one odd bit exists, then the locations are concatenated after the first one.

Another compression scheme called *COmpressed N Composable Integer SEt* (*CONCISE*) [8] is suggested for $s=1$.

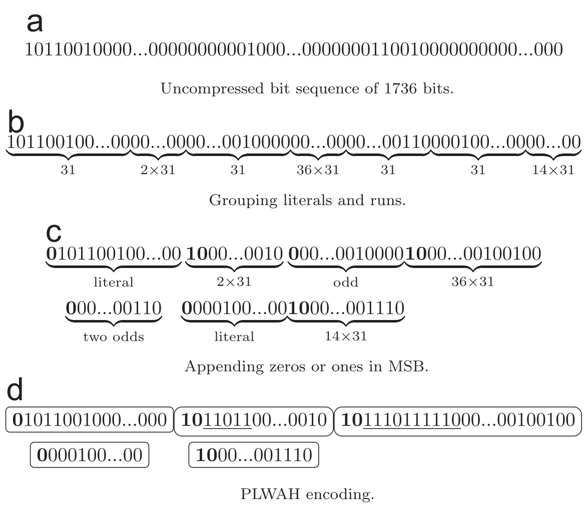


Fig. 6. A bit vector and PLWAH compression process.

This scheme has higher space usage than that of PLWAH when there is more than one odd bit per literal, but it takes longer to compress than PLWAH.

2.5. *Partitioned WAH (PWAH)*

Partitioned WAH (*PWAH*) [22] divides a single word into P partitions, while a header of P bits exists to specify the type of a certain partition, whether it is a fill or a literal. Compared to WAH and EWAH, PWAH has the efficiency of storing bitmaps that have a sequence of literal and 0-fill.

Depending on the word size, PWAH has fixed value of P. In case of $w=32$, there are two different values of P: 2, 4, whose partition occupies $\frac{32-P}{P}$ bits.

PWAH uses extended fill to save the space needed to represent contiguous zeros that exceeds the maximum value of a run to indicate. Rather than adding up the number of fill length consecutively, extended fills concatenate numbers in a bitwise manner.

Fig. 7 explains the process of PWAH compression. In each header segment, the information about subsequent segments is stored. If the segment is a literal, it is stored in uncompressed form. If the segment is a fill, then the type of fill and its length are stored. In case of the length exceeding limit of container, it is stored in multiple segments. For example, the length 66 in PWAH-4 is stored in two segments and concatenating them gives the total length.

2.6. *COMPressed adaptive indeX (COMPAX)*

Different from three WAH-based compression schemes, *COMPressed Adaptive indeX* (*COMPAX*) [10] only considers 0-fills compressible. Nevertheless, COMPAX merges a sequential combination of 0-fills and literals to a single chunk (run), which is known as piggybacking approach. This applies when only a single byte in a word is not a 0-fill.

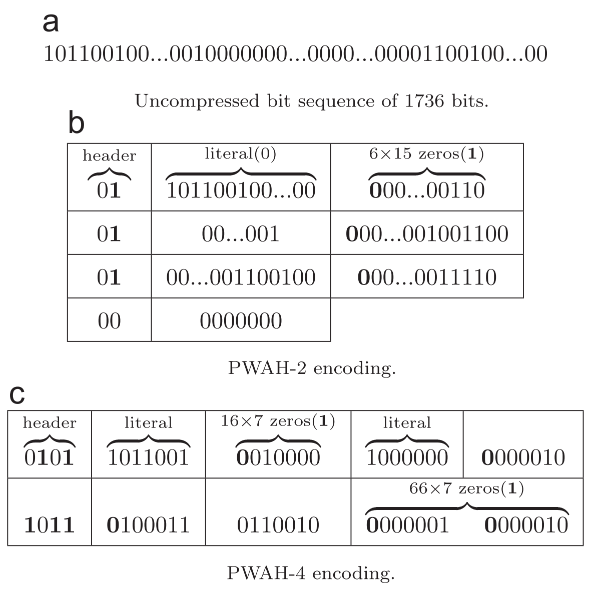


Fig. 7. A bit vector and PWAH compression process.

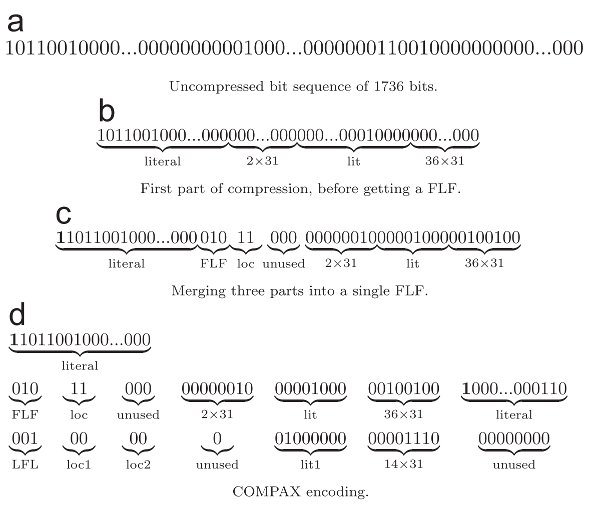


Fig. 8. A bit vector with COMPAX compression process.

There are two more types of a single chunk other than a literal and a 0-fill. One is LFL, which stands for Literal-Fill-Literal, and another is FLF representing Fill-Literal-Fill. For each literal inside this merged form, only a single byte in a single word differs from 0-fills. Compression of bitmap is done on-the-fly with a look-back of at most two words for merging components of FLF and LFL.

Fig. 8 describes the process of compressing a bitmap using COMPAX scheme. While scanning through the bitmap, when a combination of either fill-literal-fill (FLF) or literal-fill-literal (LFL) appears, the compressor backtracks two words and checks if these words could be merged into a chunk. The remaining part is compressed analogously.

2.7. Variable-aligned length WAH (VAL-WAH)

The *Variable-Aligned Length WAH* (*VAL-WAH*) [12] scheme proposes a framework that optimizes both compression and query time by allowing bitmaps to be compressed using variable-length encoding while maintaining alignment. The size of a word w is a power of 2 and it is fixed by a certain machine-dependent factor. However, setting an unit of compression different from $w-1$ is possible. Based on this, dividing a word with segments and putting compressed results there is suggested.

Assuming a word with size w bits, this scheme stores compressed results in blocks of size s, which is known as the segment factor. When $s=w-1$, this algorithm can be considered the same as WAH. Otherwise, more than two blocks appear in a single word. Given w and an alignment factor b, possible values of s are determined by:

$$s=2^i\times(b-1) \text{ where } 0\le i\le \log_2 w-\log_2 b$$

An input bitmap is first inspected by segment length selector. Upon the possible values of s and another user-tuning factor λ, the selector calculates the compressed size using a specific value of s and returns the one with the lowest probable size. After this process is finished, actual bitmap compression is done. A header is generated for each word, with the w/b most significant bits indicating the content of each segment.

An example is given in Fig. 9. When $w=32$ and $b=8$, we need 4 bits to store a flag and remaining 28 bits to store blocks. Candidates of s include 7, 14 and 28. We omit the case where $s=7$. For each candidate, the compressed size is calculated by the segment length selector. In this case, $s=15$ has the lowest compression size, so it is chosen as the segment length. Finally, WAH compression is applied.

2.8. Run-length Huffman (RLH)

Run-Length Huffman (*RLH*) [20] is a combination of run-length encoding and Huffman encoding. Firstly, this scheme calculates distances between 1s in the bitmap and gets frequency of the distance. For instance, a bit sequence 0000001 can be represented as 6 and the frequency of 6 is one. After this process finishes, the result is compressed by Huffman encoding.

Fig. 10 explains the process of compressing a bitmap using RLH. In the first step, distances between successive 1s are calculated. Frequencies of the gathered values are computed and are used to build the Huffman tree. The codes generated from this Huffman tree are used to compute the encoding.

3. Super byte-aligned hybrid (SBH)

We now describe our new scheme for bitmap compression, which we refer to as *Super Byte-aligned Hybrid* (SBH). It is a byte-based compression scheme similar to BBC. We use the concept of what we call a *super-bucket* to speed up logical operations, at the expense of a slight increase in the space when compared with BBC.

More specifically, unlike BBC and WAH, SBH first divides the bit sequence into blocks, called super-buckets, of length l_b, for some integer parameter l_b. It then applies a a

$10110010000...00000000001000...000000011001000000000...000$

Uncompressed bit sequence of 1736 bits.

b

$\underbrace{0\mathbf{1}00}_{\text{header}}\,\underbrace{1011001000...000}_{\text{literal}}\;\underbrace{\mathbf{0}000...000111}_{7\times 14}$

$\underbrace{0\mathbf{1}00}_{\text{header}}\;\underbrace{00...00100...00}_{\text{literal}}\;\underbrace{\mathbf{0}00...001010001}_{81\times 14}$

$\underbrace{0\mathbf{1}00}_{\text{header}}\;\underbrace{00...00110010}_{\text{literal}}\;\underbrace{\mathbf{0}00...00110101}_{53\times 14}$

Case where $s = 14$ (96 bytes).

c

$\underbrace{0000}_{\text{header}}\;\underbrace{101100100...00}_{\text{literal}}$	$\underbrace{\mathbf{1}000}_{\text{header}}\;\underbrace{\mathbf{0}00...0010}_{2\times 28}$
$\underbrace{0000}_{\text{header}}\;\underbrace{00...00100...00}_{\text{literal}}$	$\underbrace{\mathbf{1}000}_{\text{header}}\;\underbrace{\mathbf{0}00...00101000}_{40\times 28}$
$\underbrace{0000}_{\text{header}}\;\underbrace{00...001100100...00}_{\text{literal}}$	$\underbrace{\mathbf{1}000}_{\text{header}}\;\underbrace{\mathbf{0}00...0001000}_{16\times 28}$

Case where $s = 28$ (192 bytes).

Fig. 9. A bit vector with VAL-WAH compression process.

a $101100100...0010000000...0000...00001100100...00$

Uncompressed bit sequence of 1736 bits.

b

$\underbrace{1}_{0}\underbrace{01}_{1}\underbrace{1}_{0}\underbrace{001}_{2}\underbrace{0000...00000000001}_{112}\underbrace{000...00000001}_{1147}\underbrace{1}_{0}\underbrace{001}_{2}\underbrace{0000000000...000}_{463}$

Calculating distance between ones.

c

Distance	Frequency	Codeword
0	3	0
1	1	1010
2	2	100
112	1	1011
463	1	111
1147	1	110

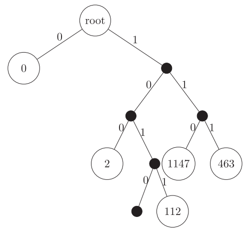
Getting frequency and retrieving codewords.

d

$\underbrace{0}_{0}\underbrace{1010}_{1}\underbrace{0}_{0}\underbrace{100}_{2}\underbrace{1011}_{112}\underbrace{110}_{1147}\underbrace{0}_{0}\underbrace{100}_{2}\underbrace{111}_{463}$

RLH encoding.

e



Huffman tree.

Fig. 10. RLH compression process.

compression scheme similar to, but simpler than, BBC to obtain the compressed bitmap. In the following, we describe compression and decompression algorithms for the SBH scheme in more detail.

3.1. Compression

We first divide the input bitmap into super-buckets of length l_b, and further divide each super-bucket into buckets of length 7 bits each. When all the seven bits in a bucket are zeros (or ones), we call that a 0-fill (or 1-fill) bucket. Otherwise, we call it a *literal* bucket. We choose l_b to be a multiple of 7, so that each super-bucket holds an integer number of buckets. For practical reasons, we choose l_b to be at most $7 \times (2^{12}-1)$, so that any runs within a super-bucket can be encoded using at most two bytes.

A literal bucket is stored as-is in a byte, with an added 0 in the MSB position. Any other bytes in the compressed bitmap that do not encode a literal bucket have their MSBs set to 1 to distinguish them from literal buckets. In particular, if we have a run of zeros (or ones) that spans k buckets in the uncompressed bitmap, for $1 \le k \le 2^{12}-1$, it is encoded by writing the value of k in binary using either one or two bytes (as explained later).

If the value k is at most 63 ($=2^6-1$), we simply write down binary representation of k using six bits, and add **10** for 0-fill (or **11** for 1-fill) in the most significant positions. On the other hand, if k is larger than 63, then we write binary representation of k using 12 bits. The procedure is: (i) split k into two equal halves; (ii) add **10** (or **11**) at the beginning of both halves; (iii) write the resulting bytes in reverse order (i.e., second byte followed by the first byte).

For example, suppose there is a run of zeros that spans 91 buckets (i.e., 91×7 consecutive 0 s). We encode this run by writing the two bytes **10**011011 and **10**000001, which is obtained by writing the binary representation of 91 with leading zeros to make its length equal to 12 (namely, 000001 011011), splitting it into two equal halves, adding **10** to both (to indicate that they represent a run of zeros), and finally, reversing the two resulting bytes. The reversal is done so that we can decode the compressed sequence without any look-ahead. In the example, the first byte is decoded as a run of 27×7 zeros, and the next byte (because it follows a byte with the same header part) is decoded as a run of 64×7 zeros, which are then added to decode a run of 91×7 zeros.

A detailed example of a bit sequence and its corresponding SBH-compressed bitmap are shown in Fig. 11, and the algorithm describing the compression for our scheme is described in Algorithm 1. Algorithms dealing with 0-fills and 1-fills are explained in Algorithms 2 and 3, respectively. For Fig. 11, the length of super-bucket is arbitrarily set to $7 \times (2^7-1)$.

Algorithm 1. Compression algorithm of SBH.

```
Inpupt: Uncompressed bitmaps, size of super-bucket ($l_b$)
Output: Compressed bitmaps.
  for $x=1$ to cardinality do
    for $y=1$ to length of bitmaps/$l_b$ do
      for $z=1$ to number of groups within super-bucket do
        // consider a group is whether a literal or a fill (zero or one);
        if group=zero-fill then
          MakeZeroFill (the number of zero-fill);
        else if group=one-fill then
```

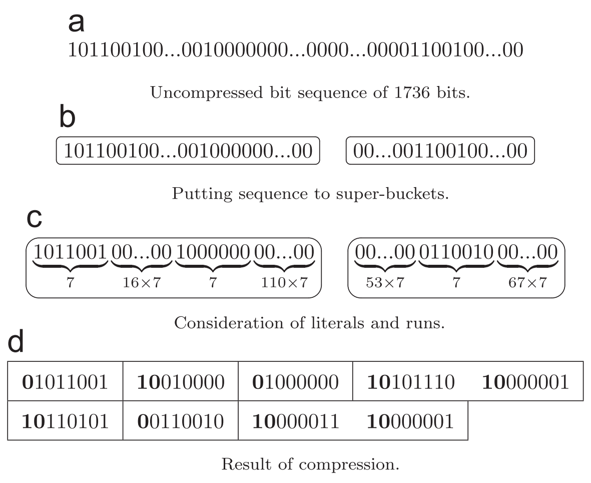


Fig. 11. A bit vector with SBH compression process.

```
        MakeOneFill (the number of one-fill);
    else if group = literal then
        append one bit = 0 and literal to compressed bitmaps;
Process the next bitmap.
```

Algorithm 2. MakeZeroFill.

```
Input: The number of zero-fills ($count_0$).
Output: The compressed zero-fills format.
    while $count_0 > 0$ do
        append two bits 10 and first 6 bits of $count_0$ to compressed
        bitmaps;
        $count_0 = count_0 \gg 6$;
    $count_0 = 0$;
```

Algorithm 3. MakeOneFill.

```
Input: The number of one-fills ($count_1$).
Output: The compressed one-fills format
    while $count_1 > 0$ do
        append two bits 11 and first 6 bits of $count_1$ to compressed
        bitmaps;
        $count_1 = count_1 \gg 6$;
    $count_1 = 0$;
```

3.2. Decompression

The basic flow of decompression is the opposite of the compression process. We read bytes from a compressed bitmap (from left to right) and decode them one by one. In addition, this algorithm uses super-bucket that reduces the query processing time. Initially, bits inside super-bucket are set to all zeros.

The MSB of each byte in the bitmap tells us whether it encodes a literal or a fill. If the MSB is zero, we perform an OR operation between the byte and the current position inside super-bucket. On the other hand, if the MSB is one, then that byte stores a fill. The bit next to the MSB represents the fill bit, either 0 or 1. The last six bits represent the counter value from which we get the length of the run. When decoding a fill, we remember the first two bits of that and check whether they match the first two bits of the following byte. If they match, then we multiply the value in second byte by 64 (2^6) and add it to the counter represented by the previous byte.

If the byte represents a 0-fill, then we do not need to explicitly decompress that byte. That is, because bits inside this byte are all zeros, the original bitmap should only contain zeros. Since we are dealing with OR operations, the result of operation is irrelevant to the content of this bitmap. Thus, we can skip the next x number of bits, where x is the value of counter. For example, assume that the decompression position in the bucket is 8 and the number of the counter on the 0-fill is 6. The decompression position on the bucket is set to $8+42$ (6×7). Between the bit position 9 and 49, the algorithm does not perform any operations because bits inside the relevant position in super-bucket are not affected by the result of decompression.

The decompression algorithm is described in Algorithm 4.

Algorithm 4. Decompression algorithm of SBH.

```
Input: Compressed bitmap indexes, super-bucket size.
Output Row IDs.
  for c = starting position of a query range to ending position of a
    query range do
    $x = 1$;
    $y = 1$;
    while $y <$ the number of bits of super-bucket do
      Initialize all of bits in super-bucket to zero;
      $multi\_num = 6$;
      if $bitmap_c[x]$ is a fill then
        // $bitmap_c[x]$: c-th bitmap
      a = value of $bitmap_c[x]$ with two MSBs removed;
      if Previous byte was a fill then
        $a = a \lll multi\_num$;
        $multi\_num = multi\_num + 6$;
      if $bitmap_c[x]$ is a 1-fill then
        set all bits in super-bucket $[y \cdots y+a\times 6]$ to 1;
      $y = y + a \times 6$;
    else
      super-bucket $[y \cdots y+7]$ = super-bucket $[y \cdots y+7] | bitmap_c[x]$;
      $y = y + 7$;
    if x is the end of bitmap bytes then
      break;
    else
      $x{+}{+}$;
  Check the super-bucket and get the corresponding Row IDs by
  looking up the position of ones.
```

4. Experimental results

In this section, we compare practical performance of our scheme with other bitmap compression schemes. As mentioned earlier, we used two of the well-known bitmap compression schemes, namely BBC and WAH schemes to compare with SBH. In addition, we also used the standard Lempel–Ziv (LZ77) compression scheme [27], which is supported by Linux. The code of SBH is uploaded to an online repository.[1]

4.1. Experimental settings

Experiments were conducted on a PC with CPU (Intel Xeon E7450) 2.40 GHz, 64 GB RAM, and 2 TB hard disk. The

[1] https://github.com/ChoriKim/SBH

machine ran Fedora 22 64-bit with 4.1.6 kernel version. All tested compression techniques were implemented in C++. Two main factors we wanted to improve are the size of the compressed bitmap index and the time required to perform the Boolean operations.

The size of the compressed bitmap indexes was measured using the Linux command "`wc -c`" to compute the overall size of bitmaps. Query time is measured for range queries which execute bitwise-OR operations. Even though other bitwise operations are also commonly used, we assume that time for these other Boolean operations is similar to that of OR operation. `std::chrono` in C++ was used to measure the running time. The result is given as the average of 10 executions per each query.

In our experiments, an OR operation can be described as an SQL query in the following example:

```
SELECT*
FROM LINEITEM
WHERE QUANTITY > 5 AND QUANTITY < 14
```

This SQL query involves searching bitmap indexes for B_6: QUANTITY=6, B_7: QUANTITY=7, … and B_{13}: QUANTITY=13, and performing an bitwise OR operation $B_6 \vee B_7 \vee \ldots \vee B_{13}$ on those.

In addition, as in [19], we used one of the commercial database management systems (DBMS), and PostgreSQL [17], an open-sourced DBMS, to compare the performance. In the analysis we did not compare the compression time of the two DBMSs with the other schemes. The core reason is that DBMSs use not only the compression technique itself, but also other proprietary techniques which we do not know about the internals in detail. PostgreSQL, by default, provides a combination of indexing schemes; we used the options for generating bitmap indices only.

4.2. Data sets used in experiments

We used two types of data sets: (i) synthetic data sets and (ii) TPC(H) benchmark [21] data sets.

4.2.1. Synthetic data

We used two types of synthetic data sets in experiments: uniform and non-uniform types. The uniform data sets were generated using `srand` and `rand` functions which are in `stdlib.h` header provided by the GNU C Library. To observe the tendency, we varied the number of rows in data from 0.05 billion to 100 billion. The non-uniform data sets were generated with 1 billion rows, and their cardinality varies from 10 to 3000. Two random distributions were used: Gaussian distribution and Zipfian distribution. Gaussian distribution is often used in the natural and social sciences for real-valued random variables, and Zipfian distribution reflects real world, especially in data types used in physical and social sciences. We set the mean of Gaussian distribution as half of the size of the data, and the standard deviation of it as 0.2 × the size of the data. The Zipfian distribution could be represented as:

$$f(k;s,N) = (1/k^s) \Big/ \sum_{n=1}^{N} (1/n^s).$$

where N is the number of elements, k is their rank, and s is the value of the exponent characterizing the distribution. We generated data sets for $s=1$.

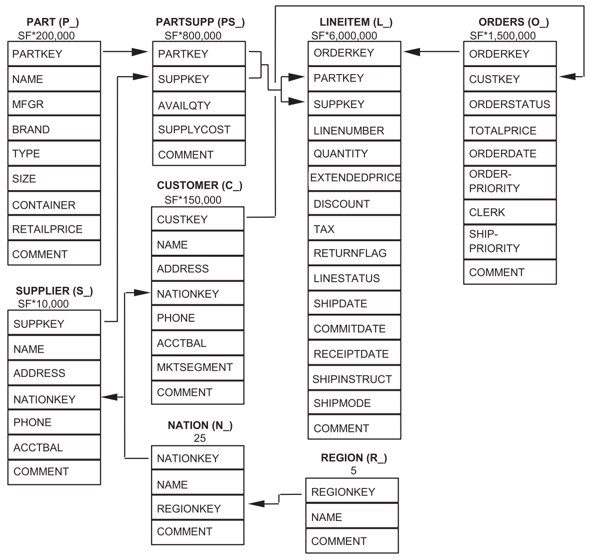


Fig. 12. Table schema of TPC(H) benchmark.

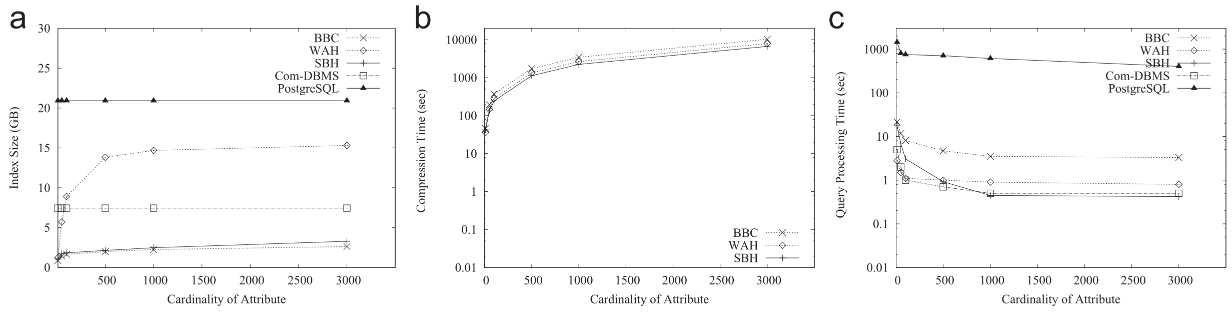
4.2.2. TPC(H) benchmark

The TPC(H) benchmark is commonly used in evaluating database systems, and is considered as a standard decision support benchmark. It consists of a suite of business oriented ad hoc queries and concurrent data modifications. TPC(H) benchmark is composed of eight separate and individual tables, as shown in Fig. 12. The benchmark data are generated along with a scale factor, *SF*. We experimented with six different scale factors: 1, 10, 30, 50, 100, and 300.

We used the table LINEITEM, which consists of 16 attributes mentioned below (the values in parentheses indicate cardinalities of the corresponding attribute):

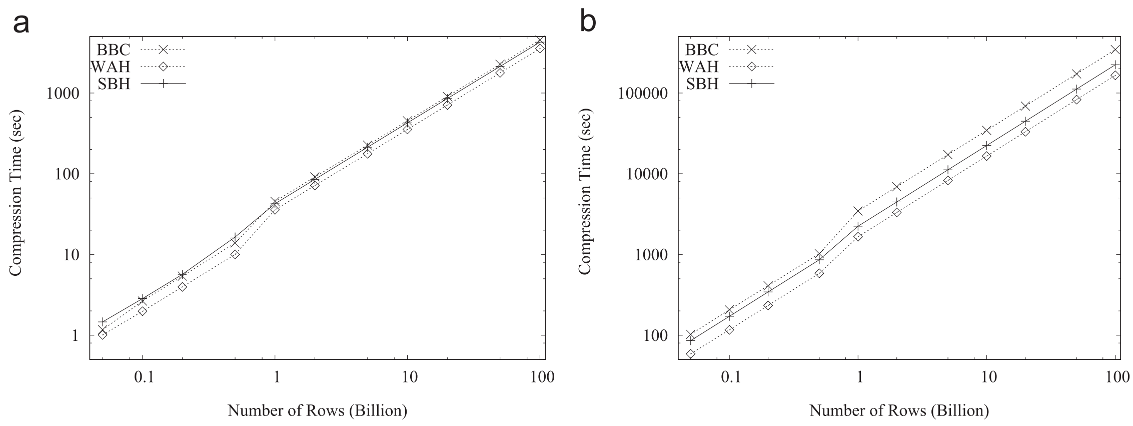
(ORDERKEY ($1,500,000 \times SF$), PARTKEY ($200,000 \times SF$), SUPPKEY ($10,000 \times SF$), LINENUMBER (7), QUANTITY (50), EXTENDEDPRICE (flexible), DISCOUNT (11), TAX (9), RETURNFLAG (3), LINESTATUS (2), SHIPDATE (2526), COMMITDATE (2466), RECEIPTDATE (2,555), SHIPINSTRUCT (4), SHIPMODE (7) and COMMENT (flexible)).

We disregarded attributes having flexible cardinality (ORDERKEY, PARTKEY, SUPPKEY, EXTENDEDPRICE and COMMENT). Among attributes having fixed cardinality, we selected three representative attributes having different cardinalities DISCOUNT, QUANTITY and SHIPDATE. This test set gives different ranges of cardinality, from small to large.



Index size. Compression time. Query processing time.

Fig. 13. Tendency with cardinality of attribute on synthetic data (random, one billion rows).



10 cardinality. 1,000 cardinality.

Fig. 14. Tendency of compression time on synthetic data along with the number of rows.

4.3. *Results*

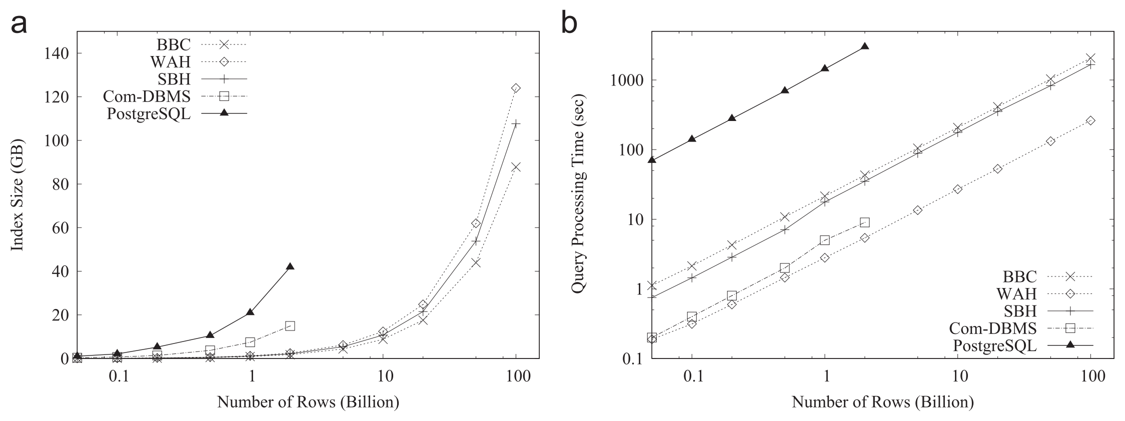
We evaluated the performance of SBH in several ways by comparing it with BBC, WAH, a commercial DBMS and PostgreSQL. First, we compare the schemes in terms of compression time by varying the cardinality of the attribute. Because bitmap indexes of the commercial DBMS and PostgreSQL are not compressed, we do not include them in the comparison for the compression times. Next, the performance of the schemes is analyzed with respect to the cardinality and the number of rows. Finally, we check the relationship between data size and query processing time, where a query consists of bitwise-OR operations. We use both the synthetic data and the TPC (H) benchmark data for scalability tests. In our machine, because both the DBMS and PostgreSQL could not load the data over 5 billion rows for the synthetic data and over scale factor 100 for the TPC(H) benchmark, we only measured their performance for data sizes up to 2 billion rows (in Figs. 15 and 16), and for scale factors up to 100 (in Figs. 20–22).

When the cardinality of bitmap is less than 50, all the techniques have similar sizes for the compressed bitmap indexes and also similar query processing times, because bitmap indexes are not compressed well in this interval. Thus, better performance of compression schemes comes when the cardinality of bitmap is above 50.

4.3.1. *Synthetic data*

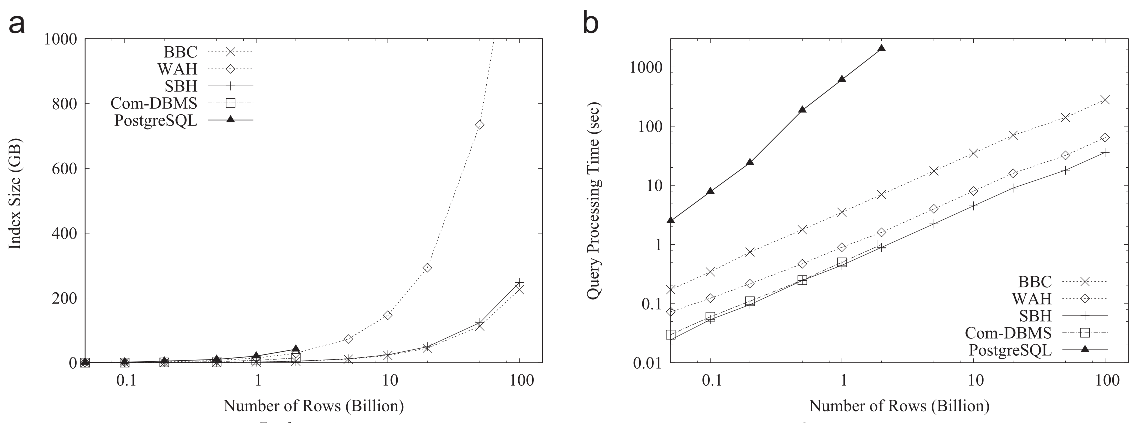
Fig. 13 shows the size of bitmap indexes and compression time in relation to the cardinality of attribute, for one billion rows. Note that except WAH, the compression ratio of all techniques gets better while having similar tendency. In Fig. 13a, we notice that the index size of WAH increases rapidly until around 100 cardinality, but the graph grows smoothly after that. The sizes of both the DBMSs are independent of the cardinality. The reason could be that they allocate a fixed number of bytes per row. In terms of compression time, SBH is the fastest. Note that we do not include two DBMSs in the comparison of the compression times since they do not compress the bitmap index. Fig. 13c shows the query processing time with cardinality of attribute at one billion rows. After 50 cardinality, the query processing time of SBH is the fastest, regardless of cardinality and number of bitwise operations. Compared to the commercial DBMS and PostgreSQL, the query processing time of SBH does not lag behind.

Fig. 13 shows that the query processing time of SBH is the fastest and the size of compressed bitmap indexes using SBH is relatively small. The size of those using BBC is the smallest in terms of compression, but the query



Index size. Query processing time.

Fig. 15. Tendency of query processing time on synthetic data (random), 10 cardinality.



Index size. Query processing time.

Fig. 16. Tendency of query processing time on synthetic data (random), 1000 cardinality.

processing time is about 50 times slower than that of SBH. Compared to WAH, SBH has also better query processing time and size. In Fig. 13c and a, abnormal shape in less cardinality range of WAH graph is found. The reason is that literals reside in most of bitmap indexes with low cardinality, so that the bitmap indexes are seldom compressible.

Figs. 14 shows the tendency of compression time for cardinalities 10 and 1000. We measured the compression time with only these two cardinalities, since bitmap indexes are hardly compressed for small (10) cardinality, while for large (1000) cardinality, bitmap indexes are compressed well. It shows that the compression time of all techniques increases linearly. Intuitively, it is clear that the larger the number of rows gets, the longer time it takes to compress the bitmap indexes.

The tendencies in Figs. 13 and 14 can be explained by noting that algorithms which guarantee a good compression ratio commonly involve complex operations trying to find the best case to get better compression, which increases its running time.

Figs. 15 and 16 show the experimental results executed in 10 and 1000 cardinality, respectively. Fig. 15a shows the tendency of compressed index size on randomly generated synthetic data. Fig. 14a indicates query processing time where each query performs eight OR operations in the bitmap index. In Fig. 15a, the graph is linearly increased with respect to the number of rows. As we mentioned before, in 10 cardinality, compressing bitmap indexes is poorly done. Thus, SBH gives good compressed index size among all the schemes compared, but the query processing time of the scheme is not as good as our expectation.

From Fig. 16a, we note that as the number of rows becomes larger, the size of bitmap indexes increases linearly except PostgreSQL. Fig. 14b shows the cases executed by the range query which is composed of eight OR operations on each data set. Except for two DBMSs, the other three schemes have the similar increasing tendency. From Fig. 16, we come to the conclusion that SBH is efficient for large cardinalities.

The relative performance does not change even though the size of data increases. To summarize, the compression time is proportional to the number of rows, and SBH has

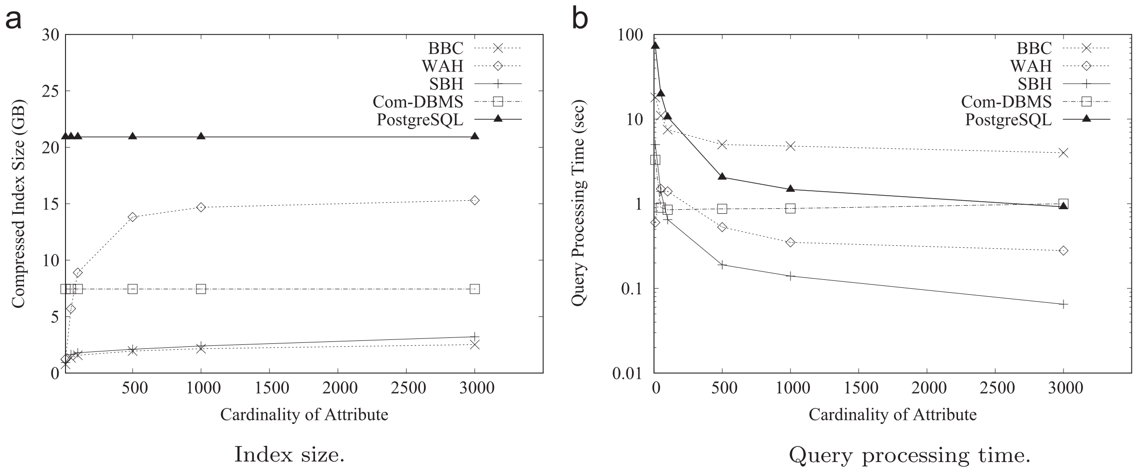


Fig. 17. Synthetic data (Gaussian distribution) with one billion rows.

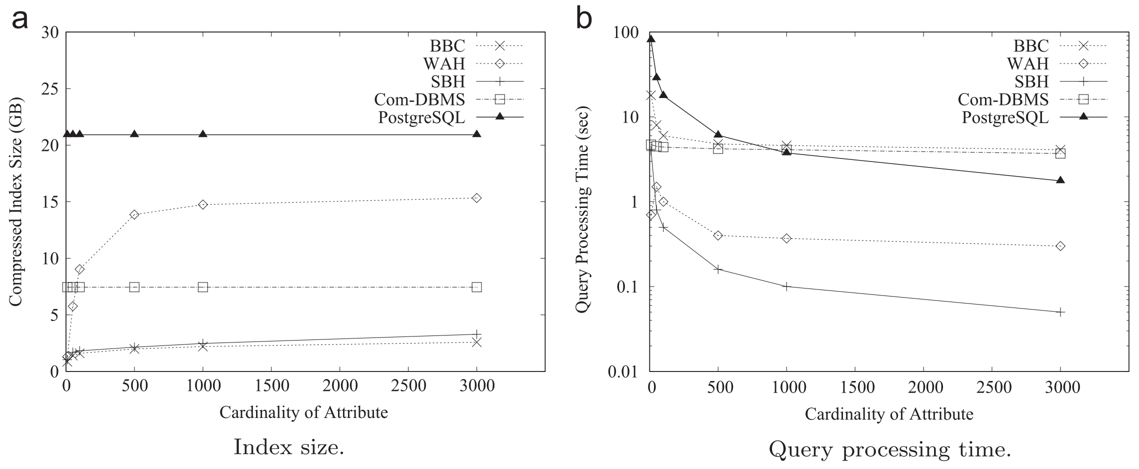


Fig. 18. Synthetic data (Zipfian distribution) with one billion rows.

the best performance regarding the query processing time and compression size.

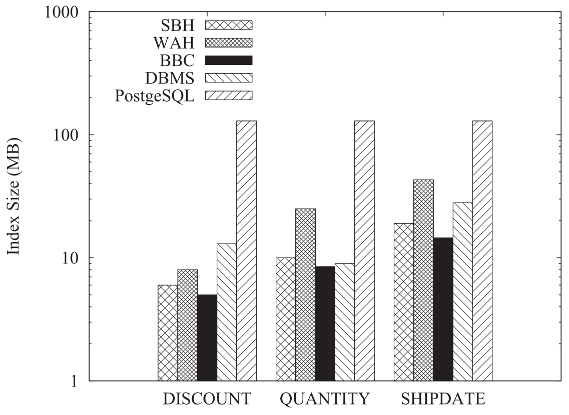
Figs. 17 and 18 show the compression index size and query processing times for data generated using the Gaussian and Zipfian distributions, respectively, which are non-uniform distributions. The query processing times of those schemes are also measured using eight OR operations. These results have similar tendency in terms of size and time as the randomly generated data. In terms of size, SBH and BBC schemes take significantly less space than WAH, since byte-based schemes have better compressibility than word-based schemes, in general. Moreover, in terms of query processing time, SBH is 50 times faster than BBC, and 5 times faster than WAH.

4.3.2. TPC(H) benchmark

Fig. 19 shows the size of compressed bitmap indexes and their query processing time. From the synthetic data experiments, we observed that for cardinalities larger than 50, the performance of SBH improves noticeably. This implies that for the attributes QUANTITY and SHIPDATE (with cardinalities 50 and 2526, respectively), SBH is better on the query processing time as well as the bitmap index size. Since the value of cardinality around 50 is a starting point to make compressed bitmap indexes and effective bitwise operations, the performance of SBH is not so good relatively for the QUANTITY attribute. However, in the case of the SHIPDATE attribute, SBH has outstanding performance as expected.

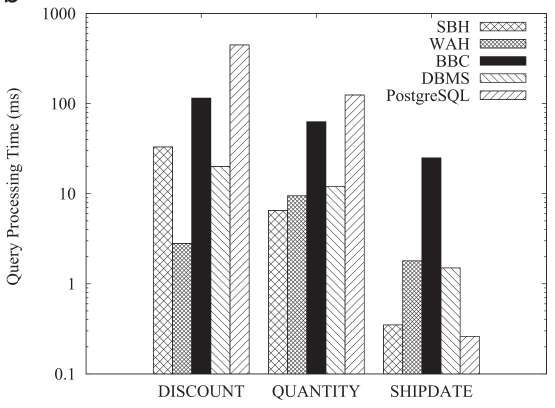
Figs. 20–22 indicate the tendency of each of the schemes with respect to the scale factor (*SF*). Fig. 20 shows the index size and query performance for the DISCOUNT attribute (with cardinality 11). If we exclude two DBMSs for the remaining three schemes, the increasing order of compressed index size is the same as the decreasing order of query processing time. As mentioned before, since this attribute is in the range of low cardinality value, SBH does not show remarkable performance.

Fig. 21 shows the size and query time performance of the four schemes for the QUANTITY attribute. Since the compressibility of bitmap index changed from Fig. 20, the tendency of query processing time is also changed. In case of SBH, the relative factor between compressed index size and query processing time is noticeable. Even though the a



Index size.

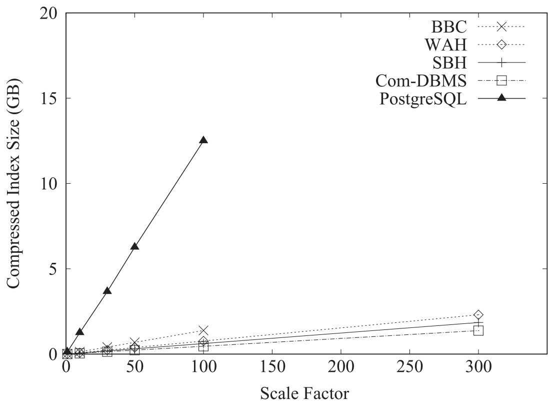
b



Query processing time.

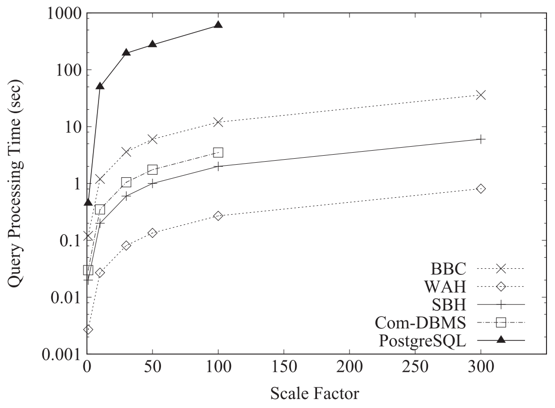
Fig. 19. TPC(H) Benchmark in scale factor 1.

a



Index size.

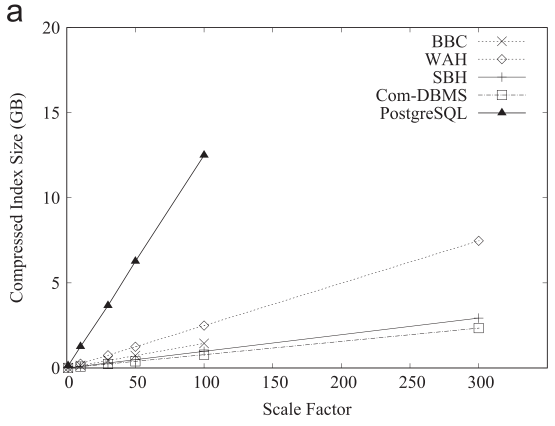
b



Query processing time.

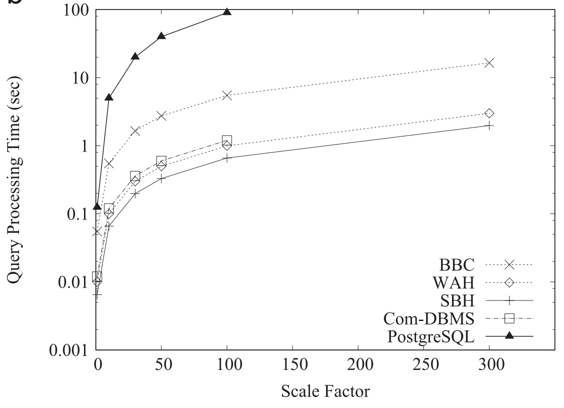
Fig. 20. TPC(H) Benchmark in DISCOUNT attribute.

a



Index size.

b



Query processing time.

Fig. 21. TPC(H) Benchmark in QUANTITY attribute.

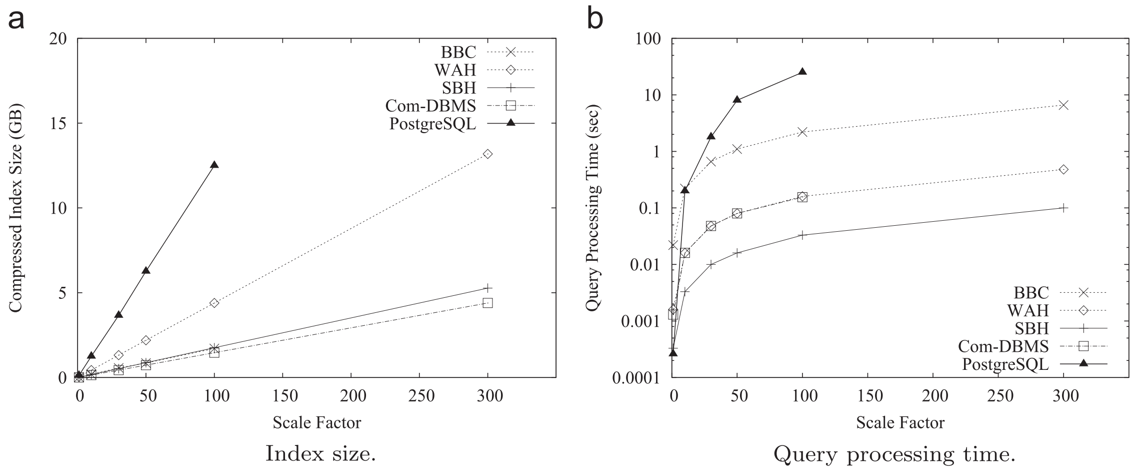


Fig. 22. TPC(H) Benchmark in SHIPDATE attribute.

increment in compressed index size in SBH is higher, its query processing time is still the fastest even in larger scale factors.

Fig. 22 displays results for the SHIPDATE attribute. Most properties are similar to those from Fig. 21. The compressed index size of SBH is the smallest. Additionally, the query processing time of SBH is the fastest.

In all three attributes, the size of compressed bitmap index is proportional to the scale factor. Similarly, the query processing time also increases along with the scale factor. SBH-compressed bitmap indexes occupy relatively small size, and their query processing time is also better than the other techniques, except for the DISCOUNT attribute.

4.3.3. Summary

- In all three techniques, the time to compress bitmap indexes increases linearly with the cardinality of the indexed column.
- The time to compress bitmap indexes is also proportional to the number of rows in all three techniques. For cardinalities less than 50, the compression ratio of all techniques is not high, i.e., all three schemes do not achieve good compression. For this reason, query processing time in all techniques is high.
- SBH scheme gets better performance for cardinalities larger than 50, and its compressed index size is lower than that of the other schemes, and is comparable to that of BBC.
- The results for the TPC(H) benchmark data are similar to those for the synthetic data.

5. Conclusions

We propose a new bitmap compression scheme called SBH which performs better than the widely used WAH compression scheme. The query processing time of our scheme is significantly better than BBC compression scheme while it achieves almost similar space consumption. Even though bitwise operations on compressed bitmaps using a byte-based scheme like BBC are much slower than those of WAH, SBH could take less time to compute logical operations for cardinalities larger than 50. Through rigorous experiments, we illustrated the remarkable performance of SBH compared to WAH, BBC and two DBMSs, a commercial DBMS and PostgreSQL. We also have shown that the differences in performance among all techniques are affected not only by the number of rows but also by the cardinality.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) Grant (No. 2015R1A5A7037372) funded by the Korean Government (MSIP). We thank Prof. Bernhard Egger for allowing us the use of their computational resources.

References

[1] G. Antoshenkov, Byte-aligned Data Compression, US Patent 5,363,098, November 8, 1994.
[2] G. Antoshenkov, Byte-aligned bitmap compression, in: Proceedings of the IEEE Data Compression Conference, DCC 1995, Snowbird, Utah, March 28-30, 1995. IEEE Computer Society, 1995, ISBN 0-8186-7012-6.
[3] G. Canahuate, M. Gibas, H. Ferhatosmanoglu, Update conscious bitmap indices, in: 19th International Conference on Scientific and Statistical Database Management, 2007, SSBDM'07, IEEE, 2007, pp. 15.
[4] S. Chambi, D. Lemire, O. Kaser, R. Godin, Better bitmap performance with roaring bitmaps, Softw.: Pract. Exp. 46 (5) (2016) 709–719.
[5] C.-Y. Chan, Y.E. Ioannidis, Bitmap index design and evaluation, in: ACM SIGMOD Record, vol. 27, ACM, New York, NY, USA, 1998, pp. 355–366.
[6] J. Chang, Z. Chen, W. Zheng, Y. Wen, J. Cao, W.-L. Huang, PLWAH+: a bitmap index compressing scheme based on PLWAH, in: Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '14, ACM, New York, NY, USA, 2014, pp. 257–258.
[7] Z. Chen, Y. Wen, J. Cao, W. Zheng, J. Chang, Y. Wu, G. Ma, M. Hakmaoui, G. Peng, A survey of bitmap index compression algorithms for big data, Tsinghua Sci. Technol. 20 (1) (2015) 100–115.

[8] A. Colantonio, R.D. Pietro, CONCISE: compressed 'n' composable integer set, Inf. Process. Lett. 110 (16) (2010) 644–650.
[9] F. Deliège, T.B. Pedersen, Position list word aligned hybrid: optimizing space and performance for compressed bitmaps, in: Proceedings of the 13th International Conference on Extending Database Technology, EDBT '10, ACM, New York, NY, USA, 2010, pp. 228–239.
[10] F. Fusco, M.P. Stoecklin, M. Vlachos, Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic, Proc. VLDB Endow. 3(1–2) (2010) 1382–1393.
[11] G. Guzun, G. Canahuate, Performance evaluation of word-aligned compression methods for bitmap indices, Knowl. Inf. Syst. (2015) 1–28.
[12] G. Guzun, G. Canahuate, D. Chiu, J. Sawin, A tunable compression framework for bitmap indices, in: 2014 IEEE 30th International Conference on Data Engineering (ICDE), March 2014, pp. 484–495.
[13] T. Johnson, Performance measurements of compressed bitmap indices, in: Proceedings of the 25th International Conference on Very Large Data Bases, 1999, pp. 278–289.
[14] K. Lee, B. Moon, Bitmap indexes for relational XML twig query processing, in: Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM '09, ACM, New York, NY, USA, 2009, pp. 465–474.
[15] D. Lemire, O. Kaser, K. Aouiche, Sorting improves word-aligned bitmap indexes, Data Knowl. Eng. 69 (1) (2010) 3–28.
[16] P.E. O'Neil, G. Graefe, Multi-table joins through bitmapped join indices, SIGMOD Record 24 (3) (1995) 8–11.
[17] PostgreSQL 9.4.5, ⟨http://www.postgresql.org/⟩.
[18] R. Sinha, S. Mitra, M. Winslett, Bitmap indexes for large scientific data sets: a case study, in: 20th International Conference on Parallel and Distributed Processing Symposium (IPDPS), April 2006, 10 pp.
[19] R.R. Sinha, M. Winslett, Multi-resolution bitmap indexes for scientific data, ACM TODS 32 (3) (2007) 16.
[20] M. Stabno, R. Wrembel, RLH: bitmap compression technique based on run-length and Huffman encoding, Inf. Syst. 34 (4–5) (2009) 400–414.
[21] Transaction Processing Performance Council, TPC BENCHMARK^{TM}H Standard Specification, Revision 2.16.0, 2013, ⟨http://www.tpc.org/tpch/default.asp⟩.
[22] S.J. van Schaik, O. de Moor, A memory efficient reachability data structure through bit vector compression, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, ACM, New York, NY, USA, 2011, pp. 913–924.
[23] Y. Wen, Z. Chen, G. Ma, J. Cao, W. Zheng, G. Peng, S. Li, W.-L. Huang, SECOMPAX: a bitmap index compression algorithm, in: 2014 23rd International Conference on Computer Communication and Networks (ICCCN), August 2014, pp. 1–7.
[24] K. Wu, E.J. Otoo, A. Shoshani, Optimizing bitmap indices with efficient compression, ACM Trans. Database Syst. 31 (1) (2006) 1–38.
[25] K. Wu, E.J. Otoo, A. Shoshani, H. Nordberg, Notes on Design and Implementation of Compressed Bit Vectors, Technical Report, LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.
[26] Y. Wu, Z. Chen, Y. Wen, J. Cao, W. Zheng, G. Ma, A general analytical model for spatial and temporal performance of bitmap index compression algorithms in big data, in: 24th International Conference on Computer Communication and Networks, ICCCN 2015, Las Vegas, NV, USA, August 3-6, 2015. IEEE, pp. 1-10.
[27] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory 23 (3) (1977) 337–343.