Information Systems 34 (2009) 400–414

Contents lists available at ScienceDirect

Information Systems

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

RLH: Bitmap compression technique based on run-length and Huffman encoding ☆

Michał Stabno, Robert Wrembel *

Institute of Computing Science, Poznań University of Technology, Piotrowo 2, 60-965 Poznań, Poland

ARTICLE INFO

Keywords:
Bitmap index
Bitmap compression
Huffman encoding
Run-length encoding
WAH
BBC
RLH

ABSTRACT

In this paper we propose a technique of compressing bitmap indexes for application in data warehouses. This technique, called *run-length Huffman* (RLH), is based on run-length encoding and on Huffman encoding. Additionally, we present a variant of RLH, called RLH-*N*. In RLH-*N* a bitmap is divided into *N*-bit words that are compressed by RLH. RLH and RLH-*N* were implemented and experimentally compared to the well-known word aligned hybrid (WAH) bitmap compression technique that has been reported to provide the shortest query execution time. The experiments discussed in this paper show that: (1) RLH-compressed bitmaps are smaller than corresponding WAH-compressed bitmaps, regardless of the cardinality of an indexed attribute, (2) RLH-*N*-compressed bitmaps are smaller than corresponding WAH-compressed bitmaps for certain range of cardinalities of an indexed attribute, (3) RLH and RLH-*N*-compressed bitmaps offer shorter query response times than WAH-compressed bitmaps, for certain range of cardinalities of an indexed attribute, and (4) RLH-*N* assures shorter update time of compressed bitmaps than RLH.

1. Introduction

A data warehouse (DW) is a large database that integrates data coming from multiple storage systems within an enterprise. These data are analyzed by the so-called on-line analytical processing (OLAP) applications, based on complex queries. OLAP queries join fact tables with multiple level tables and filter data by means of query predicates. Typically, analyzed data are stored in fact tables that reference dimensions via foreign keys. As a consequence, there are many fact records having the same value of a given foreign key. Efficient filtering of large volumes of fact records, based on foreign key values, is well supported by the so-called bitmap indexes.

A bitmap index [14,17] is one of the basic data structures applied to query optimization in DWs. Basically, in the simplest form, the bitmap index is composed of the so-called bitmaps, each of which is a vector of bits (cf. Section 2). Each bit is mapped to a row in an indexed table. If the value of a bit equals 1, then the row corresponding to this bit has a certain value of an indexed attribute. Queries whose predicates involve attributes indexed by bitmap indexes can be answered fast by performing bitwise AND, or OR, or NOT operations on bitmaps, that is a big advantage of bitmap indexes.

The size of a bitmap index strongly depends on the cardinality (domain width) of an indexed attribute, i.e., the size of a bitmap index increases when the cardinality of an indexed attribute increases. Thus, for attributes of high cardinalities (wide domains) bitmap indexes become very large. As a consequence, they cannot fit in main

☆ This work was supported from the Polish Ministry of Science and Higher Education Grant no. N N516 365834.

* Corresponding author. Tel.: +48 61 665 21 27; fax: +48 61 877 15 25.
E-mail addresses: Michal.Stabno@allegro.pl (M. Stabno), Robert.Wrembel@cs.put.poznan.pl (R. Wrembel).

0306-4379/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2008.11.002

memory and the efficiency of accessing data with the support of such indexes deteriorates [37].

1.1. Related work

In order to improve the efficiency of accessing data with the support of bitmap indexes defined on attributes of high cardinalities, the two following approaches have been proposed in the research literature, namely: (1) extensions to the structure of the basic bitmap index and (2) bitmap index compression techniques.

In the first approach, two main techniques, generally called binning as well as bit slicing, can be distinguished.

In [36] (called *range-based bitmap indexing*) and in [21,22,27] (called *binning*), values of an indexed attribute are partitioned into ranges. A bitmap is constructed for representing a given range of values, rather than a distinct value. Bits in a single bitmap indicate whether the value of a given attribute of a row is within a specific range. This technique can also be applied when values of an indexed attribute are partitioned into sets.

The technique proposed in [12] can be classified as a more general form of binning. In [12] sets of attribute values are represented together in a bitmap index. Such a technique reduces storage space for attributes of high cardinalities. The selection of attribute values represented in this kind of an index is based on query patterns and their frequencies, as well as on the distribution of attribute values.

Another form of binning was proposed in [5]. This technique, called *property maps*, focuses on managing the total number of bins assigned to all indexed attributes. A property map defines properties on each attribute, such as the set of queries using the attribute, distribution of values for the attribute, or encoded values of the attribute. The properties are represented as vectors of bits. A query processor needs extension in order to use property maps. Property maps support multi-attribute queries, inequality queries or high selectivity queries, and they are much smaller than bitmap indexes.

The second technique is based on the so-called *bit-sliced index* [4,17,37]. It is defined as an ordered list of bitmaps, $B^n, B^{n-1}, \ldots, B^1, B^0$, that are used for representing values of a given attribute A, i.e., B^0 represents the 2^0 bit, B^1 represents the 2^1 bit, etc. Every value of an indexed attribute is represented on the same number of n bits. As a result, the encoded values in a table form n bitmaps. The bitmaps are called bit-slices. Data retrieval and computation are supported either by the bit-sliced index arithmetic [20] or by means of a dedicated retrieval function [37]. Additionally, a mapping data structure is required for mapping the encoded values into their real values [37].

In the second approach to improving the efficiency of bitmap indexes defined on attributes of high cardinalities, different bitmap compression techniques are used. Two main loss-less techniques can be distinguished in the research literature, namely *byte-aligned bitmap compression* (BBC) [2] and *word-aligned hybrid* (WAH) [26,33–35].

BBC and WAH are based on the so-called *run-length* encoding. Basically, in this encoding, continuous vectors of bits with the same bit value (either "0" or "1") are represented as one instance of the value (e.g., "1") and the count of the values.

The basic difference between BBC and WAH is that BBC divides a bitmap into 8-bit words, whereas WAH divides a bitmap into 31-bit words. A more detailed description of WAH is given in Section 2.2. BBC and WAH offer the best compression ratio for bitmaps describing rows that are ordered by the value of an indexed attribute. Otherwise, the compression ratio is worse. For dense evenly distributed data, bits of value "1" are dense but they are separated with bits of value "0." Therefore, it may be difficult to find continuous bit vectors of values "0" or "1" of length $n*8$ bits for BBC and $n*31$ bits for WAH.

Yet another technique, called *approximate encoding* (AE), for compressing bitmap indexes was proposed in [3]. AE offers approximate query results. False misses are guaranteed not to occur, i.e., all rows that satisfy a query predicate are included into the query result. Moreover, the accuracy of false positives (i.e., rows that do not satisfy a query predicate are sometimes included in the query result set) ranges from 90% to 100%. AE is based on Bloom filters. In AE, the set of bitmaps is treated as a boolean matrix. The matrix is represented in a compressed form in the so-called *approximate bitmap* (AB). In order to compress the boolean matrix, it is encoded into AB using multiple hash functions, based on Bloom filters. For each vector of bits in the matrix, hashing string *hs* is constructed as the function of a row number and a column number in the matrix. Next, k independent hash functions are applied over *hs*. The positions pointed by the hash values are set to "1" in AB.

As discussed in [11,18], reordering columns or rows in matrices can improve clustering of relevant cells. Such techniques can also be applied to bitmap indexes that may be seen as matrices. The main idea is to reorder a bitmap matrix in order to gain better clustering of "0" and "1" cells and then compress the matrix. Thanks to the reordering, the compression ratio can be improved. Unfortunately the reordering problem is NP-hard [11] and only ordering heuristics can be applied.

Bitmap indexes were implemented in major commercial database management systems and can either be explicitly defined by users, e.g., Oracle, Sybase IQ, MODEL 204 or can be implicitly used by a system, e.g., MS SQL Server, IBM DB2. Advanced research implementations of bitmap indexes are represented by *FastBit* [24,15,19] and *RIDBit* [15]. *FastBit* implements basic bitmap indexes, binning, and the WAH bitmap compression technique. In *RIDBit*, dense bitmaps are stored in leaves of a B-tree. The bitmaps replace row IDs, originally stored in B-tree leaves. Sparse bitmaps are automatically converted into row IDs.

Another research area related to bitmap index compression is text compression and inverted files compression in text databases. Several compression techniques have been proposed for compressing texts, e.g., [1,8,6] they are either based on Huffman encoding [10] or on Ziv–Lempel encoding [38]. Several techniques have also been proposed for compressing inverted files, e.g., [7,13,23,29,30,39].

In [13,39] the authors propose an inverted index compression technique where the list of occurrences of term t in data file blocks is represented by means of *gaps* (integers) rather than block numbers. Gaps are further encoded by Elias-γ encoding [7]. In Elias-γ encoding a positive integer x is represented by a unary part and a binary part. The unary part specifies the number of bits required to represent x, whereas the binary part codes x on the bits. δ encoding extends Elias-γ encoding. In δ encoding of x, the unary part is replaced with a γ code. Scholer et al. [23] reports performance tests for compressing different elements of inverted lists (offsets, document numbers) with different encoding techniques (Elias-γ and δ encoding [7] as well as Golomb encoding [9]). Vo and Moffat [29] propose a compression technique for inverted lists that is based on unary coding [31]. It is used for compressing document numbers. In [30] the authors compare different approaches to compressing integers, including the Elias-γ and δ encoding, Golomb encoding, and a variable-byte integer encoding.

The compression techniques developed for inverted files, compress integers that may represent deltas (differences) between sequences of values. In the RLH and RLH-*N* compression techniques that we propose, distances between bits having values "1" may correspond to such deltas. However, RLH and RLH-*N* encode these deltas by means of Huffman encoding, rather than γ or δ encoding.

1.2. Paper focus, contribution, and outline

In this paper we present an alternative bitmap compression technique for exact encoding that offers: (1) good query response times and (2) small sizes of compressed bitmaps. The bitmap compression technique that we developed is called *run-length Huffman* (RLH). Similarly as in BBC and WAH, the proposed technique is based on run-length encoding. However, it differs from BBC and WAH with respect to the following. Firstly, RLH counts distances between bits of value "1," rather than lengths of continuous bits of the same value. The distances become symbols that are next encoded by Huffman encoding technique [10]. Secondly, RLH does not divide bitmaps into words that improves a bitmap compression ratio. In order to better support bitmap updates we proposed a variant of the RLH compression technique, called RLH-*N*. In RLH-*N* a bitmap being compressed is divided into words of *N*-bits length, then each *N*-bit word is compressed by RLH.

The RLH and RLH-*N* compression techniques were implemented and experimentally compared to WAH. As a reference we chose WAH, since bitmaps compressed with WAH offer better query response time than bitmaps compressed with BBC [26,32].

This paper extends our previous work [25] with respect to:

- the development of the RLH-*N* compression technique that accepts words of length equal to 256, 512, 1024, and 2048 bits;
- the comparison of RLH, RLH-*N*, and WAH with respect to CPU time and I/O processing time; these characteristics were measured for: (1) rows unordered, partially ordered, and ordered by the value of an indexed attribute, (2) indexed attributes of different cardinalities (up to 20,000 distinct values), and (3) a dataset composed of 100,000,000 rows;
- the comparison of RLH-*N* and RLH with respect to the efficiency of compressed bitmap modifications.

This paper is organized as follows. Section 2 explains basic definitions and concepts used in the paper. Section 3 presents the RLH and RLH-*N* compression techniques. Section 4 discusses the results of the experimental evaluation of RLH, RLH-*N*, and WAH. Finally, Section 5 summarizes and concludes the paper.

2. Basic definitions

2.1. Bitmap index

A bitmap index is based on the so-called bitmaps. A bitmap is a vector of bits. Every value from the domain of an indexed attribute A has associated its own bitmap. The number of bits in each bitmap is equal to the number of rows in table T that stores A. A bitmap created for value v of indexed attribute A describes these rows in T whose value of A is v. In this bitmap, bit number n is set to "1" if the value of A of the n-th row equals v. Otherwise the bit is set to 0. The concept of a bitmap index is illustrated with an example.

Let us consider table *clients* and a bitmap index created on its *sex* attribute, as shown in Fig. 1. Since the domain of the indexed attribute contains only two distinct values, the index is composed of two bitmaps. For example, the first bit of bitmap describing values *'female'* is set to 0 since the value of attribute *sex* of the first row is not a female.

Clients		bitmap index	
ID	sex	female	male
1	male	0	1
2	female	1	0
3	female	1	0
4	female	1	0
5	male	0	1
6	male	0	1
7	male	0	1
8	female	1	0
9	female	1	0
10	male	0	1
11	male	0	1
12	male	0	1
13	female	1	0
14	female	1	0
15	female	1	0
16	male	0	1
17	female	1	0
18	female	1	0
19	female	1	0

Fig. 1. An example *clients* table and a bitmap index created on attribute *sex*.

2.2. WAH and BBC compression

As mentioned previously, the WAH and BBC compression techniques are based on run-length encoding. The basic idea of run-length encoding consists in encoding continuous vectors of bits having the same value (either "0" or "1") into: (1) a common value of all bits in the vector (i.e., either "0" for a vector composed of zeros or "1" for a vector composed of ones) and (2) the length of the vector (i.e., the number of bits having the same value). Before encoding, a bitmap is divided into words. Next, words are grouped into the so-called runs. The *run* is composed of words that can be either a fill or a tail. The *fill* represents series of words that are composed of bits of the same value. The *tail* represents the series of words that are composed of both "0" and "1" bits. Fills are compressed because of their homogeneous content, whereas tails are not. The main difference between BBC and WAH is that BBC divides bit vectors into 8-bit words, whereas WAH divides them into 31-bit words. Moreover, BBC uses four different types of runs, depending on the length of a fill and the structure of a tail. WAH uses only one type of run.

The overall idea of the WAH compression is illustrated with an example. For the sake of simplicity let us assume that a 32-bit processor is used. A bitmap being compressed is composed of 5456 bits, as shown in Fig. 2a [26]. The WAH compression of the bitmap is executed in three following steps.

In the first step, the bitmap is divided into groups composed of 31 bits, as shown in Fig. 2b. In the example, 176 such groups are created. In the second step, adjacent groups containing identical bits are merged into one group, as shown in Fig. 2c. Since *group* 1 is heterogeneous, i.e., it is composed of "0" and "1" bits, it is not merged with a group following it. Groups 2–175 are homogeneous (composed of "0" bits) and they are merged into one large group, denoted in Fig. 2c as *group* 2–175. This group includes 174×31 bits. The last *group* 176, similarly as *group* 1, is heterogeneous and it cannot be merged with a group preceding it. As the result of group merging, three final groups are created, as shown in Fig. 2c.

In the third step, the three final groups are encoded on 32-bit words as follows (cf. Fig. 2d). The first group represents the tail of the first run. The most significant bit (the leftmost one) has value "0" denoting a tail. Next 31 bits are original bits of *group* 1. The second group (*group* 2–175) represents the fill of the second run. The most significant bit (at position^{31}) is set to "1" denoting a fill. The bit at position 2^{30} is set to "0" denoting that all bits in original *group* 2–175 have value "0", i.e., the fill is used for compressing groups whose all bits have value "0". The remaining 30 bits are used for encoding the number of homogeneous groups filled with "0." In the example, there are 174 such groups. The number of homogeneous groups is represented by the binary value equaled to 00000000000000000000010101110, stored on the remaining 30 bits. The last 31-bits group, denoted as *group* 176, represents the tail of the second run. The most significant bit in this group has value "0" denoting a tail. The remaining 31 bits are original bits of *group* 176.

2.3. Huffman encoding

In Huffman encoding [10], original symbols from a compressed file are replaced with bit strings. The more frequently a given symbol appears in the compressed file the shorter bit string is used for representing the symbol. Encoded symbols and their corresponding bit strings are represented as the so-called *Huffman tree*. Huffman tree is used for both compressing and decompressing. Huffman encoding algorithm is illustrated with an example.

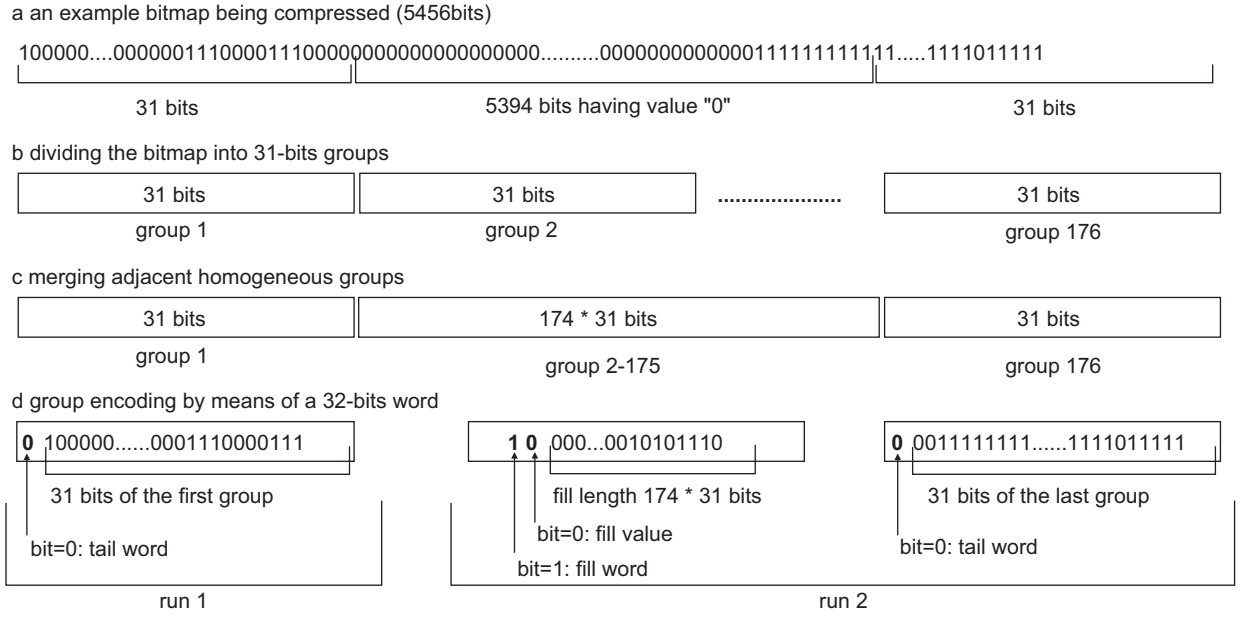

Fig. 2. The steps of the WAH compression.

Table 1
Frequencies of symbols in the compressed text 'this_is_test'.

Symbol	*t*	*s*	*i*	_	*h*	*e*
Frequency	3	3	2	2	1	1

Table 2
Encoded symbols and their Huffman codes.

Encoded symbol	*t*	*s*	*i*	_	*h*	*e*
Huffman code	11	00	01	100	1010	1011

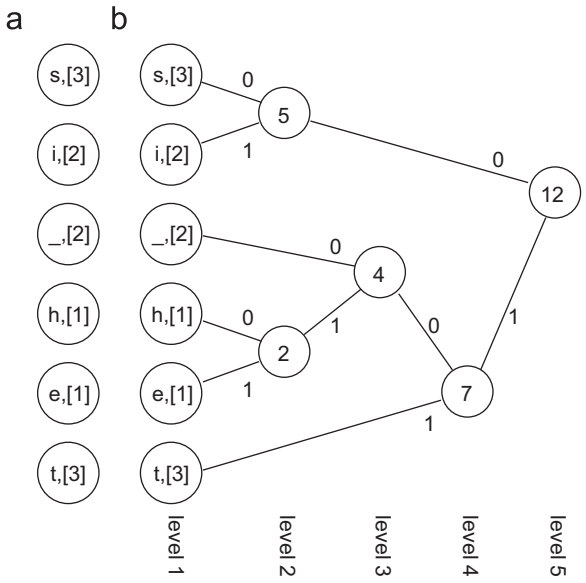

Fig. 3. An example Huffman tree.

Let us consider text 'this_is_test' that will be compressed by means of Huffman encoding. In the first step, the numbers of occurrences (also known as frequencies) of every symbol (i.e., a letter) in the compressed text are counted. Frequencies of the letters from our text are shown in Table 1.

Next, based on the frequencies, the set of nodes is built. Every node stores a symbol and its frequency (given in brackets in Fig. 3a), e.g., $(t,[3])$ denotes symbol *t* having the frequency of 3.

In the second step, Huffman tree is built from the nodes as follows. Two randomly selected nodes from level 1, representing the symbols of the lowest frequencies, are merged into one upper node at level 2. In our example, nodes $(h,[1])$ and $(e,[1])$ were merged, resulting in node (2), as shown in Fig. 3b. The value of this newly created node is equal to the sum of frequencies of its component nodes. Next, two other nodes of the lowest frequencies are merged, i.e., $(\_,[2])$ and (2). The merging procedure continues until a root node is obtained.

In the third step, edges connecting nodes are labeled with value "0" or "1." It is assumed that left-hand-side edges are labeled with "0" and right-hand-side edges are labeled with "1." This way, an encoded symbol from the lowest level is represented by an access path that starts from the root node and ends at the lowest level node storing the symbol. The access path is called a *Huffman code*. Huffman codes for the symbols from our example are shown in Table 2. Finally, all symbols from the encoded string are replaced with their Huffman codes.

Decompressing a file involves re-building Huffman tree from a stored (e.g., in the header of the compressed file) frequencies of symbols and converting a stream of bits into original characters, as follows. A stream of bits from the compressed file is used for traversing Huffman tree. Beginning at the root node and depending on the value of the bit, either the right or left branch of the tree is followed until reaching a leaf node. Then a symbol is read from the leaf node and it replaces the bit string used for navigating to this node. The procedure starts from the root for next bit.

3. RLH compression

RLH technique of compressing bitmaps proposed in this paper is based on run-length encoding and on Huffman encoding. There are two features of RLH that distinguish it from its competitors (BBC and WAH). Firstly, RLH counts distances between bits of value "1," rather than lengths of bit vectors of the same value, which is similar to delta encoding. Secondly, RLH does not divide bitmaps into words, i.e., the whole bitmap is compressed.

A *distance* between two bits of value "1" represents the number of bits of value "0" between these two bits. For example, in RLH, bit vector 000011110100 is encoded by the following sequence of digits: 400012. We assume that the beginning and end of the bit vector are interpreted as bits of value "1." This assumption does not have any impact on the concept of the RLH compression technique.

Code 400012 should be interpreted as follows. The first explicit "1" in the encoded bit vector 000011110100 is at a distance of four positions (bits) from "1" implicitly starting the vector at the leftmost position; the second "1" is at a distance of 0 bits from the closest "1" to the left; the third "1" is at a distance of 0 bits from the closest "1" to the left, etc.

Such a solution guarantees that when the density of bitmaps decreases, then the number of symbols used for encoding the bitmaps decreases too. Run-length encoding using distances will further be called a *modified run-length encoding*.

Bitmaps encoded by the modified run-length encoding are next compressed by Huffman encoding. The input values to Huffman encoding algorithm are the frequencies of all distances between bits of value "1" in all encoded bitmaps. A common Huffman tree is built for all frequencies and it is further used for encoding the distances.

In order to illustrate the RLH compression technique let us consider an example. Recall table *clients* and its bitmap index created on the *sex* attribute from the

a

the modified run length

female	male
1	0
0	3
0	0
3	0
0	2
3	0
0	0
0	3
1	3
0	
0	

b

frequencies of distances

distance	frequency
0	12
3	5
1	2
2	1

Fig. 4. The modified run-length encoding. (a) The modified run length; (b) frequencies of distances.

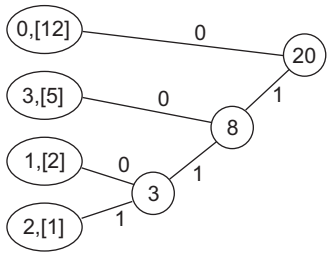

Fig. 5. Huffman tree for the symbols from Fig. 4b.

symbol	symbol code
0	0
3	10
1	110
2	111

Fig. 6. The symbols from Table 4b and their corresponding Huffman codes.

example discussed in Section 2.1. Compressing both bitmaps by means of RLH is executed as follows.

Step 1: The compression procedure starts with encoding each of the two bitmaps (i.e., *sex* = '*female*', *sex* = '*male*') separately by means of the modified run-length encoding. As a result, two sequences of digits are obtained, as shown in Fig. 4a. These sequences represent distances between bits of value "1" in the encoded bitmaps. Next, the frequencies of the distances are counted in both bitmaps, as shown in Fig. 4b. In our example, distance "0" appears 12 times in both bitmaps, distance "3" appears 5 times, etc.

Step 2: The frequencies of distances obtained in step 1 are encoded by means of Huffman algorithm, i.e., one Huffman tree is built for the purpose of obtaining Huffman codes of all distances. Huffman tree for the distances from Fig. 4b is shown in Fig. 5. Based on Huffman tree, Huffman codes for the distances are computed, as shown in Fig. 6.

Step 3: The distances shown in Fig. 4a are replaced by their Huffman codes, as shown in Fig. 7, ending the compression procedure.

The size of Huffman tree impacts the performance of bitmap compression and decompression. From the performed experiments it turns out that the size of Huffman tree is small. For example, for a test table storing 100,000,000 rows and the cardinality of an indexed attribute equal to 20,000, the size of Huffman tree ranges from 71 to 92 kB, depending on the distribution of values of an indexed attribute. For this reason, Huffman tree can be easily stored in main memory, greatly improving bitmap compression and decompression efficiency.

3.1. Decompressing bitmaps

The process of decompressing bitmaps compressed with RLH is a standard one, i.e., it uses Huffman tree, which is kept in main memory. In order to decompress a bitmap, subsequent bits of the compressed bitmap are

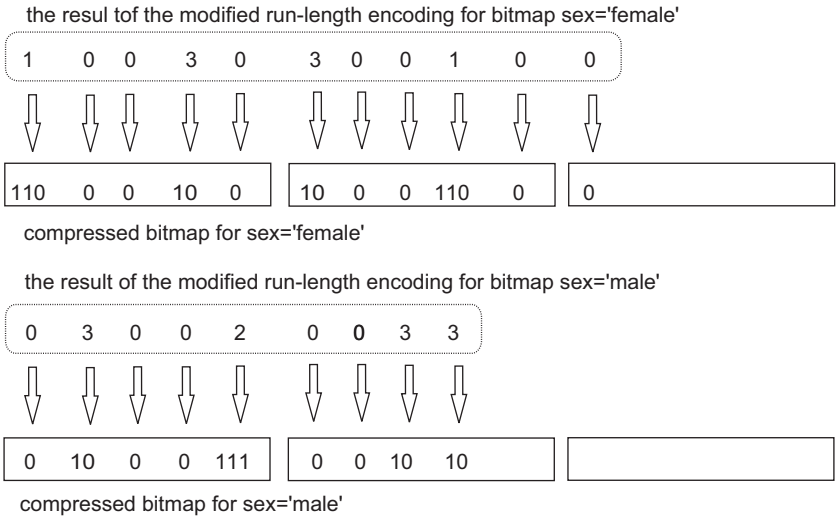

Fig. 7. Bitmaps on attribute *sex* compressed with RLH.

read by the decompression algorithm. The bits are used for the navigation in Huffman tree from its root to leaves. The symbol (distance) read from a leaf replaces the Huffman code represented by the navigation path from the root to a leaf. When the whole bitmap is decompressed, i.e., it is in a form of the sequence of distances between bits of value "1," then the distances are converted into original bit strings, which ends the decompression procedure.

Decompressing bitmaps compressed by RLH requires a larger number of operations than in the case of WAH and BBC. Despite this, its performance is good since Huffman tree is stored in main memory.

3.2. Updating bitmaps

Updating bitmaps compressed with RLH is more complicated than updating bitmaps compressed with WAH and BBC. In WAH, a compressed bitmap can be modified without decompressing the whole bitmap. In order to modify a bitmap compressed with RLH one has to: (1) decompress the whole bitmap, (2) modify the bitmap, and (3) compress the bitmap again. All the three operations are required since updating a bitmap changes frequencies of distances in the modified run-length encoding. If an old Huffman tree was used, then one could obtain bitmaps compressed non-optimally. Additionally, new distances (not existing in an old Huffman tree) might appear as the result of a bitmap update.

These kinds of problems are known in the research literature. They are partially solved by means of dynamic Huffman algorithms [28]. Unfortunately, these algorithm are computationally too complex to be applied in DWs.

The RLH compression technique will be suitable in DWs where index structures are dropped before loading a DW, and they are rebuilt from scratch at the end of the data loading process. For such DWs, the aforementioned limitation of the RLH compression technique is less important. A bitmap index will be created and compressed from scratch after every loading of a DW.

However, in order to support the modification of RLH compressed bitmap indexes for the application in DWs that do not rebuild indexes from scratch and for other applications, we proposed a modified RLH compression technique, called RLH-*N*.

3.3. RLH-N compression

In the RLH-*N* compression technique, a bitmap being compressed is divided into *N*-bit words, which is the first difference as compared to RLH. The length of a word is parameterized. Next, for each word, the frequencies of distances are counted, similarly as in RLH. All frequencies from all words are used for constructing a common Huffman tree. Thus, there is one common set of Huffman codes for all *N*-bit words. Additionally, the Huffman tree contains all the actually existing distances as well as all the possible distances that can appear in an *N*-bit word, which is the second difference as compared to RLH. The distances that do not exist in compressed bitmaps are assigned the frequency of one and, as a consequence, they obtain the longest Huffman codes.

The RLH-*N* compression has the following advantages as compared to RLH. Firstly, including all the possible distances in the Huffman tree eliminates the need for rebuilding the tree after such a bitmap update that results in a new distance. Secondly, *N*-bit words can be read and processed in parallel. Finally, in order to update a bitmap, only an appropriate *N*-bit word has to be read and uncompressed.

In the current implementation we parameterized the length of an *N*-bit word with values 256, 512, 1024, and 2048 bits. However, its length should be tailored to the length of words used by a CPU as well as to the distribution pattern of values of an indexed attribute. Selecting the most efficient length of a bit section is the subject of our future work.

4. Experimental evaluation

The RLH and RLH-*N* compression techniques were implemented and evaluated experimentally. The characteristics of RLH and RLH-*N* were compared to the characteristics of WAH and uncompressed bitmaps. The experiments were conducted on a PC equipped with AMD Athlon(TM) XP 2500+ CPU, Seagate 80 GB hard disk, and 768 MB RAM. The PC was operated by Ubuntu Linux. All tested compression techniques were implemented in C. Data and bitmap indexes (compressed and uncompressed ones) were stored on disk in operating system files, and they were not managed by any database management system.

Tests were performed on a dataset reflecting a real DW being built for the largest Internet auction provider in Eastern Europe. The dataset that we use includes 100,000,000 rows. It reflect the size of a fact table in a two-year time window. One of dimensions used in this DW includes over 12,000 items and it grows from year to year. In our experiments, bitmap indexes were defined on attributes of type *integer* having a parameterized cardinality, ranging from 2 to 20,000 distinct values. Depending on the experiment, indexed rows were either totally ordered by the value of an indexed attribute, or partially ordered, or not ordered at all.

In our tests, we decided not to use either the standard TPC-H or TPC-DS benchmarks as they are typically designed to measure an overall system's response time and query processing efficiency, whereas our aim was to test some performance characteristics of compression techniques.

The tests aimed at comparing the characteristics of RLH and RLH-*N* to WAH and to an uncompressed bitmap index (UBI) with respect to:

- the size of bitmaps;
- the response time of a query involving an indexed attribute, for randomly ordered, totally ordered, and partially ordered indexed dataset;
- the efficiency of compressed bitmap modifications.

4.1. Bitmap sizes

This experimental scenario was designed in order to compare size characteristics of RLH, RLH-*N*, WAH, and UBI. In this scenario, indexed rows were randomly ordered by the value of an indexed attribute. The results are shown in Fig. 8. Notice that the *X*- and *Y*-axes are logarithmic.

From Fig. 8 we can observe that for all tested attribute cardinalities greater than 2 bitmaps compressed with RLH are smaller than corresponding bitmaps compressed with WAH. For the data visualized in Fig. 8 we computed a bitmap size ratio s_{WAH}/s_{RLH}, where s_{WAH} and s_{RLH} are bitmap sizes compressed with WAH and RLH, respectively. With the increase of the cardinality of an indexed attribute the value of s_{WAH}/s_{RLH} increases from 0.98 (for attribute cardinality equal to 2) to 6 (for attribute cardinality equal to 100) to 7.6 (for attribute cardinality equal to 1000). Next, the value of s_{WAH}/s_{RLH} oscillates around 7.50–7.53 for higher cardinalities (from 1000 up to 20,000).

Such a characteristic can be explained as follows. When the cardinality of an indexed attribute increases, then the length of homogeneous vectors of bits "0" in bitmaps grows. Such homogeneous vectors are well compressed by WAH and RLH, however, in such cases RLH requires less bits for encoding compressed bitmaps than WAH.

From Fig. 8 we can also observe that the size of a bitmap compressed with RLH-*N* increases when the word length decreases. For the word of length equal to 256 bits ($N = 256$), bitmaps compressed with RLH-*N* are smaller than bitmaps compressed with WAH for cardinalities from 3 to 500. For the word of length equal to 512 bits ($N = 512$) bitmaps compressed with RLH-*N* are smaller than bitmaps compressed with WAH for cardinalities from 3 to 2240. For $N = 1024$ bitmaps compressed with RLH-*N* are smaller than bitmaps compressed with WAH for cardinalities from 3 to 3920. Finally, for $N = 2048$ bitmaps compressed with RLH-*N* are smaller than bitmaps compressed with WAH for cardinalities from 3 to 12,200.

4.2. Query response time

In these experiments we measured the performance characteristics of UBI and bitmaps compressed with RLH, RLH-*N*, and WAH with respect to the response time of a query. The query had the following pattern:

```
select A1, A2, A3, I1 from T
where I1 in (v1, v2, ..., v100);
```

where Ai ($i = 1, 2, 3$) and $I1$ denote attributes of text and integer values, respectively; $I1$ represents a foreign key attribute having defined a bitmap index. The cardinality of $I1$ ranges from 2 to 20,000 distinct values. $v1, v2, \ldots, v100$ represent 100 randomly selected values of $I1$. Notice that for $I1$ cardinalities lower than 100 the same bitmap was processed multiple times.

Notice that query response times for UBI, RLH, RLH-*N*, and WAH do not include times of fetching data records from disk. In all of these cases an identical number of records will be read. Therefore, we treat it as a constant value not having any impact on the comparison of the compression techniques.

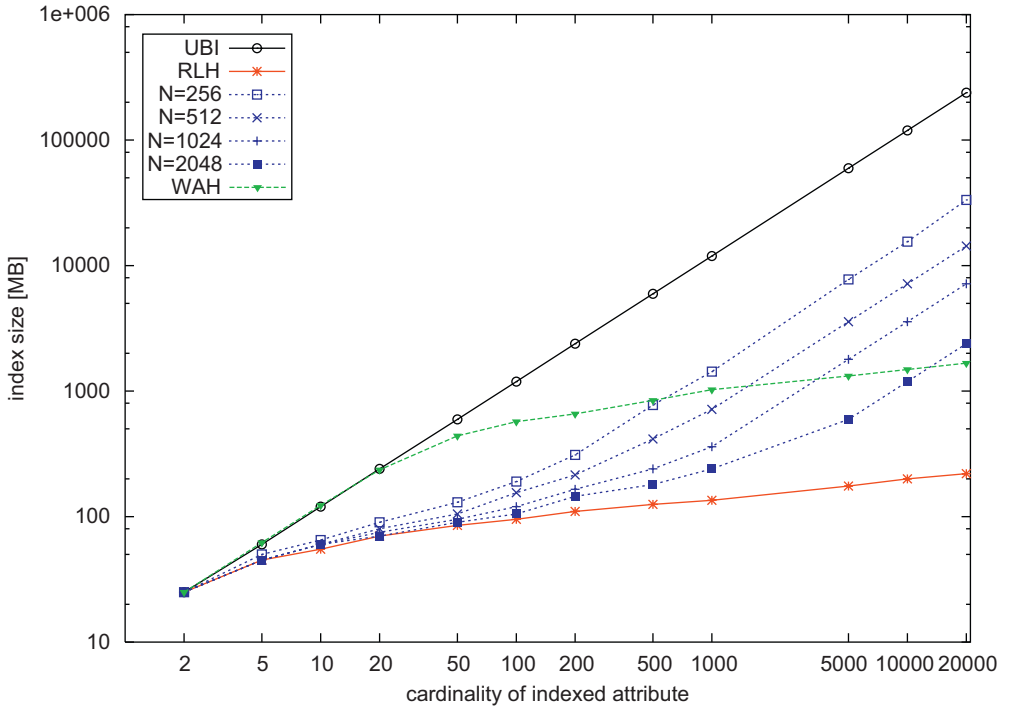

Fig. 8. Comparison of RLH, RLH-*N*, WAH, and UBI with respect to the size of a bitmap index (the number of rows in an indexed dataset: 100,000,000; the cardinality of an indexed attribute: 2–20,000 distinct values; $N = \{256, 512, 1024, 2048\}$ for RLH-*N*).

The performance characteristics include the total time spent on: (1) fetching appropriate bitmaps from disk that are required to answer the query and (2) computing the final bitmap. The total time is composed of CPU time and I/O time. The CPU time involves operations in RAM, i.e., decompressing bitmaps, logical operations on bitmaps. The I/O time involves reading compressed bitmaps from disk. In the implementation code, first the entire bitmap is read from a disk, and then it is processed in RAM. For the purpose of measuring time the `gettimeofday` function was used.

The performance characteristics were evaluated for three cases of row ordering. In the first case, the order of rows was totally random with respect to the value of an indexed attribute. In the second case, rows were ordered by the value of an indexed attribute. In the third case, a parameterized number of rows was ordered by the value of an indexed attribute.

4.2.1. Unordered dataset

For unordered indexed dataset, total query response times (including the CPU and I/O time), are shown in Fig. 9. As we can observe from the figure, RLH and RLH-*N* offer shorter query processing times for certain cardinalities of an indexed attribute, as compared to WAH and UBI.

For the data visualized in Fig. 9 we computed ratio t_{WAH}/t_{RLH}, where t_{WAH} and t_{RLH} are total query response times using the WAH-compressed and RLH-compressed bitmaps, respectively. The ratio increases from 0.2 (for the cardinality equal to 2) to 1 (for the cardinality equal to 20). Then, for cardinalities ranging from 21 to 180, t_{WAH}/t_{RLH} varies from 1.34 to 1.20. For cardinalities greater than 180, t_{WAH}/t_{RLH} decreases from 0.87 to 0.35.

From Fig. 9 we may also observe that RLH-*N* offers even shorter query response times than RLH. For $N = 256$, the query response time is shorter than for WAH with the range of attribute cardinality from 17 to 1000. For $N = 512$, the query response time is shorter than for WAH with the range of attribute cardinality from 20 to 3100. For $N = 1024$ and 2048, the query response time is shorter than for WAH with the range of attribute cardinality from 20 to 5000 and from 20 to 10,000, respectively.

Such characteristics of RLH and RLH-*N* are caused by higher sparsity of bits equal to "1" in bitmaps for domains of an indexed attribute of higher cardinality. As a consequence, in order to encode distances between neighbor "1," fewer symbols are needed. This, in turn, results in smaller Huffman tree. Smaller Huffman tree causes that bitmap compression and decompression to be faster than for larger Huffman tree. On the other hand, with an increase of attribute cardinality, the possible number of distances increases that results in larger Huffman tree. Larger Huffman tree causes that compression is less efficient with respect to time since longer paths in Huffman tree have to be searched.

For WAH, when average distances between neighbor "1" bits become longer than 31 bits, a bitmap contains more fills composed of "0." Bitmaps of such a characteristic are well compressed by WAH and they offer higher processing efficiency. The efficiency of WAH increases when the length of homogeneous bit vectors increases. Such vectors are included in fills.

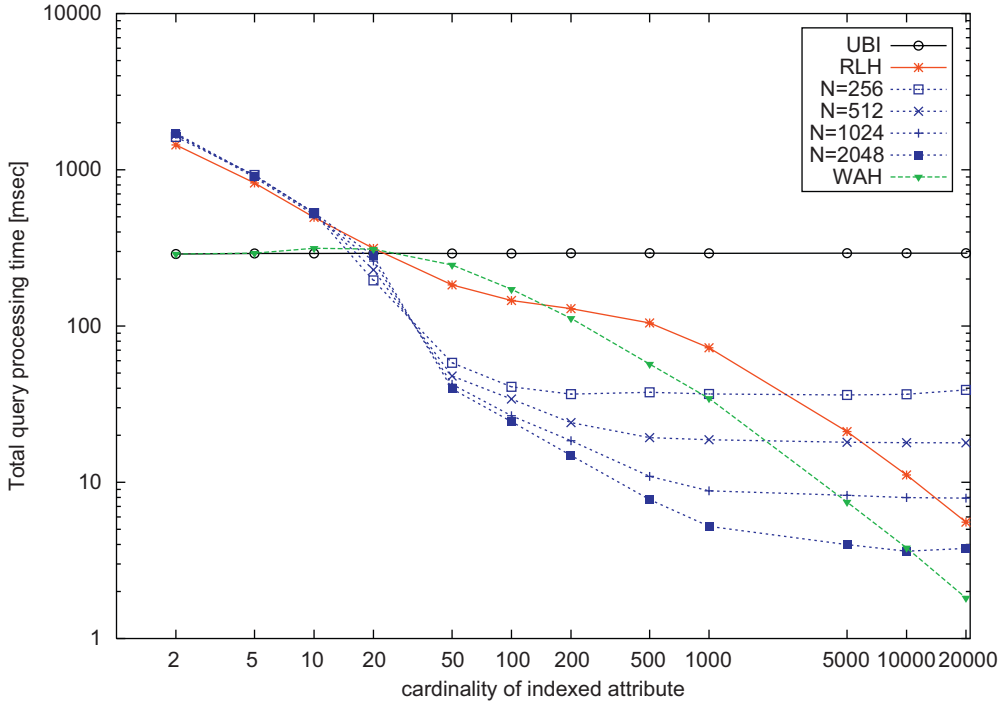

Fig. 9. Comparison of RLH, RLH-*N*, WAH, and UBI with respect to a total query response time (the number of rows in an indexed dataset: 100,000,000; the cardinality of an indexed attribute: 2–20,000 distinct values; $N = \{256, 512, 1024, 2048\}$ for RLH-*N*).

For the cardinality of an indexed attribute lower than 20, RLH offers worse performance than WAH and UBI. Such a characteristic is explained as follows. For domains of lower cardinality, bitmaps are dense, i.e., they store large number of bits equal to "1." As a consequence, average distances between neighbor "1" bits are short and there are many such distances. For this reason, bitmaps compressed with RLH are larger (having a size similar to the size of WAH-compressed bitmaps) and that, in turn, results in more I/O operations for reading a compressed bitmap. Moreover, the size of Huffman tree increases when the cardinality of an indexed attribute decreases. As a consequence, the decompression of RLH-compressed bitmaps takes longer time.

Fig. 10 presents characteristics of RLH, RLH-*N*, WAH, and UBI with respect to the CPU processing times. As we can observe from this figure, the CPU processing time of WAH is lower than RLH for the whole range of attribute cardinality. The CPU processing time of RLH-*N* is shorter than of WAH for certain ranges of attribute cardinality, i.e., RLH-256 performs better than WAH for cardinalities from 20 to 2000, RLH-512 performs better for cardinalities from 25 to 4000, RLH-1024 performs better for cardinalities from 25 to 6500, and RLH-2048 performs better for cardinalities from 25 to 14,500.

On the contrary, the I/O processing time of WAH is higher than of RLH for the whole range of attribute cardinality, as shown in Fig. 11. It is a consequence of larger sizes of WAH-compressed bitmaps than RLH-compressed bitmaps. I/O processing times of RLH-*N*-compressed bitmaps reflect the characteristics of their sizes.

In our opinion, the better performance of RLH-*N* than RLH may be caused by parallel operations on disk and CPU, i.e., a compressed *N* bit word can be decompressed in parallel with reading another compressed word. For shorter compressed words, which is the case of RLH-*N*, this parallel processing happens more frequently than for long compressed words, which is the case of RLH. Moreover, dividing a bitmap into words results in smaller Huffman tree. For example, for $N = 2048$, maximum number of symbols in Huffman tree equals only 2048. The smaller Huffman tree, the shorter CPU time spent on decompressing a bitmap.

4.2.2. *Ordered and partially ordered dataset*

This experimental scenario was designed in order to test how response times of RLH, RLH-*N*, and WAH depend on the order of rows in an indexed dataset. To this end, four different cases were considered. In the first case, rows were totally ordered by the values of an indexed attribute. In the second case, 98% of rows were ordered by the values of an indexed attribute, and 2% of randomly selected rows remained unordered. In the third and fourth cases, 5% and 10% of randomly selected rows remained unordered, respectively. The response time characteristics were measured for the CPU and I/O time, for the same query as discussed earlier.

Figs. 12 and 13 show the obtained CPU time and I/O time characteristics, respectively. In these figures, from the family of RLH-*N*, only the characteristics of RLH-1024 are shown. For other word lengths (i.e, 256, 512, 2048) the characteristics are similar to the presented ones. From these figures we can observe that when the number of

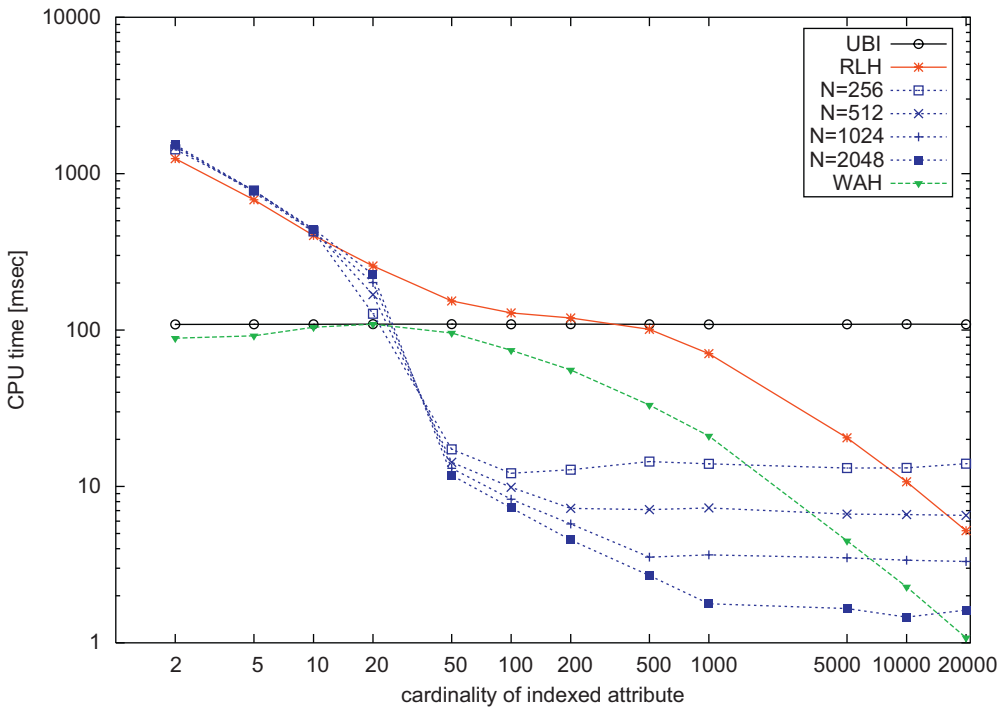

Fig. 10. Comparison of RLH, RLH-*N*, WAH, and UBI with respect to the CPU processing time (the number of rows in an indexed dataset: 100,000,000; the cardinality of an indexed attribute: 2–20,000 distinct values; $N = \{256, 512, 1024, 2048\}$ for RLH-*N*).

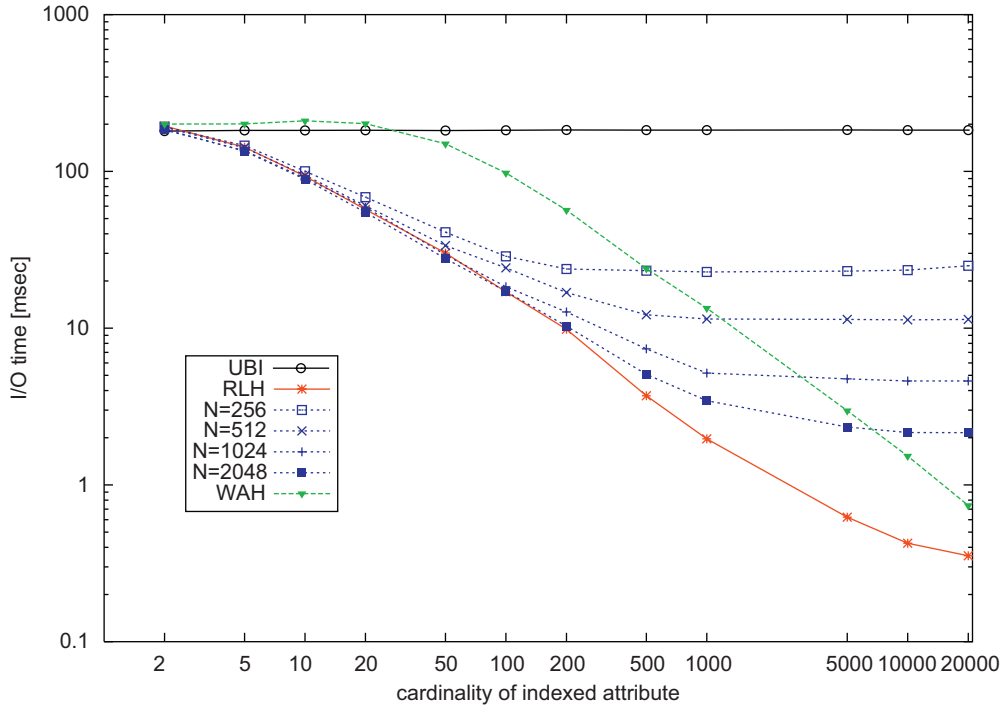

Fig. 11. Comparison of RLH, RLH-*N*, WAH, and UBI with respect to the I/O processing time (the number of rows in an indexed dataset: 100,000,000; the cardinality of an indexed attribute: 2–20,000 distinct values; $N = \{256, 512, 1024, 2048\}$ for RLH-*N*).

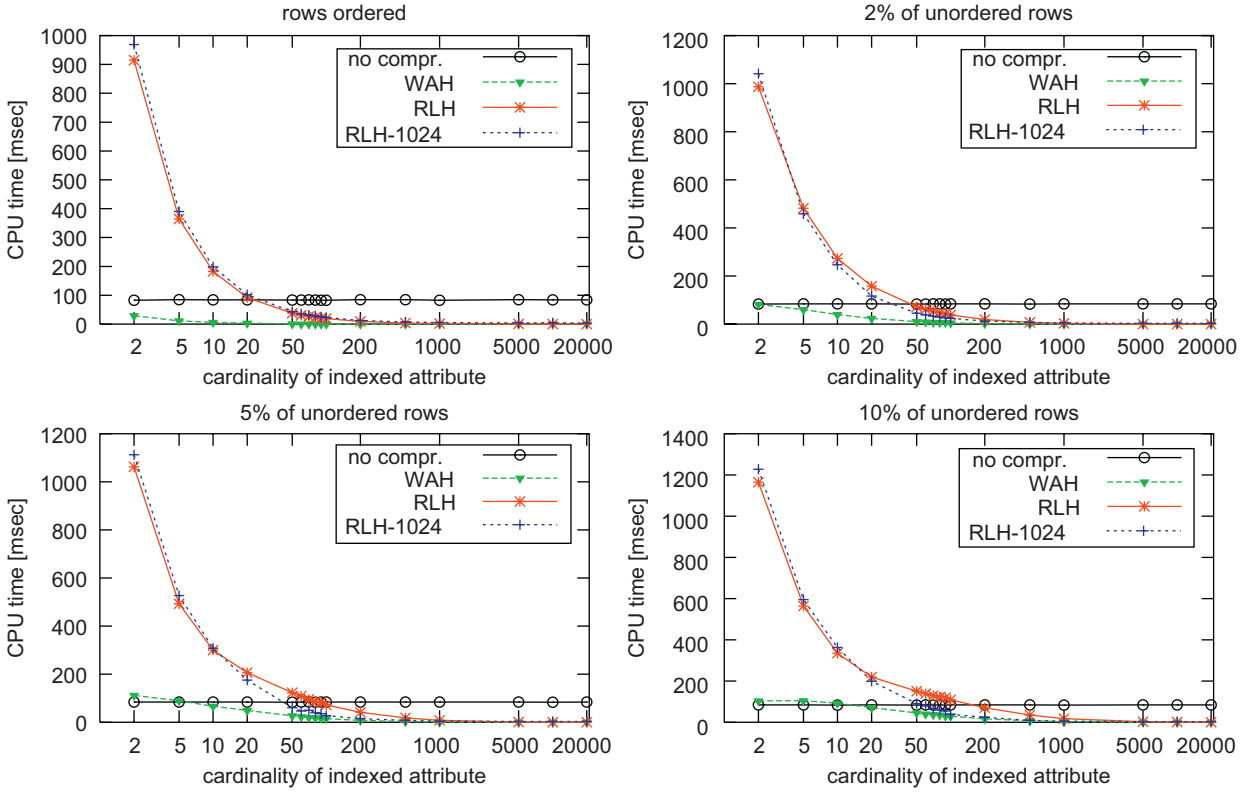

Fig. 12. Comparison of RLH, RLH-1024, WAH, and UBI with respect to the CPU processing time (the number of unordered rows in an indexed dataset: 0%, 2%, 5%, 10%; the number of rows in an indexed dataset: 100,000,000; the cardinality of an indexed attribute: 2–20,000 distinct values).

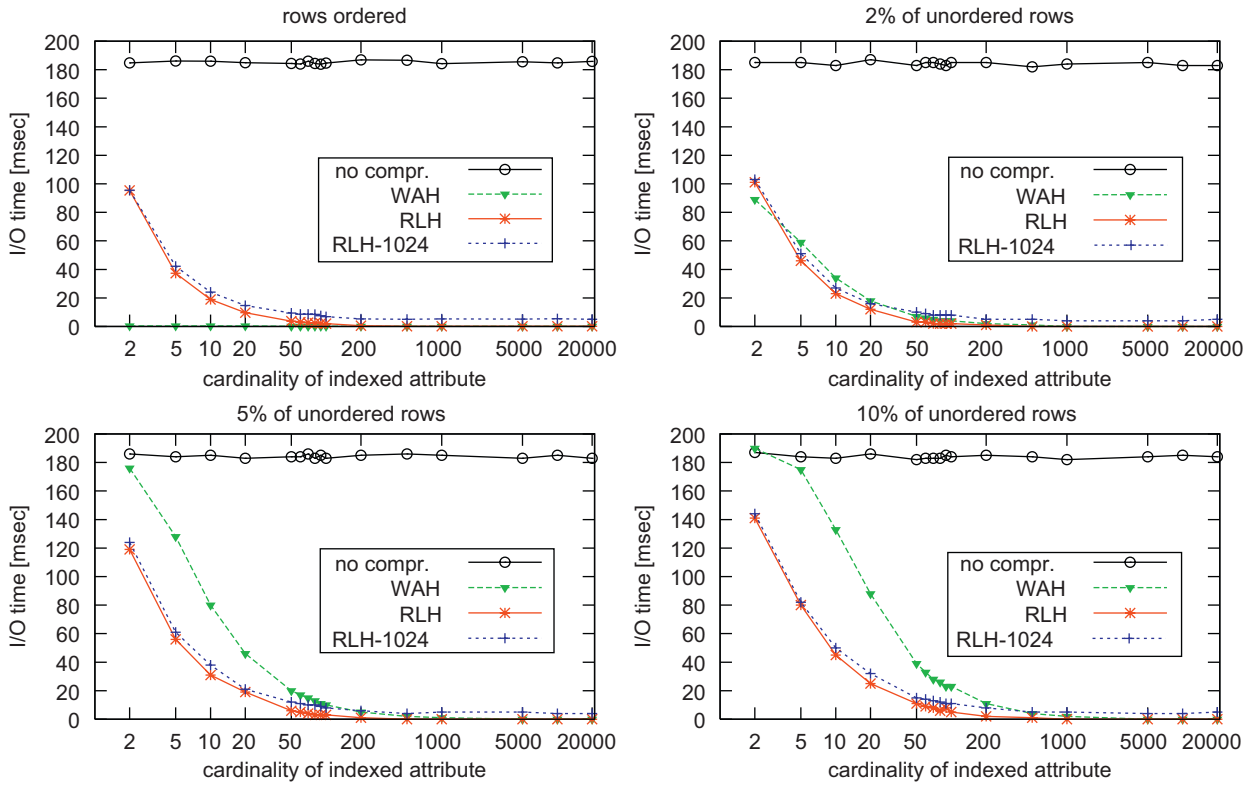

Fig. 13. Comparison of RLH, RLH-1024, WAH, and UBI with respect to the I/O processing time (the number of unordered rows in an indexed dataset: 0%, 2%, 5%, 10%; the number of rows in an indexed dataset: 100,000,000; the cardinality of an indexed attribute: 2–20,000 distinct values).

unordered rows increases in the indexed dataset, then the CPU and I/O response times increase faster for WAH than for RLH and RLH-1024. From this observation we can conclude that WAH is more sensitive to rows unordering than RLH and RLH-1024 (RLH-*N* in general).

For the purpose of quantifying the increase of the CPU and I/O processing times with respect to the number of unordered rows, we computed time ratios for the CPU and I/O time characteristics. The values of these ratios are presented in Fig. 14.

Let $tCPU^{Compr}_{\%unord}$, represent the CPU processing time, where *Compr* represents one of the bitmap compression techniques (i.e., either RLH, or RLH-1024, or WAH), and *%unord* represents the percent of unordered rows in an indexed dataset (i.e., either 2%, or 5%, or 10%). For example, $tCPU^{WAH}_{2\%}$ and $tCPU^{WAH}_{5\%}$ stand for CPU processing time of WAH for 2% and 5% of unordered rows, respectively. Let further $tCPU^{Compr}_{ord}$ represent the CPU processing time for totally ordered rows in an indexed dataset, either for RLH, or RLH-1024, or WAH-compressed bitmaps.

The CPU processing time ratio is defined as: $tCPU^{Compr}_{\%unord}/tCPU^{Compr}_{ord}$. Thus, nine ratios ($tCPU^{RLH}_{2\%}/tCPU^{RLH}_{ord}$, $tCPU^{RLH}_{5\%}/tCPU^{RLH}_{ord}$, $tCPU^{RLH}_{10\%}/tCPU^{RLH}_{ord}$, $tCPU^{WAH}_{2\%}/tCPU^{WAH}_{ord}$, ...) were computed for the CPU processing time. In a similar way we computed nine ratios $tIO^{Compr}_{\%unord}/tIO^{Compr}_{ord}$ for the I/O processing time.

Values of $tCPU^{Compr}_{\%unord}/tCPU^{Compr}_{ord}$ are shown in Fig. 14. As we can observe from the figure, values of the ratios are much higher for WAH than for RLH and RLH-1024. For example, for WAH, $tCPU^{WAH}_{10\%}/tCPU^{WAH}_{ord}$ increases from 3.6 (for attribute cardinality equal to 2) to 302 (for cardinality equal to 500). Then, its value decreases to 29 for cardinality equal to 20,000. For the same percent of unordered records (10%), values of $tCPU^{RLH}_{10\%}/tCPU^{RLH}_{ord}$ and $tCPU^{RLH-1024}_{10\%}/tCPU^{RLH-1024}_{ord}$ range from 1.27 to 10.54 and from 1.27 to 2.25, respectively.

Values of $tIO^{WAH}_{\%unord}/tIO^{WAH}_{ord}$ (cf. Fig. 14) are also much higher for WAH than for RLH and RLH-1024. For example, for WAH, $tIO^{WAH}_{10\%}/tCPU^{WAH}_{ord}$ ranges from 539 (for attribute cardinality equal to 2) to 1.6 (for cardinality equal to 20,000). For the same percent of unorderd records, values of $tCPU^{RLH}_{10\%}/tCPU^{RLH}_{ord}$ and $tCPU^{RLH-1024}_{10\%}/tCPU^{RLH-1024}_{ord}$ range from 1.5 to 6.5 and from 1.02 to 2.2, respectively.

4.3. Updating bitmaps

This experimental scenario was designed in order to test the efficiency of updating bitmaps compressed with RLH and RLH-*N*. The number of updated rows was equal to 10%. Updated rows were selected randomly.

The overall update time characteristics (including the CPU and I/O times) are shown in Fig. 15. Analyzing the characteristics we can observe that dividing a bitmap into words substantially reduces a compressed bitmap update time. This confirms what could be expected since long words (one word in the case of RLH) result in more data

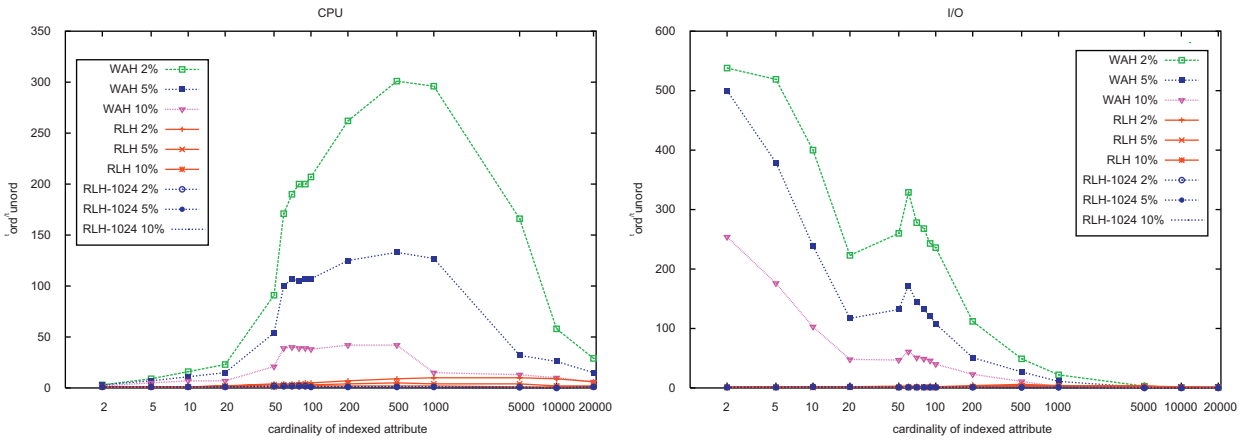

Fig. 14. Comparison of time ratios for the characteristics shown in Figs. 12 and 13.

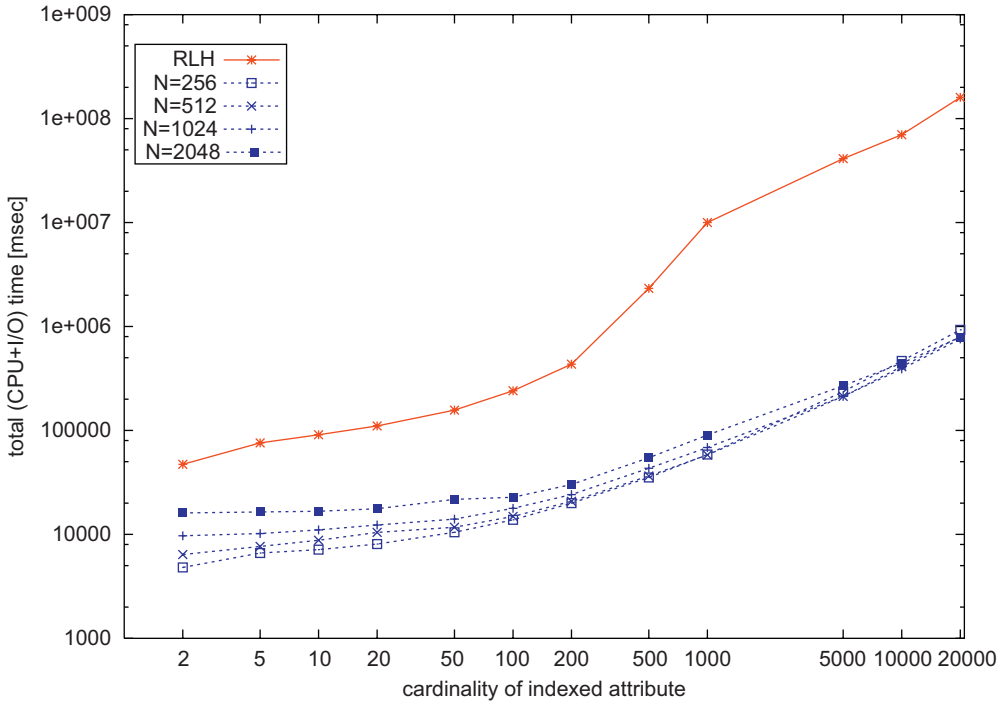

Fig. 15. The efficiency of updating bitmaps compressed with RLH and RLH-*N* (the number of rows in an indexed dataset: 100,000,000; the cardinality of an indexed attribute: 2–20,000 distinct values; $N = \{256, 512, 1024, 2048\}$ for RLH-*N*; 10% of updated rows).

that have to be read from disk, have to be decompressed, updated, and compressed again.

For the data visualized in Fig. 15 we computed ratio $t_{RLH}/t_{RLH\text{-}N}$, where t_{RLH} and $t_{RLH\text{-}N}$ are total update times of a RLH-compressed and RLH-*N*-compressed bitmap, respectively. $t_{RLH}/t_{RLH\text{-}256}$ ranges from 9.8 (for the cardinality equal to 2) to 172 (for the cardinality equal to 20,000). $t_{RLH}/t_{RLH\text{-}512}$ ranges from 7.4 (for the cardinality equal to 2) to 200 (for the cardinality equal to 20,000). $t_{RLH}/t_{RLH\text{-}1024}$ ranges from 4.9 (for the cardinality equal to 2) to 211 (for the cardinality equal to 20,000). $t_{RLH}/t_{RLH\text{-}2048}$ ranges from 2.9 (for the cardinality equal to 2) to 202 (for the cardinality equal to 20,000).

Recall that for RLH-*N* the Huffman tree contains all the actually existing distances as well as all the possible distances that can appear in an *N*-bit word (the distances that do not exist in compressed bitmaps are assigned the frequency of one). This way, rebuilding the Huffman tree can be delayed until Huffman codes become non-optimal and as a consequence, a query response time drops down below a certain threshold. This threshold could be defined by a user. The fact that the Huffman tree does not have to

be rebuild after every single row update reduces a bitmap update time.

4.4. Conclusive remarks

In this section we present the experimental evaluation of WAH, RLH, and RLH-*N* with respect to: (1) the size of bitmap indexes, (2) query processing time, and (3) update time of bitmap indexes.

Size characteristics: Comparing the sizes of bitmap indexes we can observe that for all the tested attribute cardinalities greater than 2, bitmaps compressed with RLH are smaller than corresponding bitmaps compressed with WAH. RLH-compressed bitmaps are more than six times smaller than WAH-compressed bitmaps for the attributes of cardinality greater than 100.

For RLH-*N*-compressed bitmaps we observe that a bitmap size increases when the length of an *N*-bit word decreases. From the tested lengths of *N*-bit words $N = \{256, 512, 1024, 2048\}$ RLH-2048 offers the best compression ratio. Bitmaps compressed with RLH-2048 are smaller than bitmaps compressed with WAH for cardinalities ranging from 3 to 12,200.

Time characteristics: Query processing time characteristics were computed for an unordered indexed dataset, for an indexed dataset totally ordered by the value of an indexed attribute, and for a partially ordered dataset. The total processing time characteristics, including I/O and CPU times, show that RLH-compressed bitmaps offer the total processing time shorter than WAH only for certain cardinalities (ranging from 21 to 180). We may also observe that RLH-*N* offers shorter query response times than RLH. Moreover, the longer an *N*-bit word the wider the range of cardinality, where RLH-*N* offers better performance than WAH. For example, for $N = 256$, the query processing time is shorter than for WAH within the range of attribute cardinality from 17 to 1000 but for $N = 2048$ this range extends from 20 to 10,000.

Analyzing the CPU processing time we can observe that for WAH it is lower than for RLH for the whole range of attribute cardinalities. The CPU processing time of RLH-*N* is shorter than of WAH only for certain ranges of attribute cardinality. The widest range of cardinality (from 25 to 14,500) where RLH-*N* performs better than WAH was achieved for $N = 2048$.

As far as the I/O processing time is concerned, we can observe that for WAH it is higher than for RLH for the whole range of attribute cardinalities. The I/O processing time of RLH-*N* is shorter than of WAH for these ranges of attribute cardinality where sizes of RLH-*N*-compressed bitmaps are smaller than WAH-compressed ones.

We have also verified the impact of the ordering of indexed rows on the time characteristics of RLH, RLH-*N*, and WAH-compressed bitmaps. From the experiments we can observe that when the number of unordered rows increases in the indexed dataset, then CPU and I/O response times increase faster for WAH than for RLH and RLH-*N*. From this observation we can conclude that WAH is more sensitive to rows unordering than RLH RLH-*N*.

Bitmap update characteristics: Analyzing the time spent on updating RLH and RLH-*N*-compressed bitmaps we can observe that dividing a bitmap into words substantially reduces update time. The experiments showed that the update time increases when the word length increases. The lowest update times were achieved for $N = 256$. In this case update times were shorter more than 170 times as compared to RLH for cardinalities greater than 1000. For cardinalities lower than 1000 update times were shorter from 7 to over 60 times.

5. Summary

In this paper we presented an alternative compression technique for bitmap indexes for the application in DWs. The compression technique that we proposed is based on run-length encoding and on Huffman encoding. We developed the compression technique in two variants, namely RLH and RLH-*N*. Both techniques were implemented and experimentally compared to the widely accepted WAH compression technique.

From the experiments we draw the following conclusions.

(1) RLH-compressed bitmaps are much smaller (up to 7.52 times) than corresponding WAH-compressed bitmaps for attribute cardinalities greater than 20.
(2) From the family of RLH-*N* compression techniques, RLH-2048 offers the most promising characteristics. First, RLH-2048-compressed bitmaps are smaller than WAH-compressed bitmaps in the range of attribute cardinalities from 4 to 10,000. Second, RLH-2048 offers shorter query processing times than WAH in the range of attribute cardinalities from 20 to 10,000. Finally, RLH-2048 offers much lower (from 2.09 to 202 times) updating time of compressed bitmaps than RLH.

The obtained results show that the RLH and RLH-*N* compression techniques offer better performance than WAH for certain cardinalities of an indexed attribute and that they are less sensitive to row ordering than WAH. Therefore, we believe that RLH and RLH-*N* can be complementary compression techniques to WAH. Ideally, these different compression techniques could be used in the same system for cardinalities for which they offer the best performance.

Currently we are developing an analytical cost model of RLH and RLH-*N* for the purpose of analytically comparing various compression techniques. In the future we plan to: (1) compare the compression ratio and data access characteristics of RLH, RLH-*N*, and WAH, based on nine-year time window data from an auction DW built for a major Eastern Europe Internet auction system, (2) experimentally compare RLH, WAH, BBC, approximate encoding, and FastBit, (3) optimize the RLH-*N* compression technique with additional data structures offering quick access to the *N*-bit words, (4) develop a method for automatic selection of the most appropriate section length for RLH-*N* with respect to a given data distribution, query characteristics, update characteristics, and used

hardware, (5) transfer the ideas of compressing techniques developed for text databases into the bitmap compression techniques, and (6) apply in experiments query patterns from the set query benchmark [16] and query patterns from analytical applications built for the auction DW.

Acknowledgments

The authors acknowledge anonymous reviewers for their constructive and thorough comments that greatly improved the quality of this paper.

References

[1] A. Amir, G. Benson, M. Farach, Let sleeping files lie: pattern matching in z-compressed files, Journal of Computer and Systems Sciences 52 (2) (1996) 299–307.
[2] G. Antoshenkov, M. Ziauddin, Query processing and optimization in Oracle RDB, VLDB Journal 5 (4) (1996) 229–237.
[3] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, A.S. Tosun, Approximate encoding for direct access and query processing over compressed bitmaps, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2006.
[4] C. Chan, Y. Ioannidis, Bitmap index design and evaluation, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1998.
[5] K.C. Davis, A. Gupta, Indexing in data warehouses: bitmaps and beyond, in: R. Wrembel, C. Koncilia (Eds.), Data Warehouses and OLAP: Concepts, Architectures and Solutions, Idea Group Inc., 2007, pp. 179–202 ISBN 1-59904-364-5.
[6] E.S. de Moura, G. Navarro, N. Ziviani, R. Bayeza-Yates, Fast and flexible word searching on compressed text, ACM Transactions on Information Systems 18 (2) (2000) 113–139.
[7] P. Elias, Universal codeword sets and representations of the integers, IEEE Transactions on Information Theory 21 (2) (1975) 194–203.
[8] M. Farach, M. Thorup, String matching in Lempel–Ziv compressed strings, in: Proceedings of ACM Annual Symposium on the Theory of Computing, 1995.
[9] S.W. Golomb, Run-length encodings, IEEE Transactions on Information Theory 12 (3) (1966) 399–401.
[10] D.A. Huffman, A method for the construction of minimum-redundancy codes, in: Proceedings of the Institute of Radio Engineers, 1952.
[11] D. Johnson, S. Krishnan, J. Chhugani, S. Kumar, S. Venkatasubramanian, Compressing large boolean matrices using reordering techniques, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2004.
[12] N. Koudas, Space efficient bitmap indexing, in: Proceedings of ACM Conference on Information and Knowledge Management (CIKM), 2000.
[13] G. Navarro, E.S. de Moura, M. Neubert, N. Ziviani, R. Baeza-Yates, Adding compression to block addressing inverted indexes, Information Retrieval 3 (1) (2000) 49–77.
[14] P. O'Neil, Model 204 architecture and performance, in: International Workshop on High Performance Transactions Systems, Lecture Notes in Computer Science, vol. 359, Springer, Berlin, 1987.
[15] E. O'Neil, P. O'Neil, K. Wu, Bitmap index design choices and their performance implications, Research Report, Lawrence Berkeley National Laboratory, 2007.
[16] P. O'Neil, The set query benchmark, retrieved October 27, 2008, from ⟨http://www.cs.umb.edu/poneil/SetQBM.pdf⟩.
[17] P. O'Neil, D. Quass, Improved query performance with variant indexes, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1997.
[18] A. Pinar, T. Tao, H. Ferhatosmanoglu, Compressing bitmap indices by data reorganization, in: Proceedings of International Conference on Data Engineering (ICDE), 2005.
[19] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, J.M. Hellerstein, Efficient analysis of live and historical streaming data and its application to cybersecurity, Research Report, Lawrence Berkeley National Laboratory, 2006.
[20] D. Rinfret, P. O'Neil, E. O'Neil, Bit-sliced index arithmetic, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2001.
[21] D. Rotem, K. Stockinger, K. Wu, Optimizing candidate check costs for bitmap indices, in: Proceedings of ACM Conference on Information and Knowledge Management (CIKM), 2005.
[22] D. Rotem, K. Stockinger, K. Wu, Optimizing I/O costs of multi-dimensional queries using bitmap indices, in: Proceedings of International Conference on Database and Expert Systems Applications (DEXA), Lecture Notes in Computer Science, vol. 3588, Springer, Berlin, 2005.
[23] F. Scholer, H.E. Williams, J. Yiannis, J. Zobel, Compression of inverted indexes for fast query evaluation, in: Proceedings of International ACM SIGIR Conference, 2002.
[24] Scientific Data Management Research Group, FastBit: An efficient compressed bitmap index technology, retrieved November 10, 2006, from ⟨http://sdm.lbl.gov/fastbit/⟩.
[25] M. Stabno, R. Wrembel, RLH: Bitmap compression technique based on run-length and Huffman encoding, in: Proceedings of ACM International Workshop on Data Warehousing and OLAP (DOLAP), 2007.
[26] K. Stockinger, K. Wu, Bitmap indices for data warehouses, in: R. Wrembel, C. Koncilia (Eds.), Data Warehouses and OLAP: Concepts, Architectures and Solutions, Idea Group Inc., 2007, pp. 157–178 ISBN 1-59904-364-5.
[27] K. Stockinger, K. Wu, A. Shoshani, Evaluation strategies for bitmap indices with binning, in: Proceedings of International Conference on Database and Expert Systems Applications (DEXA), Lecture Notes in Computer Science, vol. 3180, Springer, Berlin, 2004.
[28] J.S. Vitter, Dynamic Huffman coding, ACM Transactions on Mathematical Software 15 (2) (1989) 158–167.
[29] A.N. Vo, A. Moffat, Compressed inverted files with reduced decoding overheads, in: Proceedings of International ACM SIGIR Conference, 1998.
[30] H. Williams, J. Zobel, Compressing integers for fast file access, Computer Journal 42 (3) (1999) 193–201.
[31] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann Publishers, Los Altos, CA, 1999.
[32] K. Wu, E.J. Otoo, A. Shoshani, Compressing bitmap indexes for faster search operations, in: Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM), 2002.
[33] K. Wu, E.J. Otoo, A. Shoshani, An efficient compression scheme for bitmap indices, Research Report, Lawrence Berkeley National Laboratory, 2004.
[34] K. Wu, E. J. Otoo, A. Shoshani, On the performance of bitmap indices for high cardinality attributes, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2004.
[35] K. Wu, E.J. Otoo, A. Shoshani, Optimizing bitmap indices with efficient compression, ACM Transactions on Database Systems (TODS) 31 (1) (2006) 1–38.
[36] K. Wu, P. Yu, Range-based bitmap indexing for high cardinality attributes with skew, in: International Computer Software and Applications Conference (COMPSAC), 1998.
[37] M. Wu, A. Buchmann, Encoded bitmap indexing for data warehouses, in: Proceedings of International Conference on Data Engineering (ICDE), 1998.
[38] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (3) (1977) 337–343.
[39] N. Ziviani, E.S. de Moura, G. Navarro, R. Bayeza-Yates, Compression: a key for next-generation text retrieval systems, IEEE Computer 33 (11) (2000) 37–44.