BAH: A Bitmap Index Compression Algorithm for Fast Data Retrieval

Chenxing Li[§†], Zhen Chen[*†‡], Wenxun Zheng[†‡], Yinjun Wu[†‡], Junwei Cao[†‡]
[*]Fundamental Industry Training Center (iCenter), Beijing, China
[†]Research Institute of Information Technology, Beijing, China
[‡]Tsinghua National Lab for Information Science and Technologies (TNList), Beijing, China
[§]Institute of Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
Corresponding e-mail: zhenchen@tsinghua.edu.cn

***Abstract*—Efficient retrieval of traffic archival data is a must-have technique to detect network attacks, such as APT(advanced persistent threat) attack. In order to take insight from Internet traffic, the bitmap index is increasingly used for efficiently querying over large datasets. However, a raw bitmap index leads to high space consumption and overhead on loading indexes. Various bitmap index compression algorithms are proposed to save storage while improving query efficiency. This paper proposes a new bitmap index compression algorithm called BAH (Byte Aligned Hybrid compression coding). An acceleration algorithm using SIMD is designed to increase the efficiency of AND operation over multiple compressed bitmaps. In all, BAH has a better compression ratio and faster intersection querying speed compared with several previous works such as WAH, PLWAH, COMPAX, Roaring etc. The theoretical analysis shows that the space required by BAH is no larger than 1.6 times the information entropy of the bitmap with density larger than 0.2%. In the experiments, BAH saves about 65% space and 60% space compared with WAH on two datasets. The experiments also demonstrate the query efficiency of BAH with the application in Internet Traffic and Web pages.**

***Index Terms*—Big Data, Traffic Archival, Web pages, Index, Bitmap Index**

I. Introduction

Efficient retrieval of traffic archival data is a must-have technique to detect network attacks, such as APT(advanced persistent threat) attack. However, the fast increasing amount of traffic data makes the efficient network analysis a challenge work. The VNI report[1] from Cisco indicated that the Internet traffic data had increased more than fivefold in the past 5 years and would increase about threefold in the next 3 years. It remains a problem that how to achieve the real-time data retrieval in massive archived Internet traffic data.

To achieve efficient query, the index for database is designed for avoiding a mass of loading tasks while responding queries. When a query was launched, the program would load corresponding index files instead of original data. Among the various index methods, the bitmap indexing obtains advantages from the efficiency of bitwise operation on the modern CPU, and is increasingly used for efficiently querying [2].

However, a raw bitmap index requires large storage, which results in a serious problem with I/O speed. Compared with an immense amount of time on the loading process, the advantage of the bitwise operation is negligible. Fortunately, various bitmap index compression algorithms have been devised for the relief of the I/O problem on the raw bitmap indexes, such as BBC[3], WAH[4], PLWAH[5], EWAH[6], PWAH[7], COMPAX[8], SECOMPAX[9], PLWAH+[10], CONCISE[11], SPLWAH[12], Roaring[13], MASC[14], CAMP[15], SBH[16]. A detailed survey has been presented in [17]. All of these algorithms have both acceptable I/O time and efficient bitwise operations.

Most bitmap compression algorithms are the variants of WAH. These algorithms use more complicated coding schemes to save the space consumption and I/O time usage. But the complicated encoding scheme increases the bitwise operation time. Some algorithms achieve the efficient query in another way, which adopt more simple coding schemes, such as Roaring[13] and CAMP[15]. This two algorithms have a marvelous speed of the bitwise operation. However, the simple coding scheme makes some concessions to space consumption. As the limited in-memory size can't restore the index for the huge amount of data, the optimization on I/O time must be also taken into consideration.

In this paper, a new bitmap index compression algorithm named BAH (Byte Aligned Hybrid compression coding) is proposed to improve the compression performance without loss in query efficiency. BAH uses the same basic ideas as WAH, which are Run Length Encoding and dealing with the raw bitmap in equal-length chunk. Unlike most variants of WAH, BAH doesn't use more complicated codebook. Instead, BAH encodes the raw bitmap using a simple rule and stores the result in byte other than word, with the help of the other three auxiliary arrays. Benefiting from the simple encoding scheme, the SIMD operations can be applied to the compressed bitmap to accelerate the AND operation.

This paper is organized as follows: Section 2 introduces the background of the bitmap index compression algorithms. Section 3 describes the details of the BAH algorithm. A theoretical analysis of the space consumption of WAH, PLWAH, COMPAX, Roaring and BAH is presented in Section 4. Section 5 presents the applications of BAH in Internet Traffic archival and Web page statistic. The experiments demonstrate the compression ratio and query efficiency of BAH compared with other algorithms. The conclusion and future work are

© 2016, Chenxing Li. Under license to IEEE.
DOI 10.1109/LCN.2016.120

IEEE computer society

given in Section 6.

II. Background for Bitmap Index

The basic notion of *bitmap index* is to keep several bitmaps, each of which corresponds to a possible value of an attribute. An example of a bitmap index for one attribute ranged from 1 to 4 is shown in Table I. Bitmap indexes allow fast bitwise AND/OR operation between columns, which makes the query process efficient.

However, a raw bitmap index causes a large space consumption. One attribute with n items and d possible values requires nd bits for the bitmap index, in which $n(d-1)$ bits are 0. To deal with such problem, a considerable amount of bitmap index compression algorithms, such as WAH, have been proposed.

WAH divides the raw bitmap into 31-bit chunks, and encodes continuous the fill chunks (all 31 bits are the same) into a single fill word (which is noted as 0F or 1F), while encoding all the other chunks into a literal word (which is noted as L).

It is mentioned in the previous works that WAH has a dissatisfactory performance when encoding some specific Words. For example, if the set bit appears every 31 bits in the raw bitmap, WAH will encode all the chunks as Literal, which leads to a large space consumption. In order to avoid such consumption, some other bitmap index compression algorithms are proposed to improve the compression ratio.

PLWAH is an encoding algorithm over the result of WAH, which tries to deal with the chunk with only one set bit to save more space than WAH. PLWAH encodes the fill word and its next literal word into one word if the chunk of the literal word contains only one set bit.

COMPAX improves WAH in a different way. COMPAX also tries to combine literal words and fill words for the result of WAH. However, COMPAX uses a codebook to deal with various situations. Instead of only considering the fill word followed with a special literal word, COMPAX finds the pattern LFL(literal-fill-literal) and FLF(fill-literal-fill) from the result of WAH and tries to combine them into one word.

Roaring considers this method in a different way. Roaring divides the raw bitmap into chunks with 65536 bits. The chunk with low density (no more than 4096 set bits) is encoded in 16-bit integer list. The chunk with high density (more than 4096 set bits) are stored in raw bitmap directly.

However, most algorithms only focus on the optimization on some specific Literal Words in WAH, with the space wasted on the fill words ignored. If the average length of continuous fill chunks was small, most of the counter in the fill words would require a few bits instead of an entire word. For example, if the set bit appears every 124 bits in the raw bitmap, only 2 bits of 31 bits-counter part are used, while 29 bits are wasted.

In order to deal with such situations, BAH uses bytes instead of words to record the run length. BAH also adopts the idea that the word patterns appealing frequently needs special treatment. The coding scheme of BAH is given in the following section.

TABLE I: An Example of a Bitmap Index

RowID	Value	Bitmap index			
		=1	=2	=3	=4
1	4	0	0	0	1
2	3	0	0	1	0
3	2	0	1	0	0
4	3	0	0	1	0
5	4	0	0	0	1
6	1	1	0	0	0
...	...	⋮	⋮	⋮	⋮

III. Coding Algorithm

A. Notations

For clarity, all the nouns, symbols and notations used in the following part of this paper are defined in this sub-section.

1) *Word* means a binary string with 32 bits.
2) All the binary strings are subscripted with 2. Binary string and an integer are used interchangeably.
3) The set S_n is defined as $\{x \in \mathbb{Z} \mid 0 \leq x \leq n-1\}$.
4) $w[i]$ $(i \in S_{32})$ represents the i^{th} bit of the Word w. $w[0]$ is the least significant bit and $w[31]$ is the most significant bit.
5) $w[i:j]$ represents the substring $w[j-1]\cdots w[i+1]w[i]$.
5) The *Workload* of a Word means the number of set bit in a Word. For example, the overload of Word $00000001000000000001000000000000_2$ is 2.
6) The *Loading Part* of w is the substring removing leading zero bits and trailing zero bits of w. The *Loading Length* of a Word is the length of its Loading Part. For the Word $00000001001000000000000000000000_2$, the Loading Part is 1001_2 and the Loading Length is 4.

B. Encodable Pattern and Pattern Function

BAH divides the raw bitmap index into chunks with 32 bits. In other words, the algorithm regards the input as a Word array. In a real implementation, some Words in the input array may appear more frequently than the others. For example, in a sparse bitmap, the Word with a low Workload appears more frequently than the Word with a high Workload. Using less space to encode these Word patterns can save the space consumption.

Two functions $f : S_{64} \to \{0,1\}^{32}$, $g : S_{64} \times S_{256} \to \{0,1\}^{32}$ are defined in order to encode such Word. f makes a mapping relation between 64 different Words and integers in S_{64}. So it is possible that all the Words in the image set of f are encoded in one byte. (More details are shown in subsection C). Each Word in the image set of f is called *One byte Encodable Pattern (OEP)*. Each Word in the image set of g can be encoded using two bytes in coding method, which is called *Two bytes Encodable Pattern (TEP)*.

BAH contains 64 OEPs and 16384 TEPs. Both OEPs and TEPs are called *Encodable Patterns*. Function f, g are called *Pattern Functions*.

In the encoding process, it is necessary to recognize the OEPs and TEPs from in the input Words. For each Encodable Pattern w, the algorithm needs to find out the pre-image of w in the pattern function. A general way to deal with this problem is Hash Set, which stores the pairs (*key,value*)=$(w, f^{-1}(w))$ or $(w, g^{-1}(w))$ for all the Encodable Patterns. Another way is designing a customized program. If the choice of Encodable Patterns is fixed before compiling the program, we can design a special procedure to judge and compute the pre-image of these Encodable Patterns. The experiments in this paper use the second way.

In the decoding process, the algorithm constructs two lookup tables for Pattern Functions f, g. The lookup table for f requires 256 bytes, which can be stored in the L1 cache. The lookup table for g requires 64 KB, which can be stored in the L2 cache. So the computation of f, g can be applied efficiently.

The choice of Encodable Patterns depends on the character of the original data. In our experiment, the Words with Loading Part 11_2 and the Words whose Workload is 1 or 32 are chosen as the OEPs. We choose the Words having one of the following characters to be the TEPs.

- The Word whose Workload is 3, 30 or 31.
- The Word whose Loading Part are all set bits.
- The Word whose Loading Length is less than 10.

C. Coding Method

It is mentioned in the previous subsection that the raw bitmap is regarded as a Word array. The coding method classifies all the Words into 3 types: *Zero, Encodable* and *Literal*.

- Zero(Z): All the 32 bits are 0.
- Encodable(E): The Words belong to Encodable Patterns.
- Literal(L): All the other Words.

BAH uses a *main array* in byte to encode the original sequence, with the help of 3 auxiliary arrays: *data array*(in Word), *index array* (in byte) and *counter array* (in Word). Each byte in the main array may contain some corresponding elements in the other 3 arrays. In the encoding process, each time the algorithm appending one byte to the main array, corresponding elements are appended to the other 3 arrays concurrently. In the decoding process, each time a byte is retrieved from the main array, the algorithm accesses its corresponding elements in the other 3 arrays.

The design of elements in Main array is shown in Table II. The two most significant bits of each byte in main array denote the *type* and the six least significant bits are regarded as an integer $n \in S_{64}$.

D. Encoding process

In the Encoding process, the input array is divided into several segments. Continuous Words with the same type construct one segment. After that, all the Literal segments are divided into sub-segments with no more than 63 Words. All the Zero segments with less than 253 words are divided into sub-segments with no more than 63 Words. All the Encodable segments are divided into sub-segments with length 1.

TABLE II: The design for bytes in main array

Type	n	Corresponding elements and **Meaning**
00_2	$n = 0$	One integer x in the counter array x **continuous Zero Words**
	$1 \le n \le 63$	n **continuous Zero Words**
01_2	$n = 0$	(Undefined)
	$1 \le n \le 63$	n continuous Word in the data array n **continuous Literal Words**
10_2	$0 \le n \le 63$	**One Encodable Word:** $f(n)$
11_2	$0 \le n \le 63$	One byte m in the index array **One Encodable Word:** $g(n, m)$

TABLE III: The encoding scheme for each segment type

Type	Length l	Main array		Corresponding Elements
		type	n	
Zero	$1 \le l \le 63$	00_2	l	None
	$l > 252$	00_2	0	Append integer l to counter array
Literal	$1 \le l \le 63$	01_2	l	Append l Literal Words to data array
OEP $w = f(a)$	$l = 1$	10_2	a	None
TEP $w = g(a, b)$	$l = 1$	11_2	a	Append byte b to index array

The following example shows the way to divide the Word array into 4 segments and 10 sub-segments.

Uncompressed bitmap, regarded as a Word array:

$$\underbrace{\texttt{ZZZZ}\cdots\texttt{ZZZ}}_{70\times\texttt{Z}}\texttt{EE}\underbrace{\texttt{LLLLL}\cdots\texttt{LLL}}_{270\times\texttt{L}}\underbrace{\texttt{ZZZZZ}\cdots\texttt{ZZZ}}_{270\times\texttt{Z}}$$

Word array is divided into 4 segments:

$$\underbrace{\texttt{ZZZZ}\cdots\texttt{ZZZ}}_{70\times\texttt{Z}}\ \texttt{EE}\ \underbrace{\texttt{LLLLL}\cdots\texttt{LLL}}_{270\times\texttt{L}}\ \underbrace{\texttt{ZZZZZ}\cdots\texttt{ZZZ}}_{270\times\texttt{Z}}$$

4 segments are divided into 10 sub-segments:

$$\underbrace{\texttt{Z}\cdots\texttt{Z}}_{63\times\texttt{Z}}\ \underbrace{\texttt{Z}\cdots\texttt{Z}}_{7\times\texttt{Z}}\ \texttt{E}\ \texttt{E}\ \underbrace{\texttt{L}\cdots\texttt{L}}_{63\times\texttt{L}}\ \underbrace{\texttt{L}\cdots\texttt{L}}_{63\times\texttt{L}}\ \underbrace{\texttt{L}\cdots\texttt{L}}_{63\times\texttt{L}}\ \underbrace{\texttt{L}\cdots\texttt{L}}_{63\times\texttt{L}}\ \underbrace{\texttt{L}\cdots\texttt{L}}_{18\times\texttt{L}}\ \underbrace{\texttt{Z}\cdots\texttt{Z}}_{270\times\texttt{Z}}$$

After dividing step, each segment or sub-segment is encoded using the rules showed in Table III.

In the implementation, two dividing steps and one encoding step are executed in one scan. The Words are fetched from the original bitmap and stored it into a buffer. Once the next Word has the different type from the Words in buffer, or the length of the buffer exceeds the threshold (63 for Literal and 1 for Encodable Word), the buffer is regarded as a sub-segment and encoded according to Table III.

E. AND operation over multiple bitmaps

AND operation can be applied over multiple compressed bitmaps $B_0, \cdots, B_{n-1}$ concurrently. The corresponding raw

Input: The compressed bitmaps $B_0, \cdots, B_{n-1}$.
Output: The result of AND operation formatted in integer array.

1: Construct 4 pointers $p_{main,i}, p_{data,i}, p_{index,i}, p_{counter,i}$ and two interger k_i, m_i for each B_i.
2: Initialize all the pointers to the start of corresponding array.
3: Set $k_i \leftarrow -1, m_i \leftarrow 0$ for all i
4: $k_{max} \leftarrow -1, i \leftarrow 0, leader \leftarrow -1, w \leftarrow$ `0xffffffff`
5: **while** all the pointers are within the boundary **do**
6: **if** $leader = i$ **then**
7: Output $32(k_{max}+1)+j$ for all j with $w[j]=1$.
8: $k_{max} \leftarrow k_{max}+1, leader \leftarrow -1$
9: $w \leftarrow$ `0xffffffff`
10: **end if**
11: **if** $leader = -1$ **then** $leader \leftarrow i$
12: $w \leftarrow$*move*(B_i, k_{max}, w)
13: **if** $w = 0$ **then**
14: $k_{max} \leftarrow k_i, w \leftarrow$ `0xffffffff`$, leader \leftarrow -1$
15: **end if**
16: $i \leftarrow (i+1 \bmod n)$
17: **end while**

Algorithm 1: AND operation over n bitmaps

bitmaps are denoted by $C_0, \cdots, C_{n-1}$. During AND operation, the pointers for 4 arrays are maintained for each bitmap. Each bitmap also maintains an integer k to denote the current position in the original bitmap and an integer m to record the number of literal that has not been read in current Literal sub-segment.

Algorithm 1 shows AND operation briefly, outputs the result formatted in integer list, each element of which is an index of set bit in the answer bitmap. Algorithm 2 is a sub-function called by Algorithm 1. Let $w_{i,k_{max}+1}$ denote the word of bitmap C_i at position $k_{max}+1$. The return value of Algorithm 2 is $w_{i,k_{max}+1}$ AND w.

In these two algorithms, the pointers always point to the next elements to be read and move to the next element immediately once the content pointed has been read.

Algorithm 1 sequentially computes the result of each position. k_{max} denotes the last position which has been dealt with. The result of AND operation at position $k_{max}+1$ will be recorded in w.

Each time updating k_{max}, to get the result at position $k_{max}+1$, the Algorithm 1initialize $w \leftarrow$ `0xffffffff` firstly. Then each bitmap B_i reads its original Word at position $k_{max}+1$ and merges the answer to w by applying $w \leftarrow w_i$ AND w in turn (Line 12). Once all the bitmaps merge their answers, or w equals to 0 during this process, the algorithm outputs the result at position $k_{max}+1$ and updates k_{max}. The variable $leader$ is used to determine whether all the bitmaps have merged their answers.

In Algorithm 2, k_i represents the position in C_i which is

Input: B_i, corresponding varibles of B_i, k_{max} and w
Output: Updated w
Note: This algorithm may modify some variables used in Algorithm 1, such as k_i.

1: $c_i \leftarrow m_i$
2: $b_f \leftarrow 01_2$
3: **while** $k_{max} > k_i + c_i$ **do**
4: Move pointer $p_{data,i}$ forward m_i words.
5: $m_i \leftarrow 0, k_i \leftarrow k_i + c_i$
6: Read the byte b pointed by $p_{main,i}$.
7: $b_f \leftarrow b[6:8]$
8: $b_l \leftarrow b[0:6]$
9: **if** $b = 0$ **then**
10: Read the integer a_i pointed by $p_{counter,i}$
11: $c_i \leftarrow a_i$
12: **else**
13: **if** $b_f = 00_2$ **then** $c_i \leftarrow b_l$
14: **if** $b_f = 01_2$ **then** $c_i \leftarrow b_l, m_i \leftarrow b_l$
15: **if** $b_f = 10_2$ **then** $c_i \leftarrow 1$
16: **if** $b_f = 11_2$ **then** $c_i \leftarrow 1$, Read the byte $index$ pointed by $p_{index,i}$
17: **end if**
18: **end while**
19: **if** b_f is 00_2 **then**
20: $k_i \leftarrow k_i + c_i$
21: **return** `0`.
22: **end if**
23: **if** b_f is 01_2 **then**
24: $r_i \leftarrow k_{max} - k_i$
25: Move pointer $p_{data,i}$ forward r_i words.
26: $m_i \leftarrow m_i - r_i$.
27: Read the byte w_i pointed by $p_{data,i}$
28: $m_i \leftarrow m_i - 1$.
29: **end if**
30: **if** b_f is 10_2 or 11_2 **then**
31: Calculate w_i using pattern function and $index$
32: **end if**
33: $k_i \leftarrow k_{max} + 1$
34: **return** w AND w_i

Algorithm 2: move(b_i, k_{max}, w)

the last Word been decoded. If the last element read out from main array is of type 10_2, m_i denotes the number of Literals that haven't been read in current Literal sub-segment. c_i and r_i are two temporary variables. The loop between Line 3 to 18 is called *Main Loop*. The Word at position $k_{max}+1$ in the original bitmap is called *Objective Word*.

The Main Loop reads byte b from main array repeatedly and maintains all corresponding variables of B_i, until the main array element encoding the Objective Word is found. If the Objective Word is in a Zero sub-Segment, k_i will be moved to the end position of this sub-segment.

Input: B_i, corresponding varibles of B_i and k_{max}
Output: None.

1: **if** $k_{max} - k_i < \min\{64, c_i\}$ **then** Exit this procedure
2: Move pointer $p_{data,i}$ forward m_i words.
3: $m_i \leftarrow 0, k_i \leftarrow k_i + c_i$
4: $c_i \leftarrow 0$
5: **while** $k_{max} > k_i + c_i$ **do**
6: $k_i \leftarrow k_i + c_i$
7: Record all the corresponding variables of B_i.
8: Read the next 16 bytes $b_0 \cdots b_{15}$ pointed by $p_{main,i}$.
9: Set $s_1 \leftarrow \sum_{j \in S_{16}, b_j = 0} 1$.
10: Set $s_2 \leftarrow \sum_{j \in S_{16}, b_j[7]=1} 1$.
11: Set $s_3 \leftarrow \sum_{j \in S_{16}, b_j[7]=0} b_j[0:6]$.
12: Set $s_4 \leftarrow \sum_{j \in S_{16}, b_j[6:8]=01_2} b_j[0:6]$.
13: Set $s_5 \leftarrow \sum_{j \in S_{16}, b_j[6:8]=11_2} 1$.
14: Read the next s_1 integers pointed by $p_{counter,i}$ and sum them into a_i
15: $c_i \leftarrow a_i + s_3 + s_2$.
16: **if** $k_{max} < k_i + c_i$ **then**
17: Rewind all the corresponding variable to the states recorded in Line 7
18: Exit this procedure
19: **end if**
20: Move pointer $p_{data,i}$ forward s_4 elements.
21: Move pointer $p_{index,i}$ forward s_5 elements.
22: **end while**

Algorithm 3: SIMD acceleration procedure

F. SIMD acceleration

A SIMD acceleration procedure could be inserted between Line 2 and Line 3 in Algorithm 2. This can accelerate the process finding the main array element encoding the Objective Word. The Algorithm 3 shows the SIMD acceleration procedure. Each round of the loop in Algorithm 3 is equivalent to 16 rounds of the loop in Algorithm 2.

The calculation of $s_1 \sim s_5$ in algorithm 3 can be executed by SIMD operations in a few CPU cycles. The C style functions for Intel intrinsic instructions used is list as follows, more details about these functions can be found on the web page [18].

1. `_mm_cmpgt_epi8`
2. `_mm_cmplt_epi8`
3. `_mm_cmpeq_epi8`
4. `_mm_movemask_epi8`
5. `_mm_and_si128`
6. `_mm_sad_epu8`
7. `_mm_cvtsi128_si32`
8. `_popcnt32`
9. `_mm_srli_si128`

The Algorithm 4 shows the way to calculate $s_1 \sim s_5$ using above functions. We use some simple symbols to represent these 9 functions: `cmpgt`, `cmplt`, `cmpeq`, `mvm`, `and`, `sad`, `cvtsi`, `popcnt`, `srli`. The brace $\{x_j\}$ represents the 16-element byte array $(x_0, x_1, \cdots, x_{15})$.

Input: 16 bytes $b_0 \cdots b_{15}$ read from main array
Output: $s_1 \sim s_5$.

1: $\forall j \in S_{16}$, Set $x_j \leftarrow 63, y_j \leftarrow -63, z_j \leftarrow 0$
2: $s_1 \leftarrow \mathit{popcnt}\left(\mathit{mvm}\left(\mathit{cmpeq}\left(\{b_j\}, \{z_j\}\right)\right)\right)$
3: $s_2 \leftarrow \mathit{popcnt}\left(\mathit{mvm}\left(\mathit{cmplt}\left(\{b_j\}, \{z_j\}\right)\right)\right)$
4: $s_5 \leftarrow \mathit{popcnt}\left(\mathit{mvm}\left(\mathit{cmpgt}\left(\mathit{cmplt}\left(\{b_j\}, \{z_j\}\right), \{y_j\}\right)\right)\right)$
5: $\{u_j\} \leftarrow \mathit{sad}\left(\mathit{and}\left(\mathit{cmpgt}\left(\{b_j\}, \{z_j\}\right), \{x_j\}\right)\right)$
6: $s_3 \leftarrow \mathit{cvtsi}\left(\{u_j\}\right) + \mathit{cvtsi}\left(\mathit{srli}\left(\{u_j\}, 8\right)\right)$
7: $\{v_j\} \leftarrow \mathit{sad}\left(\mathit{and}\left(\mathit{cmpgt}\left(\{b_j\}, \{x_j\}\right), \{x_j\}\right)\right)$
8: $s_4 \leftarrow \mathit{cvtsi}\left(\{v_j\}\right) + \mathit{cvtsi}\left(\mathit{srli}\left(\{v_j\}, 8\right)\right)$

Algorithm 4: Details for SIMD implementation

G. Simple Version

The simple version of BAH denoted by BAH_simp is BAH without TEP. In BAH_simp, the pattern function $g : S_{64} \times S_{256} \rightarrow \{0,1\}^{32}$ is replaced with $g : S_{64} \rightarrow \{0,1\}^{32}$. In other words, the type 11_2 in BAH_simp is designed similar with the type 10_2 in BAH. The removal of TEP and index array makes the AND operation on BAH_simp simpler.

The BAH_simp contains 128 OEPs and no TEP. In our experiments, we choose the Words satisfying one of the following conditions to be the OEPs of BAH_simp:

- The Word whose Workload is 1.
- The Word whose Workload is 2 and Loading Length is no more than 4.

IV. Theoretical Analysis

In this section, we compare the theoretical space consumption of BAH with other 4 algorithms: WAH, PLWAH, COMPAX and Roaring. A former theoretical analysis with bitmap index is presented in [19]. Suppose the raw bitmap is a sequence of independent and identically distribution random bits. The bit density (the proportion of set bits) of the raw bitmap is p. Here we suppose $p \leq 0.5$. The analysis ignores the coding scheme for 1-fill chunks because the proportion of 1-fill chunks is negligible.

In the following subsections, we will calculate the expectation of space consumption for each algorithm. $E[\mathit{WAH}]$, $E[\mathit{PLWAH}]$, $E[\mathit{COMPAX}]$, $E[\mathit{Roaring}]$, $E[\mathit{BAH}]$ denotes the expectation of space consumption under n bits input. (n is sufficiently large).

A. WAH

Algorithm WAH divides the raw bitmap into a sequence of 31-bit chunks and encodes continuous 0-fill chunks (the chunks with no set bit) between two non-0-fill chunks in a Word called Fill (F), which contains one bit flag and 31-bit counter part. Each non-0-fill chunk is encoded in one Word called Literal (L).

Let f denote a 0-fill Chunk, l denote a non-0-fill Chunk, f^k denote k continuous 0-fill Chunk. Set $x = (1-p)^{31}$. For each

pattern $s \in \{f, l\}^*$, $E[s]$ denotes the expectation number of occurences of s in the chunk sequence. In WAH, we can get:

$$E[f] = x \cdot \frac{n}{31}$$
$$E[\,l\,] = (1-x) \cdot \frac{n}{31}$$
$$E[\,lf^k l] = x^k(1-x)^2 \cdot \frac{n}{31}$$

In WAH algorithm, each l is encoded in one Literal Word, and each f^k bounded by l is encoded in one Fill Word. The expected length of WAH algorithm is

$$\begin{aligned} E[\textit{WAH}] &= 32 \cdot E[\,l\,] + \sum_{k=1}^{2^{31}-1} 32 \cdot E[\,lf^k l] \\ &= \frac{32n}{31} \cdot \left((1-x) + (1-x)(x - x^{2^{31}})\right) \\ &\approx \frac{32n}{31} \cdot (1-x^2) \end{aligned}$$

Since $0 < x < 1$, we suppose $x^{2^{31}} \approx 0$ here.

B. PLWAH

The structure of Fill Word in PLWAH consists of 2-bit flag, 5-bit "position" part and 25-bit counter part.

In PLWAH algorithm, a 31-bit chunk with Workload 1 is called "nearly identical". If a sequence of continuous 0-fill chunks is followed by a nearly identical chunk (a chunk with only one set bit), PLWAH would encode them together in a Fill Word. The position part will record the position of the set bit in the nearly identical chunk and the counter part will record the length of the fill chunks.

Let $y = 31p(1-p)^{30}$. Let f denote a 0-fill chunk and l a non-0-fill chunk. The non-0-fill chunk contains two sub-types, let l_1 denote the nearly identical chunk and l_2 denote the other non-fill chunks. In PLWAH, we have

$$E[\,l\,] = (1-x) \cdot \frac{n}{31}$$
$$E[\,l_1\,] = y \cdot \frac{n}{31}$$
$$E[\,l_2\,] = (1-x-y) \cdot \frac{n}{31}$$
$$E[\,ll_1\,] = (1-x)\,y \cdot \frac{n}{31}$$
$$E[\,lf^k l_1\,] = x^k(1-x)\,y \cdot \frac{n}{31}$$
$$E[\,lf^k l_2\,] = x^k(1-x)(1-x-y) \cdot \frac{n}{31}$$

PLWAH algorithm encodes l_2 and l_1 following l in Literal Word and encodes the other situations in Fill Word.

$$\begin{aligned} &E[\textit{PLWAH}] \\ &= 32 \cdot \left(E[l_2] + E[ll_1] + \sum_{k=1}^{2^{25}-1} \left(E[lf^k l_1] + E[lf^k l_2]\right) \right) \\ &= \frac{32n}{31} \cdot \left((1-x-xy) + (1-x)(x - x^{2^{25}})\right) \\ &\approx \frac{32n}{31} \cdot (1 - x^2 - xy) \end{aligned}$$

Since $0 < x < 1$, we also suppose $x^{2^{25}} \approx 0$ here.

C. COMPAX

The expectation of space consumption with COMPAX can be computed using Markov Chain. Because of space restriction, we only show the result for COMPAX here. We define 3 variables:

$$\begin{aligned} x &= (1-p)^{31} \\ y &= (1-p)^{23} + 3(1-p)^{24} - 4(1-p)^{31} \\ z &= x - x^{64}. \end{aligned}$$

The space reduction of COMPAX compared with WAH is:

$$\begin{aligned} &E[\textit{COMPAX}] - E[\textit{WAH}] = \\ &\frac{64nyz\left((1-x^2)(1+x-z)^2 + (1-x^2)^2 z + (1+x)^2 y^2 z\right)}{31(1+2x)\left(1+x^4+yz(1+yz) - x^2(2+yz)\right)} \end{aligned}$$

D. Roaring

Roaring divides the raw bitmap into a sequence of 2^{16}-bit chunks. For each chunk with k set bits:

- $k = 0$, Nothing is recorded.
- $1 \le k \le 4096$, Roaring records the index of each set bit using 16-bit integers, which need $16k$ bits. 32 extra bits are required to record some other information.
- $k > 4096$, Roaring records 2^{16} bits directly. 32 extra bits are required to record some other information.

The expectation of space consumption with Roaring is:

$$\begin{aligned} E[\textit{Roaring}] = &\frac{n}{2^{16}} \cdot \sum_{i=1}^{4096} \binom{2^{16}}{i} (16i + 32) p^i (1-p)^{32-i} \\ &+ \frac{n}{2^{16}} \cdot \sum_{i=4097}^{2^{16}} \binom{2^{16}}{i} (2^{16} + 32) p^i (1-p)^{32-i} \end{aligned}$$

E. BAH

The performance of BAH depends on the choice of the Encodable Words. Here we choose 32 Words with Workload 1 as the OEP, and 5456 Words with Workload 2 or 3 as the TEP.

Let z denote the Zero Word, l the Literal Word, c the Encodable Word, c_1 the OEP and c_2 the TEP. Let $\bar{x}$ denotes all the Words excluding x. ($x \in \{z, l, c\}$). Let $p_z = (1-p)^{32}$, $p_{c_1} = 32p(1-p)^{31}$, $p_{c_2} = 496p^2(1-p)^{30} + 4960p^3(1-p)^{29}$, $p_l = 1 - p_z - p_{c_1} - p_{c_2}$.

In BAH, we have $E[\,z\,] = p_z n/32$, $E[\,c_1\,] = p_{c_1} n/32$, $E[\,c_2\,] = p_{c_2} n/32$, $E[\,l\,] = p_l n/32$, and

$$E[\,\bar{l} l^k \bar{l}\,] = (1-p_l)^2 p_l^k \cdot \frac{n}{32}$$
$$E[\,\bar{z} z^k \bar{z}\,] = (1-p_z)^2 p_z^k \cdot \frac{n}{32}$$

The expectation of space consumption on each part is shown in Table IV. Summing all these results, we can get

$$\begin{aligned} &E[\textit{BAH}] \\ &= \frac{n}{4} \cdot \big(p_z(1-p_z)(1 + p_z^{63} + p_z^{126} + p_z^{189} + p_z^{252}) \\ &\quad + p_{c_1} + 2p_{c_2} + 4p_l + (1-p_l)^2 p_l / (1-p_l^{63})^2\big) \end{aligned}$$

TABLE IV: Space consumption of different part

Array	Type	Space consumption (bits)
Main	00_2	$\sum_{t=1}^{4}\sum_{k=63t-62}^{63t} 8tE[\bar{z}z^k\bar{z}] + \sum_{k=253}^{\infty} 32E[\bar{z}z^k\bar{z}]$
	01_2	$\sum_{t=1}^{\infty}\sum_{k=63t-62}^{63t} 8tE[\bar{l}l^k\bar{l}]$
	10_2	$8p_{c_1}\cdot\frac{n}{32}$
	11_2	$8p_{c_2}\cdot\frac{n}{32}$
Index		$8p_{c_2}\cdot\frac{n}{32}$
Literal		$32p_l\cdot\frac{n}{32}$
Counter		$\sum_{k=253}^{\infty} 32E[\bar{z}z^k\bar{z}]$

F. Comparison

Fig. 5a shows the average number of compressed bytes per set bit required by each algorithm. The density p is chosen from 0.2% to 50%. We can observe that BAH has the best performance when $p \in [0.002, 0.5]$.

The information entropy of each bit in a random bitmap with density p is $H(p) = -p\log_2 p - (1-p)\log_2(1-p)$. For a bitmap with length n, the theoretical best space consumption is information entropy $nH(p)$ bits. Fig. 5b and 5c shows the ratio of space consumption between each algorithm and information entropy. It can be shown that the lower bound of ratio of BAH is 1.6, which is about one half of the next best algorithm PLWAH.

V. Experiment

In the experiment, the space consumption and the query efficiency of each algorithm will be evaluated using the real world datasets.

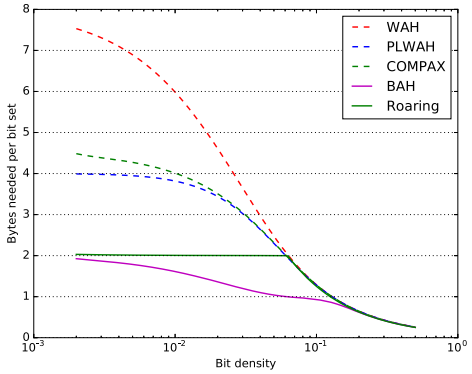
A. Environment

The algorithm is implemented by C++. All the experiments were conducted on a 64-bit machine with operating system Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-24-generic x86_64) and compiler gcc in version 4.8.2. The experimental machine has a Intel i7-3770 CPU @ 3.40GHz, 20G RAM and 2TB magnetic disk.

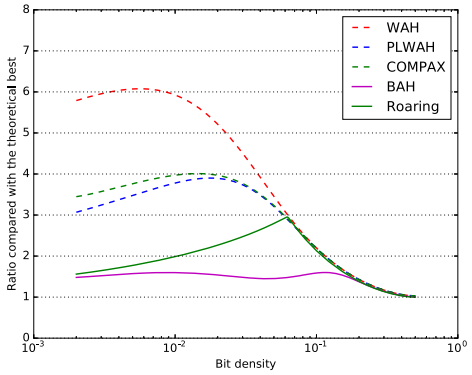
B. Datasets and Experimental Design

1) CAIDA: The CAIDA dataset [20] is the real network trace data. About 150 million IPv4 records from 6 files in CAIDA anonymized internet traces are used in the experiment. The task in the experiment on dataset CAIDA is retrieving the IP addresses.

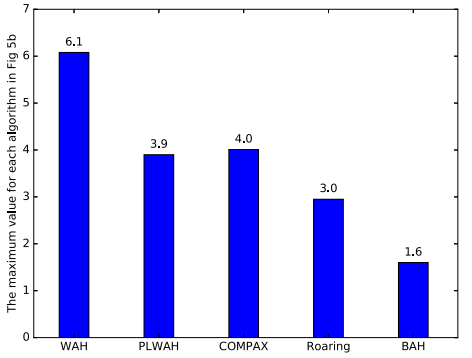
2) Clueweb09: We downloaded ClueWeb09 Gap Data set[21], which represents posting lists extracted from the category B html files of the ClueWeb09 collection. The category B contains 50 million English web pages. ClueWeb09 Gap Data set contains 1 million posting lists of the most frequent words. Decompressed ClueWeb09 Gap Data data is formatted in ordered integer lists.



(a) Space Consumption per set bit in the raw bitmap



(b) The ratio between the algorithms and information entropy



(c) The upper bound of the ratio in Fig. 5b

Fig. 5: Comparison of theoretical compression ratio.

C. Experimental Settings

In the following experiments, the BAH and BAH_simp are compared with WAH, PLWAH, COMPAX and Roaring.

PLWAH and COMPAX are chosen as the representative of the variants of WAH. Roaring is chosen as the representative of algorithms using simple coding scheme.

The performance on the space consumption and query efficiency are compared among these 6 algorithms.

1) Experimental Setting for CAIDA: For CAIDA, each byte of IP addresses was treated as an attribute. There were 8 attributes in total for source addresses and destination addresses, each of which has 256 possible values. The compressed bitmaps for these 8 attributes are prepared before the query experiments.

In order to simulate real queries, 500 IP addresses are chosen randomly as query tasks from all the appearing IP addresses for each file. Each IP address contains a tag picked randomly from "source" and "destination". For each IP address, the algorithms are required to output the positions of all the records with the same source address or destination address.

2) Experimental Setting for Clueweb09: In Clueweb09, 1 million ordered integer lists were converted to 1 million bitmaps. The compressed bitmaps for all these lists are prepared before the query experiment.

In order to simulate the query performance on Clueweb09, 1000 random compressed bitmap are picked from 100000 bitmaps. These bitmaps are paired in 500 pairs, each of which represents an AND operation task on two bitmaps.

D. Result

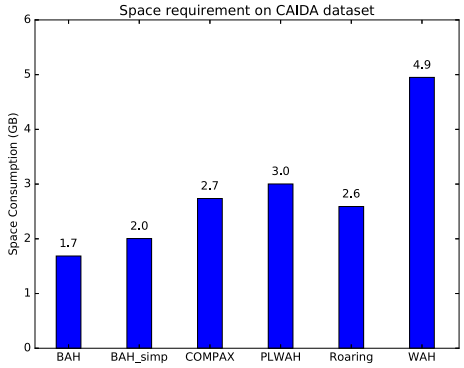
1) Space consumption: Fig. 6 shows the space consumption of different algorithms under real datasets. On CAIDA dataset (Fig. 6a), BAH saves about 65% compared with WAH, about 40% with COMPAX and about 35% with Roaring in space consumption. On Clueweb09 dataset (Fig. 6b), BAH saves about 60% compared with WAH, about 46% with Roaring, about 39% with PLWAH and 37% space of COMPAX.

In all these datasets, BAH_simp needs about 20% more space than BAH, which is still better than the other algorithms.

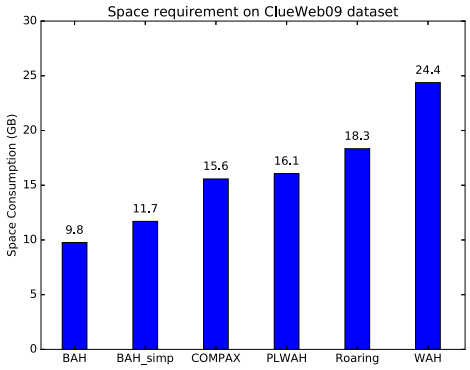
2) Query efficiency: The time for loading index files and the time for bitwise operations are both taken into account.

Fig. 7a shows the total time per query on CAIDA. Benefiting from the compression ratio, BAH saves much time on loading process and brings about 1.08× and 1.17× speed-up compared with Roaring and COMPAX. Fig. 7b shows the total time per query on Clueweb09, which is similar with the performance on CAIDA.

The Table V shows the accurate query time. It can be shown that the query efficiency largely depends on the loading time. The compression ratio with good bitwise operation time guarantees fast query on BAH and BAH_simp.



(a) The space consumption of compressed bitmap on CAIDA dataset using experimental algorithms



(b) The space consumption of compressed bitmap on Clueweb09 dataset using experimental algorithms

Fig. 6: Space consumption on real datasets

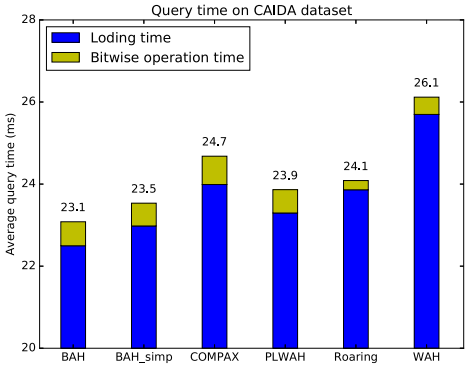
TABLE V: The accurate query time(ms)

Total time (Loading Time + Bitwise operation)

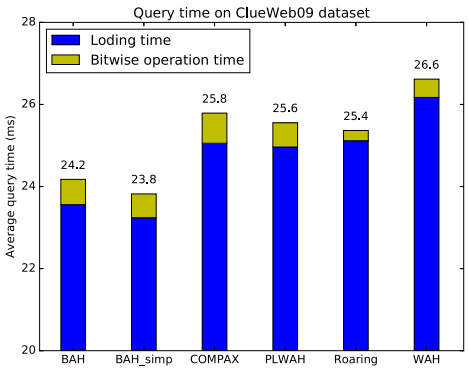
Algorithm	CAIDA	Clueweb09
WAH	26.12 (25.70 + 0.42)	26.42 (26.17 + 0.45)
PLWAH	23.87 (23.30 + 0.57)	25.55 (24.96 + 0.59)
COMPAX	24.68 (23.99 + 0.69)	25.79 (25.05 + 0.74)
Roaring	24.09 (23.86 + 0.22)	25.37 (25.12 + 0.25)
BAH	23.08 (22.50 + 0.58)	24.17 (23.56 + 0.62)
BAH_simp	23.53 (22.98 + 0.56)	23.82 (23.24 + 0.58)

VI. Conclusion and Future work

In this paper, we present BAH, a bitmap compression algorithm that achieves better performance in space consumption and query efficiency. BAH uses arrays in bytes instead of Words to save space usage, which decreases the loading time of indexes. The design of Encodable Word is helpful for the compression. Simple coding scheme and SIMD acceleration algorithm make BAH conduct efficient bitwise operation. The theoretical analysis shows that BAH requires no more 1.6 times space of entropy information under a random bitmap



(a) The query time on CAIDA dataset



(b) The query time on Clueweb09 dataset

Fig. 7: Query time on real dataset

with density larger than 0.2%. The experiment based on the dataset CAIDA and Clueweb09 demonstrates the performance on network traffic data retrieval and the Web pages. BAH has the best performance in both application scenarios.

The future work focus on the Encodable Word. There could be more efficient algorithms for encoding Encodable Word. The approach to choose Encodable Word from specific original data has not been considered yet. BAH tries to optimize the space consumption of bitmaps with density larger than 0.2%, but does not consider more sparse bitmap. An alternative choice may be added to the design of the algorithm to optimize the compression ratio on these situation.

ACKNOWLEDGMENT

This work was supported in part by National Natural Science Foundation of China (grants No. 61472200 and No. 61233016), Ministry of Science and Technology of China under National 973 Basic Research Program (grant No. 2013CB228206), State Grid R&D project "Research on the Architecture of Information Communication System for Internet of Energy" (grant No. SGRIXTKJ[2015]253), National Training program of Innovation and Entrepreneurship for Undergraduates (No.201610003B010 and No.201610003031, 201610003032, 201610003033). This work is also supported by Sumavision Technologies.

REFERENCES

[1] I. Cisco, "Cisco visual networking index: Forecast and methodology, 2014–2019," *CISCO White paper*, 2015.

[2] I. Spiegler and R. Maayan, "Storage and retrieval considerations of binary data bases," *Information processing & management*, vol. 21, no. 3, pp. 233–254, 1985.

[3] G. Antoshenkov, "Byte-aligned bitmap compression," in *Data Compression Conference, 1995. DCC'95. Proceedings.* IEEE, 1995, p. 476.

[4] K. Wu, E. J. Otoo, and A. Shoshani, "Compressing bitmap indexes for faster search operations," in *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*. IEEE, 2002, pp. 99–108.

[5] F. Deliège and 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*. ACM, 2010, pp. 228–239.

[6] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," *Data & Knowledge Engineering*, vol. 69, no. 1, pp. 3–28, 2010.

[7] S. J. van Schaik and 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*. ACM, 2011, pp. 913–924.

[8] F. Fusco, M. P. Stoecklin, and M. Vlachos, "Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1382–1393, 2010.

[9] Y. Wen, Z. Chen, G. Ma, J. Cao, W. Zheng, G. Peng, S. Li, and W.-L. Huang, "SECOMPAX: A bitmap index compression algorithm," in *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*. IEEE, 2014, pp. 1–7.

[10] J. Chang, Z. Chen, W. Zheng, Y. Wen, J. Cao, and 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*. ACM, 2014, pp. 257–258.

[11] A. Colantonio and R. Di Pietro, "Concise: Compressed ncomposable integer set," *Information Processing Letters*, vol. 110, no. 16, pp. 644–650, 2010.

[12] J. Chang, Z. Chen, W. Zheng, J. Cao, Y. Wen, G. Peng, and W.-L. Huang, "SPLWAH: a bitmap index compression scheme for searching in archival internet traffic," in *Communications (ICC), 2015 IEEE International Conference on*. IEEE, 2015, pp. 7089–7094.

[13] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with roaring bitmaps," *Software: Practice and Experience*, 2015.

[14] Y. Wen, H. Wang, Z. Chen, J. Cao, G. Peng, W. Huang, Z. Hu, J. Zhou, and J. Guo, "MASC: A bitmap index encoding algorithm for fast data retrieval," in *Communications (ICC), 2016 IEEE International Conference on*. IEEE, 2016.

[15] Y. Wu, Z. Chen, J. Cao, H. Li, C. Li, Y. Wang, and W. Zheng, "CAMP: A new bitmap index for data retrieval in traffic archival," *IEEE Communication Letters*, 2016.

[16] S. Kim, J. Lee, S. R. Satti, and B. Moon, "SBH: Super byte-aligned hybrid bitmap compression," *Information Systems*, vol. 62, pp. 155–168, 2016.

[17] Z. Chen, Y. Wen, J. Cao, W. Zheng, J. Chang, Y. Wu, G. Ma, M. Hakmaoui, and G. Peng, "A survey of bitmap index compression algorithms for big data," *Tsinghua Science and Technology*, vol. 20, no. 1, pp. 100–115, 2015.

[18] "Intel intrinsics guide," http://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[19] Y. Wu, Z. Chen, Y. Wen, J. Cao, W. Zheng, and G. Ma, "A general analytical model for spatial and temporal performance of bitmap index compression algorithms in big data," in *2015 24th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2015, pp. 1–10.

[20] "The CAIDA UCSD anonymized internet traces 2013-20130529," http://www.caida.org/data/passive/passive_2013_dataset.xml.

[21] "Clueweb09 gap data set," http://searchivarius.org/personal/data-sets/clueweb09gap.