CAMP: A New Bitmap Index for Data Retrieval in Traffic Archival

Yinjun Wu, Zhen Chen, Junwei Cao, Haoxun Li, Chenxing Li, Yijie Wang, Wenxun Zheng, Jiahui Chang, Jing Zhou, Ziwei Hu, and Jinghong Guo

***Abstract*— Traffic archival, an indispensable task for network analysis, has suffered from a huge amount of data. These rapid growing data can exceed storage capacity and thwart real-time analysis. In order to take insight from Internet traffic, bitmap indexing is applied in this field. However, raw bitmap indexes can consume more space and longer delay for loading larger indexes into memory. This leads to the invention of various bitmap index compression algorithms to save storage and ensure query efficiency. This paper proposes a new algorithm called common affix merging with partition. By merging common affixes in a bit sequence, it saves more storage and conducts at least one order of magnitude faster bitwise operations when compared with ROARING, WAH, CONCISE, and COMPAX. In practice, besides bitwise operations, a query should also contain the operations of loading index files into memory. The experimental results demonstrate that the speedup is still considerable even taking loading time into consideration.**

***Index Terms*— Bitmap index, big data, CAMP, compression, query, traffic archival.**

I. Introduction

CURRENTLY, traffic archival is very important in the field of data-driven network security, which is beneficial to prevent malicious network attack, detect network faults, etc. Unfortunately, it is hard for network analysis because of the fast increasing amount of data. According to a report [1] from Cisco, the Internet traffic data will quadruple in recent years and reach 1.3ZB (that is 1.3×10^{12} GB) in 2016. In that case, how to achieve the goal of real-time data retrieval in archived traffic becomes an enormous challenge.

To achieve efficient query, a reasonable choice is to store all the incoming data in the form of index files in storage such as disks. When a query is launched, corresponding index files are loaded into memory. In order to speed up the following bitwise operations, bitmap indexing is used.

But the efficiency of bitwise operations is often subject to many factors, among which I/O operations are most influential. Because raw bitmap indexes can consume a lot of storage, they can lead to a serious problem in I/O speed.

In order to relief the problem, various bitmap index compression algorithms have been devised, such as BBC [2], WAH [3], PLWAH [4], EWAH [5], PWAH [6], COMPAX [7], SECOMPAX [8], PLWAH+ [9], BREAD [10], CONCISE [11], SPLWAH [12], Roaring bitmap [13] etc. According to a survey [14], some of them are already applied in traffic archival.

Most of the bitmap index compression algorithms currently derive from WAH, which is mainly based on Run Length Encoding (RLE). However, they can increase the time complexity in decompression and the following bitwise operations. Some novel ideas have been presented to resolve this problem. One of the most effective methods is Roaring bitmap. When dealing with in-memory indexes, Roaring can beat WAH and its derivatives in both spatial and temporal performance. However, as referred before, in the field of traffic retrieval, it is inconceivable to store all the data in memory. Thus, in-memory bitmap index algorithms such as Roaring may lose their advantages in this application. In order to demonstrate the advantage of our design, we compare our algorithm with Roaring in experiments.

In this paper, a new algorithm named CAMP is proposed, which differs from previous algorithms based on WAH. Its main feature is to divide raw bitmap indexes into many bitmap snippets and then merge the common affixes. It can achieve better performance in both storage and speed. We also propose an improved version of CAMP, CAMP_block, which is to partition the whole dataset equally into many blocks and apply CAMP for each block.

The following sections are organized as follows. In section II, the motivation and design details of CAMP are presented. In Section III, experimental results based on both synthetic data and real data are provided. The conclusion is presented in Section IV.

II. Methodology

A. Compression Schemes

Fig. 1 shows the compression scheme of CAMP. First, a raw bitmap index containing n bits is partitioned

Manuscript received November 15, 2015; revised March 21, 2016; accepted March 23, 2016. 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), and National Training program of Innovation and Entrepreneurship for Undergraduates (No.201510003049 and No.201510003B066). The associate editor coordinating the review of this paper and approving it for publication was V. Eramo.

Y. Wu and W. Zheng are with the Department of Automation, Tsinghua University, Beijing 100084, China (e-mail: wu-yy12@mails.tsinghua.edu.cn; zhengwxthu@gmail.com).

Z. Chen is with the iCenter of Tsinghua University, Beijing 100084, China (e-mail: zhenchen@tsinghua.edu.cn).

J. Cao is with the Tsinghua National Laboratory for Information Science and Technology, Research Institute of Information Technology, Tsinghua University, Beijing 100084, China (e-mail: jcao@tsinghua.edu.cn).

H. Li is with the Department of Electronic Engineering, Tsinghua University, Beijing 100084, China (e-mail: li-hx13@mails.tsinghua.edu.cn).

C. Li is with the Institute for Interdisciplinary Information Science, Tsinghua University, Beijing 100084, China (e-mail: cx-li13@mails.tsinghua.edu.cn).

Y. Wang is with the Department of Hydraulic Engineering, Tsinghua University, Beijing 100084, China (e-mail: yj-wang14@mails.tsinghua.edu.cn).

J. Chang is with the Department of Aerospace Engineering, Tsinghua University, Beijing 100084, China (e-mail: changjh13@mails.tsinghua.edu.cn).

J. Zhou, Z. Hu, and J. Guo are with the Global Energy Internet Research Institute, Beijing 102209, China (e-mail: zhoujing@geiri.sgcc.com.cn; huziwei@geiri.sgcc.com.cn; guojinghong@geiri.sgcc.com.cn).

Digital Object Identifier 10.1109/LCOMM.2016.2551719

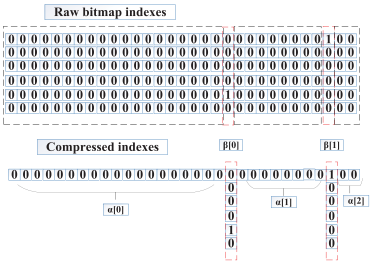

Fig. 1. The compression scheme of CAMP.

TABLE I
THE DISTRIBUTION SEQUENCE OF THE SPACE OVERHEAD OF INTEGER LIST

Bit position interval	[0,10)	[10,100)	··	$[10^k,10^{k+1})$	··	$[10^{\lfloor \log w \rfloor},w)$
Space overhead (bytes)	2	3	··	$k+2$	··	$\lfloor \log w \rfloor+2$
Total number of bits	10	90	··	9×10^k	··	$w-10^{\lfloor \log w \rfloor}$

into many bit snippets in the same length (denoted by $interval$) and common affixes will be merged among all the bit snippets. The common affixes are stored in an array named as $\alpha[i](i = 0, 1\cdots)$ and all the bits in each bit snippet between two adjacent common affixes will compose an array named as $\beta[i](i = 0, 1\cdots)$. In our application, the value of $interval$ is assigned as $\lceil 0.1\sqrt{n} \rceil$.

B. Index Storage

α and β should be stored in the index files after encoding. Details about how to store α and β are described below.

1) Storage Description: Since long sequences of zeros are expected in the real case, run-length encoding is better to compress α, i.e. the numbers of continuous zeros and ones are recorded alternatively.

When the density of 1 is very low, nearly approaching zero, integer list is better to record all the positions of 1s in each $\beta[i](i = 0, 1\cdots)$. Otherwise, it is stored with bit sequence directly. Obviously, there exists a threshold for the conversion between integer list and bit sequence and we found that the threshold depends on the size of given datasets, which could be illustrated via a mathematical model.

2) A Mathematical Model for Calculating the Threshold: In this model, we assume that each $\beta[i](i = 0, 1\cdots)$ contains w bits and bit 1s obey uniform distribution. The density of 1 is denoted as p. Detailed calculation process is shown at https://github.com/thuwuyinjun/CAMP/blob/master/threshold.nb.

In practice, both the integer list and bit sequence will be held in the form of integer arrays in memory and then stored in disks. The occupied space in index files is determined by the number of decimal digits of these integers and other characters such as minus signs.

•$Integer\ list$

If $\beta[i]$ is stored as an integer list which denotes the positions of bit 1s, according to the space overhead of the integers, a distribution sequence is presented in Table I. Since bit 1s follow the uniform distribution, the number of bit 1s is proportional to the density of 1s. So Table I shows expected number of bit 1s within interval $[10^k, 10^{k+1})$.

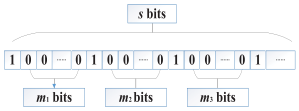

Fig. 2. Binary form of v.

According to Table I, the expected space consumed (denoted as $S_{integer\_list}$) of $\beta[i]$ is provided in Equation (1).

$$\begin{aligned} &S_{integer\_list} \\ &= \sum_{k=1}^{\lfloor logw \rfloor -1} 9\times10^k p(k+2) + 20p \\ &\quad + (\lfloor logw \rfloor + 2)(w - 10^{\lfloor logw \rfloor})p \\ &= \frac{p}{9}(-170 + 8\times10^{\lfloor logw \rfloor} + 9\times10^{\lfloor logw \rfloor}\times\lfloor logw \rfloor) + 20p \\ &\quad + (\lfloor logw \rfloor + 2)(w - 10^{\lfloor logw \rfloor})p \end{aligned} \tag{1}$$

•$Bit\ sequence$

First, we use a function $f(v)$ to represent the probability that a 64-bit integer is smaller than v. In order to count all the possible eligible integers smaller than v (these integers are denoted by I), we consider each bit from the leftmost bit (the 63^{rd} bit) to the rightmost bit (the 0^{th} bit). Fig. 2 shows the binary form of v. According to Fig. 2, the leftmost significant bit locates at $(s-1)^{th}$ bit in v.

Since no bit 1 exists from the 63^{rd} bit to the s^{th} bit in v and I must be smaller than v, the values of the 63^{rd} bit to s^{th} bit should be also all 0s in I. The probability is $(1-p)^{64-s}$. Then for the $(s-1)^{th}$ bit (i.e. the leftmost 1 in v), two possible cases appear in I. The first case is that the $(s-1)^{th}$ bit is 0. In this case, the value of I must be smaller than v no matter what the following bits are. Corresponding probability is $1-p$.

The other case is that the $(s-1)^{th}$ bit in I is 1. Since the following m_1 bits in v are all 0s, they should be all 0s in corresponding positions in I. After that the next 1 appear, we make the same choices as those in the $(s-1)^{th}$ bit. The same procedures proceed recursively. It should be noted that when the procedures above continue until the rightmost 1 (supposing its position is m_t in v), the value in the m_t^{th} bit in I must be 0. Otherwise, the value of I would be greater than or equal to v. Before calculating the value of $f(v)$, an intermediate result (denoted by P_0) can be written as follows based on the analysis above:

$$\begin{aligned} P_0 &= p(1-p)^{m_1}(1-p+p(1-p)^{m_2} \\ &\quad \times(((1-p)+p(1-p)^{m_t-1}(1-p))))p \\ &= p(1-p)^{m_1}h(p)(t\geq 2) \end{aligned} \tag{2}$$

When $t > 2$, in equation (2), we omit the terms of the higher degree in $h(p)$. Then the value of $f(v)$ can be derived as follows:

$$f(v) = (1-p)^{64-s}(1-p+P_0) \approx (1-p)^{64-s}(1-m_1p^2) \tag{3}$$

When $t = 2$ or $t = 1$, $f(v)$ can be calculated directly:

$$\begin{aligned} f(v) &= (1-p)^{64-s}(1-p+p(1-p)^{m_1+1}) \\ &\approx (1-p)^{64-s}(1-(m_1+1)p^2)(t=2) \end{aligned} \tag{4}$$

$$f(v) = (1-p)^{64-s}(1-p) = (1-p)^{65-s}(t=1) \tag{5}$$

TABLE II
THE DISTRIBUTION SEQUENCE FOR A 64-BIT INTEGER v

64-bit integer value	$[0,10)$	$[10,100)$	$\cdots$	$[10^{m-1},10^{m})$	$[10^{m},2^{63})$
Probability	$f(10)$	$f(100)-f(10)$	$\cdots$	$f(10^{m})-f(10^{m-1})$	$1-f(10^{m})$
Storage cost (byte)	2	3	$\cdots$	$m+1$	$m+2$

Based on equation (3) - (5), we can get the distribution sequence for the value of each 64-bit integer v, which is shown in Table II. After considering the sign bit in practice, the expected value of v is:

$$\begin{aligned} Ev = &\sum_{k=1}^{m-1} p(k+2)(f(10^{k+1}) - f(10^{k})) \\ &+ 2 \times f(10) + (1 - f(10^{m})(m+2) \\ &+ \sum_{k=1}^{m-1}(1-p)(k+3)(f(10^{k+1}) - f(10^{k})) + 3 \\ &\times f(10) + (1 - f(10^{m})(m+3) \end{aligned} \quad (6)$$

In a 64-bit integer, since the leftmost bit denotes the sign, the value of m in Table II should be 18. So all the value of $f(10^{k})$ can be enumerated according to equation (3), (4) and (5).

According to equations above, the value of $S_{bit\_sequence}$ is shown below, which is a 61^{st} degree polynomial.

$$\begin{aligned} S_{bit\_sequence} = Ev \times \lceil \frac{w}{64} \rceil = \lceil \frac{w}{64} \rceil \\ \times (2 + 559p - 11033p^2 + 160838p^3) \end{aligned} \quad (7)$$

When $S_{bit\_sequence}$ equals $S_{integer\_list}$, corresponding value of p is the threshold. After derivation, we find that the value of p depends on the size of w. Supposing a dataset contains r rows of records, then $w = \frac{r}{\lceil 0.1\sqrt{r} \rceil}$. Here we give an example. It is assumed that there are 10^6 rows in a database, w will be $\frac{10^6}{\lceil 0.1\sqrt{10^6} \rceil} = 10000$. In this case, when the value of p is determined as 0.046, $S_{bit\_sequence} = S_{integer\_list}$.

3) Further Improvement: To build the indexes using CAMP, common affixes cannot be built until the entire bitmap is checked, which means that we cannot append the indexes of new data onto the end of the existing indexes "on the fly." This is a serious limitation for on-line network monitoring. In order to solve this problem, we propose an improved version called CAMP_block. At first, the whole dataset is separated into many blocks equally and then CAMP is used for each block. Each block contains 1 million rows.

III. EXPERIMENTS

A. Datasets

In order to demonstrate the advantage of CAMP, two datasets are used in experiments. One is synthetic data following the uniform distribution. The other is CAIDA Anonymized Internet Traces 2013 Dataset [15].

1) Synthetic Data: In order to show the overall performance of the synthetic data, the density ranged from 1×10^{-6} to 0.1, which covered most densities in reality. The experiments simulated the case where 1,000,000 rows exist in a database. In order to avoid fortuity, 40 diverse datasets were created for each density. According to Section II-B, the threshold in CAMP for storing β is 0.046.

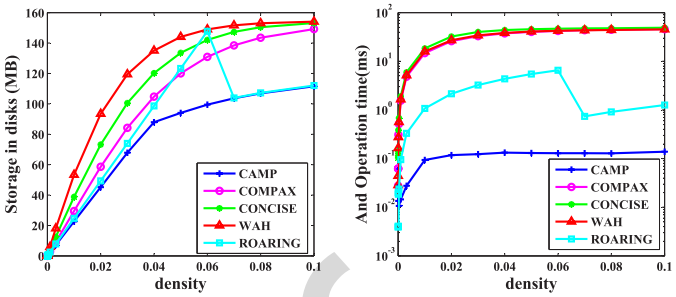

Fig. 3. Storage cost and AND operation time with varied density on synthetic data.

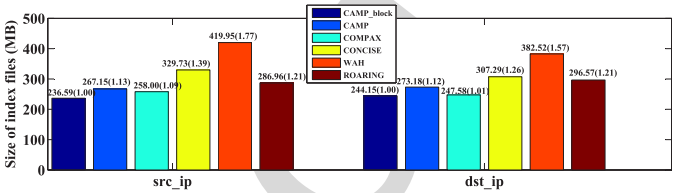

Fig. 4. Storage cost on CAIDA-2013.

2) Real Data: The real data is CAIDA-2013, which traces the real network traffic. It is composed of up to thirteen million quintuples, i.e. source IP address, source port, destination IP address, destination port and protocol type.

The main work in our application was to retrieve the IP addresses. In the following experiments, each byte of IP addresses was treated as an attribute and corresponding indexes were built on these attributes. Since there were 8 bytes in total and each byte varies from 0 to 255, the total number of bitmaps used would be $256 \times 8 = 2048$ and the average density of each bitmap was $1/256 \approx 0.4\%$. Since there were thirteen million records, according to the mathematical model in Section II-B, the threshold for CAMP is 0.037.

B. Experimental Settings

In order to compare CAMP and other traditional algorithms, all the experiments were conducted on the same 64-bit machine with single core, whose CPU is Intel Core i7-3770 and RAM is 18G. All the codes are shown at https://github.com/thuwuyinjun/CAMP.

In our experiments, we compared the performance between CAMP and other 4 algorithms, i.e. Roaring, WAH, COMPAX and CONCISE. COMPAX and CONCISE belong to the WAHs derivatives and they are frequently applied in practice. Besides, as referred before, Roaring is a promising alternative algorithm, which was also used in the experiments.

C. Performance Comparison

1) Synthetic Data: The sizes of index files using various algorithms are given in Fig. 3. In the given dataset, there exists only one block in CAMP_block, the structure of which is same as CAMP except the extra header information in CAMP_block. So we only compare CAMP and other competitors. As shown in Fig. 3, CAMP creates smallest index files in disks in all the cases, although when the density is high enough (greater than $1/16 \approx 6\%$), CAMP and Roaring share nearly the same performance in storage.

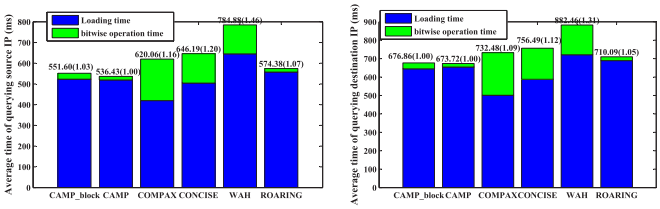

Fig. 5. Time of querying IP addresses.

When conducting bitwise operations (mainly AND operations in our applications, OR operations are also provided in github), each bitwise operation is conducted for 100 times in order to avoid fortuity. Fig. 3 also exhibits the advantage of bitwise operations in CAMP. When the density increases, CAMP can beat its competitors gradually and the speedup can be at least one order of magnitude. It is interesting to see that when the density varies from 0 to 2%, AND operation time grows accordingly. This is reasonable because the number of common affixes will decrease as the density increase. It demonstrates that the compression brought by merging common affixes definitely accelerate bitwise operations.

Admittedly, when the density is greater than about 3%, the operation time keeps nearly the same in CAMP. Obviously, all the common affixes disappear and only the β parts exist. In that case, the reason why CAMP is still faster than WAH and its derivatives is that the bitwise operations in them are conducted word by word (each word contains 32 bits), which is not efficient on a 64-bit CPU. Besides, another serious problem that WAH and its derivatives must face is alignment, which is a time-consuming procedure.

When the density is greater than 3% but less than 6%, bitwise operations with CAMP are faster than that with Roaring. As referred before, all the common affixes disappear in CAMP and the bitwise operations with CAMP equal to that in uncompressed bitmaps, which is certainly faster than intersection between integer lists in Roaring.

When the density is greater than 6%, CAMP and Roaring both contain uncompressed bitmaps. However, they differ drastically in bitwise operations. The reason is that after bitwise operations Roaring needs to convert the result into integer list when the density of the result is less than 1/16, which is completely avoided by CAMP.

2) Real Data: The real dataset is CAIDA-2013 in the experiments. At first, we build indexes on this dataset and then store them in disks waiting for the following queries.

Fig. 4 gives the size of index files in disks using different algorithms. Unfortunately, CAMP does not own the best performance. This is because, in the real network, a single host is likely to send or receive packets continuously, which can create a lot of clustered records in this dataset. However, CAMP_block can have better spatial performance than other competitors overall.

Fig. 5 shows the results of query operations in CAIDA-2013. In order to simulate real queries, we randomly selected 1000 source IP addresses and destination IP addresses which had occurred in our datasets, loaded corresponding index files from disks to memory and then conducted bitwise operations to retrieve targeted IP addresses.

When comparing to WAH and its derivatives, CAMP consumes more time in loading parts. The reason is that the density of β arrays (average value is 0.4%) is much smaller than the threshold (3.7%), which means that β array should be stored as integer list and be added into bit sequence one by one in loading process. The extra time is needed. However, CAMP can compensate it through its fast bitwise operations (up to 10 times). As a result, CAMP is faster than WAH and its derivatives in the whole querying process. The speedup is about 1.16x and 1.09x in source IP and destination IP. Besides, CAMP and CAMP_block consume less querying time than Roaring. The low density leads to the existence of integer lists in Roaring. So it shares nearly the same loading time as CAMP. However, because of smaller index files according to Fig. 4, CAMP consumes less loading time than Roaring, which brings about 1.07× and 1.05× speed-up in the querying process.

IV. Conclusion

The faster bitwise operations are critical for retrieval and query processing of archived network traffic. In this paper, we proposed a new bitmap index compression algorithm named CAMP. CAMP not only achieves better performance in storage, but also achieves faster bitwise operations. Experiments based on synthetic data and real data prove the superiority of this new algorithm. It can be expected that CAMP will have an impressive performance in the field of traffic archival.

References

[1] I. Cisco, "Cisco visual networking index: Forecast and methodology, 2011-2016," CISCO, White Paper 2016, 2011.
[2] G. Antoshenkov, "Byte-aligned bitmap compression," in *Proc. DCC*, 1995, p. 476.
[3] K. Wu, E. J. Otoo, and A. Shoshani, "Compressing bitmap indexes for faster search operations," in *Proc. 14th SSDBM*, 2002, pp. 99–108.
[4] F. Deliège and T. B. Pedersen, "Position list word aligned hybrid: Optimizing space and performance for compressed bitmaps," in *Proc. 13th EDBT*, 2010, pp. 228–239.
[5] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," *Data Knowl. Eng.*, vol. 69, no. 1, pp. 3–28, 2010.
[6] S. J. van Schaik and O. de Moor, "A memory efficient reachability data structure through bit vector compression," in *Proc. SIGMOD*, 2011, pp. 913–924.
[7] F. Fusco, M. P. Stoecklin, and M. Vlachos, "NET-FLi: On-the-fly compression, archiving and indexing of streaming network traffic," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1382–1393, 2010.
[8] Y. Wen *et al.*, "SECOMPAX: A bitmap index compression algorithm," in *Proc. 23rd ICCCN*, Aug. 2014, pp. 1–7.
[9] J. Chang, Z. Chen, W. Zheng, Y. Wen, J. Cao, and W.-L. Huang, "PLWAH+: A bitmap index compressing scheme based on PLWAH," in *Proc. 10th ACM/IEEE ANCS*, Oct. 2014, pp. 257–258.
[10] G. Ma *et al.*, "BreadZip: A combination of network traffic data and bitmap index encoding algorithm," in *Proc. IEEE SMC*, Oct. 2014, pp. 3235–3240.
[11] A. Colantonio and R. Di Pietro, "Concise: Compressed 'n' composable integer set," *Inf. Process. Lett.*, vol. 110, no. 16, pp. 644–650, 2010.
[12] J. Chang *et al.*, "SPLWAH: A bitmap index compression scheme for searching in archival Internet traffic," in *Proc. ICC*, 2015, pp. 7089–7094.
[13] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with Roaring bitmaps," *Softw., Pract. Exper.*, vol. 46, no. 5, pp. 709–719, May 2016.
[14] Z. Chen *et al.*, "A survey of bitmap index compression algorithms for big data," *Tsinghua Sci. Technol.*, vol. 20, no. 1, pp. 100–115, Feb. 2015.
[15] *The CAIDA UCSD Anonymized Internet Traces 2013-20130529*. [Online]. Available: http://www.caida.org/data/passive/passive_2013_dataset.xml