MASC: a Bitmap Index Coding Algorithm for Fast Data Retrieval

***Abstract*—Bitmap index is a structure that can accelerate search queries on low-cardinality attributes. But the space consumption is always a serious problem. As the state-of-art coding schemes, WAH, PLWAH and COMPAX are proposed for bitmap indexes. In this paper, a new bitmap index coding scheme, named MASC, is proposed to further improve the compression performance without impairing the query performance. Instead of being limited to a fixed length (31 bits) in PLWAH and COMPAX, the stride size is set as long as possible to encode consecutive zero bits and nonzero bits in a more compact way. MASC introduces a new structure called "carrier" instead of "piggyback" used in PLWAH as "piggyback" only carries an individual nonzero bit in PLWAH. Based on our experiment with real Internet traffic data set from CAIDA, MASC has a better compression ratio than PLWAH and COMPAX without the penalty of query performance. To satisfy 10Gbps and more high speed network, we propose GPU-MASC, a GPU based implementation of MASC coding schemes. By experiments, GPU-MASC can archive tenfold more acceleration compared with CPU and index millions of packets with the of-the-shelf-commodity GPU. It is also demonstrated that MASC is a practical solution in traffic forensic in Gbps and 10Gbps links with the off-the-shelf-commodity CPU and GPU.**

***Index Terms*—bitmap index coding, bitmap index compression, PLWAH, COMPAX, traffic archival, GPU, Internet security.**

I. Introduction

Indexes provide fast search over large data collections in a limited time. A typical usage is the full-text indexing for web search engine. But these traditional indexes have huge space consumption, which spurs the research on index compression for space and performance issues. Index compression is both interesting in theory and practice. Gonzalo Navarro [1] et al. present a comprehensive survey on this research topic. Falk Scholer [2] et al. find that index compression not only saves the space cost, but also accelerates the query speed. Vo Ngoc Anh and Alistair Moffat [3] et al. also propose a word-aligned binary codes for inverted index compression for full-text search engine. Jeff Dean [4] introduces the Block-Based Index Format, Byte-Aligned Variable-length Encodings and then Group Variant Encoding schemes used in several generations of Google search engine for space optimization and performance improvement.

A bitmap index is a structure that can accelerate search queries on low-cardinality attributes. It is useful in scientific data and traffic archival. But the space consumption is always a serious problem. Ming-Chuan Wu [5] et al. propose the bitmap index coding method and its usage in data warehouse. Kesheng Wu [6] et al. propose WAH (Word-Aligned Hybrid) compression scheme for bitmap indexes and give a practical implementation called Fastbit [7]. As the state-of-art, PLWAH (Position List Word-Aligned Hybrid) [8] and COMPAX (COMPressed Adaptive indeX) [9] compression scheme are proposed for bitmap indexes to make further improvement for the WAH scheme. COMPAX2 is the new version of COMPAX scheme which provides an extended codebook and has better compression ratio with similar query speed compared with COMPAX. It needs notice that COMPAX2 is used in our experiments. Hence when COMPAX is mentioned in later sections, it actually means COMPAX2.

In this paper, a new bitmap index coding scheme, named MASC (MAximized Stride with Carrier), is proposed to further improve the compression performance without impairing the query performance. MASC uses the stride size as long as possible, not limited to 31 bits in PLWAH and COMPAX, to encode the consecutive zero bits and nonzero bits in a more compact way. MASC records origin bitmap index sequences into a new designed format. We demonstrate the validity of MASC with the application in Internet Traffic archival system. Based on our experiments using real Internet traffic data set from CAIDA, MASC has better compression ratio than PLWAH and COMPAX2 for more than 10%.

Besides, in order to satisfy high speed network which can reach more than 10Gbps, Fusco et al. [11] evaluate the GPU based WAH and PLWAH with a sequence of random integers to mimic the five tuples of Internet trace, and prove that the potential of GPU can achieve the speed of indexing millions of packets per second. GPU, typically consists of about one thousand cores, can accelerate encoding of bitmap even further. The GPU-based WAH and PLWAH are also introduced in [12][13]. However those implementations cannot avoid extending the original data into bitmap before processing, which consumes more memory and decreases the performance. GPU-MASC is also proposed and accelerates the coding throughput by more than ten times in this paper.

This paper is organized as follows: Section 2 introduces the background of bitmap index coding scheme. Section 3 describes details in the design principle and coding procedure of MASC. In Section 4, GPU-MASC is demonstrated in detail. Section 5 presents the applications of the proposed MASC in Internet traffic archival system. The experiment result for MASC and GPU-MASC is also presented with real Internet traffic trace from CAIDA in Section 5. Finally, this paper is concluded with future work in Section 6.

II. Bitmap index

A bitmap index is a structure that can accelerate the process of searching queries. Its format is shown in Fig. 1.

However, the shortcoming of bitmap index is that it requires large storage space, which has plenty of room to be improved. A considerable amount of bitmap index coding

algorithms have been raised and several of them are widely-used.

RowID	Column number	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

Fig.1 An example of bitmap index.

III. MASC BITMAP INDEX CODING ALGORITHM

A. *MASC Coding Scheme*

We propose a bitmap index coding algorithm which can reduce the bitmap index size with the comparable query performance with the state-of-art algorithms, i.e., COMPAX and PLWAH. For clarity, a chunk is a fixed size block in 0-1 sequences for encoding operation. For the comparison purpose, the chunk is set to 31 bits as well as indicated in COMPAX and PLWAH. The bit order is from MSB to LSB.

In essential, MASC makes the encoding stride as long as possible and use a concept called "carrier" instead of "piggyback" in PLWAH. The design details of MASC are introduced as follows:

1. A 0-fill word encodes a sequence of consecutive zero bits. For example, the 0-fill word in Fig. 2 encodes 6 chunks (6*31) and 5 bits consecutive zero bits (191 bits in total).

0	0	000000 00000000 00000000 110	00101

Fig.2 0-fill word.

2. A 1-fill word encodes a sequence of consecutive nonzero bits. For example, the 1-fill word in Fig. 3 encodes 6 chunks (6*31) and 5 bits consecutive nonzero bits (191 bits in total).

1	0000000 00000000 00000000 110	00101

Fig.3 1-fill word.

3. A 1-carried 0-fill word encodes a sequence of consecutive zero bits followed by consecutive nonzero bits (at most 30 bits, 1 bit less than a complete chunk).

0	1	10010	000000 00000000 001110	01111

Fig.4 1-carried 0-fill word.

In Fig. 4, the 2nd bit is used as "carrier flag". When the flag is set to 1, it means that the word is a 1-carried 0-fill word. The 3rd-7th bit is carrier, counting the amount of carried consecutive nonzero bits. The 8th-27the bit is used as counter, counting chunks of consecutive zero bits. The 28th-32nd bit is additional counter, counting consecutive zero bits that less than a chunk. In Fig. 4, the entire word represents 14*31+15 consecutive zero bits followed by 18 consecutive nonzero bits.

Actually a 1-carried 0-fill word also belongs to 0-fill word, however a PURE 0-fill word does not carry any nonzero bits with its carrier flag set to 0, while 1-carried 0-fill word (must) carries up to 30 nonzero bits with its carrier flag set to 1. So in following sections, if no special instructions, when a 0-fill word is mentioned, it means the word does not have a carrier.

B. *MASC Coding Procedure*

The coding steps in MASC are explained in detail in the following.

Step 1: Uncompressed bitmap index is divided into equal chunks of 31 bits. There is an example shown in Fig. 5.

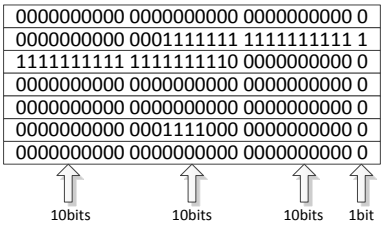

Fig. 5 Uncompressed bitmap index.

Step 2: From the very first word (or chunk), MASC counts consecutive zero bits or nonzero bits and encodes them into MASC form. Fig. 6 to Fig. 10 shows detailed procedure based on the example presented in Fig. 5.

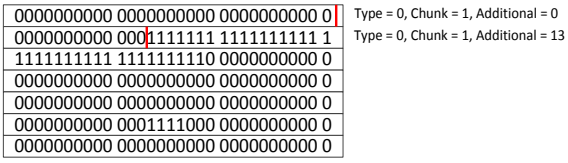

Fig.6 Dealing with the first 0-fill word.

In Fig. 6, all bits in the first chunk are zero bits, and the first 13 bits in the second chunk are zero bits too. So they are encoded into a 0-fill word. However, the next word should be checked to determine whether the two words are combined into 1-carried 0-fill word or not.

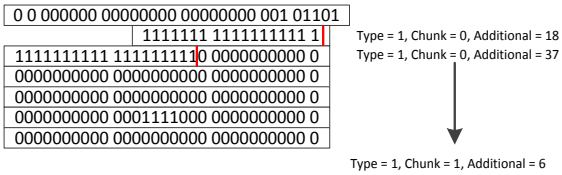

Fig.7 Dealing with the first 1-fill word.

In Fig.7, the last 18 bits in the second chunk and the first 19 bits in the third chunk is nonzero bits. The total number is 37, exceeds 30 which is the limitation of 1-carried 0-fill word. So they are encoded into a single 1-fill word, and the former 0-fill word would not have any carrier.

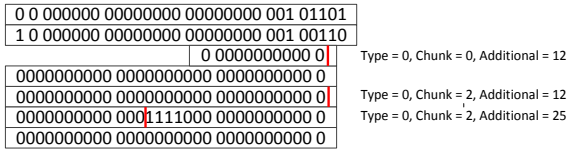

Fig.8 Dealing with the second 0-fill word (step 1)

The last 12 bits in the third chunk, the whole 4th and 5th chunk, and the first 13 bits in the 6th chunk are zero bits. As a result, they are encoded into a single 0-fill word. MASC should still check if it has to combine these zero bits and following consecutive nonzero bits into 1-carried 0-fill word.

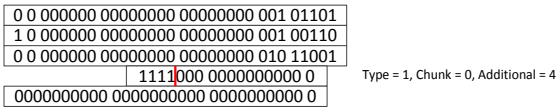

Fig.9 Dealing with the second 0-fill word (step 2)

The 13th to 16th bit in the 6th chunk are nonzero bits, no more than the carrier limitation. So MASC encodes the four consecutive nonzero bits and the former zero bits into a 1-carried 0-fill word.

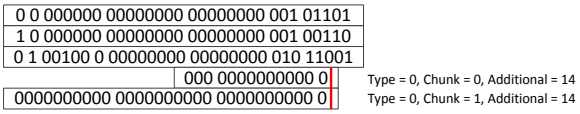

Fig.10 Dealing with the last 0-fill word

The last 14 bits in the 6th chunk and the whole 7th chunk are zero bits. As a consequence, MASC encodes them into a 0-fill word (without carrier). Till now the coding process of MASC has been finished and Fig. 11 is the result.

0 0 000000 00000000 00000000 001 01101
1 0 000000 00000000 00000000 001 00110
0 1 00100 0 00000000 00000000 010 11001
0 0 000000 00000000 00000000 001 01110

Fig.11 The coding result using MASC.

Fig. 11 is the bit sequence after encoding the origin bitmap index by MASC.

Final results of PLWAH and COMPAX are shown in Fig. 12 and Fig.13 respectively. It is clearly observed that both PLWAH and COMPAX encode the original bit sequence into 6 words, while MASC's result consumes only 4 words in all.

1 0 00000 0 00000000 00000000 00000001
0 0000000 00000011 11111111 11111111
0 1111111 11111111 11110000 00000000
1 0 00000 0 00000000 00000000 00000010
0 0000000 00000011 11000000 00000000
1 0 00000 0 00000000 00000000 00000001

Fig.12 The encoding result using PLWAH.

000 00000 00000000 00000000 00000001
1 0000000 00000011 11111111 11111111
1 1111111 11111111 11110000 00000000
000 00000 00000000 00000000 00000010
1 0000000 00000011 11000000 00000000
000 00000 00000000 00000000 00000001

Fig.13 The encoding result using COMPAX.

C. *Comparison among MASC, PLWAH and COMPAX*

The concept "carrier" in MASC and "piggybacked" in PLWAH are similar. However, the carrier can carry at most 30 nonzero bits while PLWAH can piggyback only a single nonzero bit. Besides, MASC generalizes the concept of literal word and eventually obsolete this concept. As a consequence, several (no more than 30) nonzero bits can be carried by the former 0-fill word and output a 1-carried 0-fill word, while PLWAH has to encode them in a literal word or two literal words in the worst condition when the consecutive nonzero bits locate in two adjacent chunks. Considering zero bits' and nonzero bits' distribution in real data set, zero bits and nonzero bits appear usually in batch especially after being sorted by the hash value of each record. Thus MASC can perform much better than PLWAH.

Though both MASC and COMPAX use the concept of fill words, there are many differences between them. In Section 4, experiments are conducted to show MASC's advantages over the other two coding schemes.

D. *Query Table*

Inspired by Group Variant Encoding scheme used in Google, some modifications are made on plain MASC algorithm to raise the query efficiency. An extra bit, i.e. the second bit is used as a flag too. Similar to the 256-entry table used by Google, we introduce the concept of query table which contains type tag and position offset in an encoding window. The encoding window size is typically set to 4k in our experiment carried out in Section 4.

The second bit of 1-fill words are set to 1. Fig. 14 shows the new 1-fill word.

1	1000000 00000000 00000000 110	00101

Fig.14. a 1-fill word after revision.

The encoded bit sequence shown in Fig. 11 is converted into those in Fig.15 as the original bit sequence shown in Fig. 5. The italics bit shows the difference between two results.

0 0 000000 00000000 00000000 001 01101
1 *1* 000000 00000000 00000000 001 00110
0 1 00100 0 00000000 00000000 010 11001
0 0 000000 00000000 00000000 001 01110

Fig.15. The new encoding result from original bit sequence in Fig.5

Query table is used during query process. It consists of two parts: type tag and position offset.

A *type tag* shows whether the word contains nonzero bits. The second bit in a 0-fill word is set to 0 and all of the rest's second bit is 1, as a consequence the type tag can be directly duplicated from the second bit of words.

The *position offset* consists of chunk offset and bit offset. Chunk offset shows the number of chunks from the first one of current encoding window, while bit offset is determined by the number of bits from the first one of current chunk. The number of bits needed to represent the bit offset is 5. While bits needed for chunk offset is flexible, the number of bits is set it to 7 in this example. Fig. 16 shows the whole query table for the example in Fig. 15.

0	0000000	00000
1	0000001	01101
1	0000010	10011
0	0000101	10001

Fig.16 Query table for the example in Fig. 5.

Consider a computer can carry out a bitwise sum in a seconds and carry out a logical sum in b seconds. Let total words after encoding procedure to be W. Before introducing query table, assume that the average time for querying for a particular record is A and the average time querying for all records hit by a particular prerequisite (e.g. querying for all packets satisfying the first byte of source IP in traffic is 166) is B. Then $A = 0.75Wa + Wb$ and $B = 1.5Wa + 2Wb$. If the whole records are compressed together, using query table, the

time for querying for a particular record remains unchanged, but the time for querying for all records belong to a particular prerequisite (such as the same query above) declines to *Wa* + *Wb*.

IV. GPU-MASC

GPU-MASC is another version of MASC which is builded on GPUs in order to take advantage of the parallel-computing character. Detailed definitions and algorithm explanation are presented as follows.

A. Definitions

Before the introduction of parallelization algorithms, some definitions are given as follows.

Consecutive value sequence: A sequence of some identical value belongs to an attribute is classified as a consecutive value sequence. The initial position of a consecutive value sequence is the first place that the value appears, and the final position is the last place the value appears.

Attribute Table (A-Table): Contains the *initial position* and the *final position* of a consecutive value sequence, each record of A-table has parameters representing the value, the initial position and the final position of that sequence.

Fill Table (F-table for short): Contains all fill words in the order of value (first priority) and row id (second priority), prepared for further encoding. Each record contains the fill word's type (0-fill, 1-fill, carried-0-fill), counter, and carrier counter (only for carried 0-fill). The F-table needs an index to inform its distribution by value, so F-table index is introduced containing each value's initial position.

B. Index Creation and Encoding

Exploiting the massive parallelism offered by GPUs needs a complete algorithm redesign. The core idea of the approach is to exploit the capability in integer sorting provided by modern GPUs to be able to build different attributes of input data and even different bitmap index columns for each attribute and in parallel. Different attributes are dealt with by distributed into different blocks in GPUs, thus blocks can run independently. The following paragraphs focus on parallelization of different index columns.

Data Preprocessing. At first each input value is associated with a row identifier (row ID, or rid for short), which encodes its position in the value. GPU-MASC creates a vector storing the row ID of each input, and then sorts the row ID with the input value array as keys through stable-sorting. This is implement by Thrust, a CUDA library provided by NVIDIA. Then the output would be a sequence of tuples like (value, row ID), where values are listed in ascending order. Particularly for stable-sorting, the row ID is listed in ascending order too within the sub-sequence whose elements have the same value**.** Algorithm 1 shows the general blueprint of GPU-MASC.

Algorithm 1 GPU-MASC

Input: value array
Output: encoding result of bitmap index (result), result's index.
1: AttTableIdx ← AttTableIdxCreation (value, rid)
2: (AttTable, AttTableValueIdx) ← AttTableCreation (value, rid, AttTableIdx);
3: (FillTable, FillTableIdx) ← FillTableCreation (AttTable,AttTableValueIdx);
4: mergeFills (FillTable, FillTableIdx);
5: (result, resultIdx) ← Encoding (FillTable, FillTableIdx);
6: copyFromGPUToCPU(result, resultIdx);

GPU-MASC encoding. At first GPU-MASC has to create the index of A-table for allocation room. In algorithm 2, the parallelization is accomplished by equally dividing input values to different threads, with adjacent threads share one same bit. For example, the first thread deal with 1st to 32nd value, while the second thread deal with 32nd to 63rd value, in this case the 32nd value is shared by the adjacent threads. This can effectively find out all initial positions and final positions without omission. The start and end positions differs among different threads in GPU.

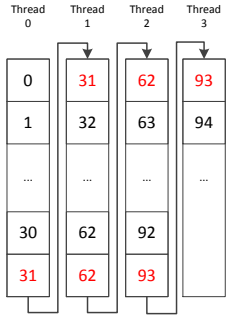

Fig. 17 Parallelization in threads

Algorithm 2 AttTableIdxCreation

Input: the value array, the row id array.
Output: attribute table's index
1: for i = start → end **do in parallel**
2: **if** value[i] ≠ value[i-1] **or** rid[i] – rid[i-1] ≠ 1 **then**
3. AttTableIdx[threadNum]++;
4. **end if**
5: i ← i+1;
6: end for
7: AttTableIdx[threadNum] = Sum (AttTableIdx[i], i = 0 → threadNum - 1);

After creation of A-table index, all consecutive value sequences with their initial positions and final positions can be put in A-table in the form of records. Records in A-table is sorted by value in ascending order. When two records share the same value, they are ordered by the initial position. Meanwhile A-table value index is produced which is indispensable for latter procedures. Parallelization here is also using equal chunks with sharing bits between adjacent threads (chunks).

Algorithm 3 AttTableCreation

Input: the value array, the row id array, attribute table index
Output: attribute table, attribute table's value index
1. tablePos = AttTableIdx[threadNum];
2: **for** i = start → end **do in parallel**
3: **if** value[i] ≠ value[i-1] **then**
4: AttTable.value[tablePos] ← value[i];
5: AttTable.iniPos[tablePos] ← rid[i];
6: AttTable.finPos[tablePos-1] ← rid[i-1];
7: tablePos ← tablePos + 1;
8: AttTableValIdx[value[i]] = rid[i];
9: **else if** rid[i] – rid[i-1] ≠ 1 **then** //means not in the same record
10: AttTable.iniPos[tablePos] ← rid[i];
11: AttTable.finPos[tablePos-1] ← rid[i-1];
12: tablePos ← tablePos + 1;
13: **end if**
14: i ← i+1;
15: end for

Next, A-table and A-table value index are used to produce F-Table. Because records in A-table contains all consecutive value sequences, GPU-MASC doesn't need the value array when creating F-table and its index, all input parameters needed is A-Table and A-table value index. The method of parallelization is different from those above. Thanks to the A-table value index, GPU-MASC separates different values so that sequences of different values can be encoded at the same time in GPU. The values of start and end are decided by value which the thread is dealing with and the A-table value index.

Algorithm 4 FillTableCreation

```
Input: attribute table, attribute table value index
Output: fill table, fill table index
1:  FillTableIdx[threadNum] = 2*AttTableValIdx[threadNum] + threadNum;
2:  for i = start → end do in parallel
3:      if i == start then
4:          if AttTable.iniPos[i] ≠ 0 then
5:              FillTable ← FillTable + (0, AttTable.iniPos[i] - 1);
6:          end if
7:          FillTable ←FillTable + (1, AttTable.finPos[i] – AttTable.finPos[i]+1);
8:      else
9:          FillTable ←FillTable+(0,AttTable.iniPos[i]–AttTable.finPos[i-1]+1);
10:         FillTable ← (1,AttTable.finPos[i] – AttTable.iniPos[i] - 1);
11:     end if
12:     if i == end and AttTable.finPos[i] ≠ SeqLen - 1 then
13:         FillTable ← (0, SeqLen – AttTable.finPos[i] -1);
14:     end if
15:     i ← i+1;
16: end for
```

SeqLen mentioned in Algorithm 4 is the length of input value array. That number is pre-stored and can be modified in practice.

Based on MASC, records in F-Table may be carried by the former one. A 0-fill word and a 1-fill word is available to be combined when length of the 1-fill word is no more than 30. Different thread merge fill words of its own value in parallel.

Algorithm 5 MergeFills

```
Input: fill table, fill table's index
Output: fill table
1:  for i = start → end do in parallel
2:      if FillTable.type[i] == 0 and FillTable.type[i+1] == 1 then
3:          if FillTable.len[i+1] < 31 then
4:              FillTable.type[i] ← CARRIED
5:              FillTable.carLen[i] ← FillTable.len[i+1];
6:              FillTable.type[i+1] ← BLANK;
7:              i ← i+1;
8:          end if
9:      end if
10:     i ← i+1
11: end for
```

Next GPU-MASC encodes records in F-table into 32-bit words. GPU-MASC uses unsigned int for storing each word. When an unsigned int number is set to 0, it means this word doesn't belong to final result and will be removed in latter stages.

Algorithm 6 Encoding

```
Input: fill table, fill table's index.
Output: result, result's index
1:  for i = start → end do in parallel
2:      if FillTable.type[i] == 0 then
3:          result[i] ← FillTable.len[i]/31;
4:          result[i] ← result[i] << 5;
5:          result[i] ← result[i] + mod(FillTable.len[i],31);
6:      else if FillTable.type[i] == 1 then
7:          result[i] ← 0x04000000;
8:          result[i] ← result[i] + (FillTable.len[i]/31);
9:          result[i] ← result[i] << 5;
10:         result[i] ← result[i] + mod(FillTable.len[i],31);
11:     else if FillTable.type[i] == CARRIED then
12:         result[i] ← 0x00000020;
13:         result[i] ← result[i] + FillTable.carLen[i];
14:         result[i] ← result[i] << 22;
15:         result[i] ← result[i] + (FillTable.len[i]/31);
16:         result[i] ← result[i] << 5;
17:         result[i] ← result[i] + mod(FillTable.len[i],31);
18:     else if FillTable.type[i] == BLANK then
19:       result[i] = 0;
20:     end if
21:     i ← i+1;
22: end for
```

After encoding the bitmap index, GPU-MASC copies result and result's indexes from GPU to CPU and then writes the result into files. Each columns' positions is recorded in F-table's indexes. When writing from CPU to files, GPU-MASC omits all zero words (0x00000000) in the result array because zero words are formed from records whose type is blank in F-table after being merged, and those zero numbers mean nothing and should not be written into bitmap index files

V. Applications

A. *Traffic Forensic with MASC*

Cisco [16] predicts that the volume of Internet traffic will quadruple between 2011 and 2016 reaching 1.3 Zettabytes per year in 2016. According to the internal statistics of China Unicom [24], mobile user traffic increases rapidly with CAGR (Compound Annual Growth Rate) of 135%. From the data of China Unicom in 2013, its monthly records are more than 2 trillion (2×10^{12}), monthly data volume is over 525TB, and has reached 5PB.

L. Deri and F. Fusco [17-18] propose MicroCloud-based flow aggregation for fixed and mobile networks. This architecture is used to provide real-time traffic monitoring and correlation in large distributed environments.

P. Giura and N. Memon [19] propose NetStore, a column oriented storage with IP address based on inverted indexes for fast retrieval. Each of the segments within a column in NetStore is compressed independently. They also discuss different possible compression methods. As Netstore maintains the strict time order of packets, it does not consider the reordering of packet based on flow level and utilizing bitmap indexing compression in its system.

A 10 Gbps network link can arrive at a maximum of 14.8 million packets per second. It is a big challenge to index these packets in one second. For any mobile network operator manages several such links, even records only flow statistics, the volume of resulting data could easily reach Terabytes in one year.

We apply the proposed MASC coding scheme to record the origin bitmap index sequences into a new format. In the following sections, The performance of three bitmap index compression codings is evaluated using the real data trace from CAIDA. Our experiment results suggest that MASC has the best performance among all of the three coding schemes.

B. *MASC's Performance Evaluation with Real Data*

The real Internet trace data from CAIDA are chosen to evaluate three bitmap indexes coding schemes. Comparisons are made among MASC and the state-of-art algorithms, i.e., PLWAH and COMPAX. There are totally 13,578,496 IPv4 packets in the CAIDA's data set (this method apply in IPv6 too). Source IPs are chosen in our experiment for bitmap indexing.

At first the coding schemes calculate hash values of each packet's five tuples and reorder packets by hash values. This procedure is similar to oLSH function in COMPAX for better compression ratio. The difference is the hashing method being used is the uniform hashing instead of locality hashing.

TABLE 1.1 CAIDA DATASET ATTRIBUTES

Column	*Type*	*Bytes*
Source IP	int	4
Destination IP	int	4
Source Port	short	2
Destination Port	short	2
Protocol	byte	1

After that, packets belong to the same flow can be aggregated. Then we pick out each byte in source IP address (four bytes in all) of each packet respectively, convert each byte into bitmap index and write the bitmap index into files for further coding. After that, each file corresponds to a column of bitmap index as shown in Fig. 18. Then 256*4 bitmap files are encoded by three coding algorithms, i.e., PLWAH, COMPAX and MASC respectively, to evaluate the coding schemes and make comparison among them.

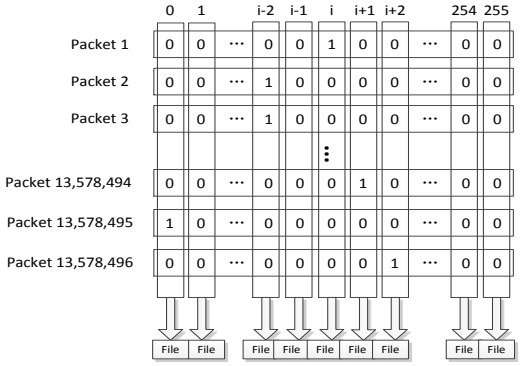

Fig. 18 Bitmap indexes of a byte from source IP in Real Internet Trace.

From Fig. 19-21, it is clearly shown that MASC has a better compression ratio and smaller disk consumption than COMPAX and PLWAH. Detailed information is illustrated in Table 2- 4.

TABLE 2.1 DISK CONSUMPTION AMONG THREE ALGORITHM SCHEMES (SOURCE IP)

	Byte 1	*Byte 2*	*Byte 3*	*Byte 4*	*Sum*
PLWAH	54,214,560	55,329,248	56,368,288	57,185,856	223,097,952
COMPAX	51,052,672	54,872,928	56,141,312	57,092,320	219,159,232
MASC	45,437,856	45,627,296	45,804,480	45,922,496	182,792,128

TABLE 2.2 DISK CONSUMPTION AMONG THREE ALGORITHM SCHEMES (DESTINATION IP)

	Byte 1	*Byte 2*	*Byte 3*	*Byte 4*	*Sum*
PLWAH	54,510,304	56,336,512	56,791,328	57,105,536	224,743,680
COMPAX	50,742,688	54,793,568	56,233,472	56,848,032	218,617,760
MASC	45,518,272	45,813,056	45,873,216	45,911,968	183,116,512

TABLE 2.3 DISK CONSUMPTION AMONG THREE ALGORITHM SCHEMES (PORTS)

	Source Port(former byte)	*Source Port(latter byte)*	*Destination Port(former byte)*	*Destination Port(latter byte)*
PLWAH	53,636,416	52,786,176	52,374,976	50,406,464
COMPAX	54,011,232	52,992,608	53,180,640	51,223,744
MASC	45,272,704	44,948,416	45,143,136	44,312,960

TABLE 3 MASC VS. PLWAH

Disk Consumption Compared with PLWAH					
	Byte 1	*Byte 2*	*Byte 3*	*Byte 4*	*Sum*
Source IP	-16.19%	-17.53%	-19.03%	-19.70%	**-18.07%**
Dest IP	-16.49%	-18.70%	-19.22%	-19.60%	**-18.52%**
Ports	-15.59%	-14.85%	-13.81%	-12.09%	*

TABLE 4 MASC VS. COMPAX

Disk Consumption Compared with COMPAX					
	Byte 1	*Byte 2*	*Byte 3*	*Byte 4*	*Sum*
Source IP	-11.00%	-16.85%	-18.41%	-19.56%	**-16.59%**
Dest IP	-10.30%	-16.39%	-18.42%	-19.24%	**-16.24%**
Ports	-16.18%	-15.18%	-15.11%	-13.49%	*

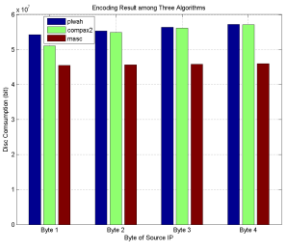

Fig. 19 The size source IP after compression using three coding schemes.

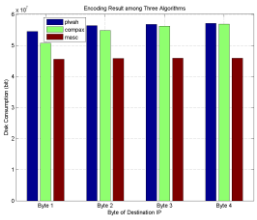

Fig. 20 The size of destination IP after compression using three coding schemes.

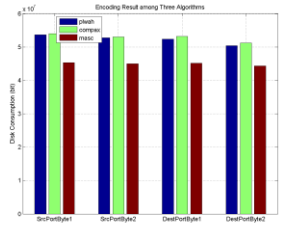

Fig. 21 The size of ports after compression using three coding schemes.

From Table 2-4, MASC's disk consumption is 18.07% less than PLWAH's and 16.59% less than COMPAX. Fig. 22-33 show the relationship between nonzero bits (using the number of matching results as equality, because the number of nonzero bits is equal to the number of matching results) and room consumption. The result is also illustrated by Fig. 12 in F. Fusco's paper [11].

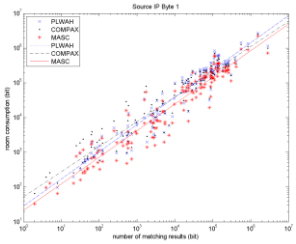

Fig. 22 Relationship between number of matching results and room consumption in byte 1 of source IP

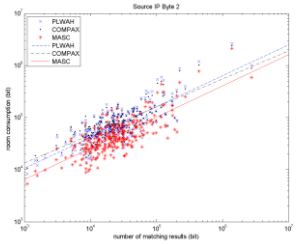

Fig. 23 Relationship between number of matching results and room consumption in byte 2 of source IP

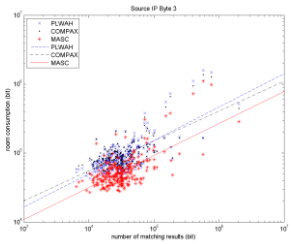

Fig. 24 Relationship between number of matching results and room consumption in byte 3 of source IP

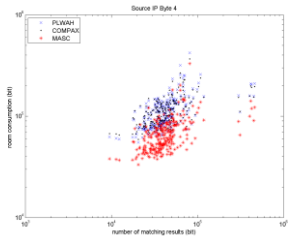

Fig. 25 Relationship between number of matching results and room consumption in byte 4 of source IP

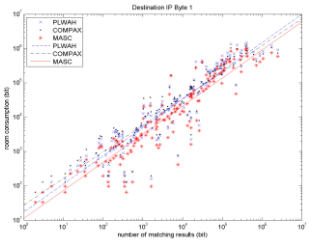

Fig. 26 Relationship between number of matching results and room consumption in byte 1 of destination IP

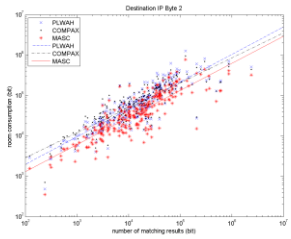

Fig. 27 Relationship between number of matching results and room consumption in byte 2 of destination IP

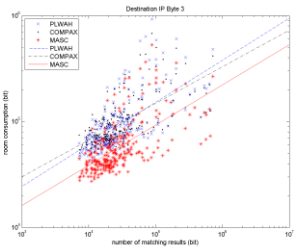

Fig. 28 Relationship between number of matching results and room consumption in byte 3 of destination IP

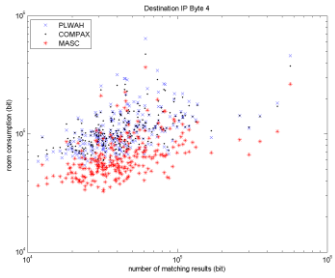

Fig. 29 Relationship between number of matching results and room consumption in byte 4 of destination IP

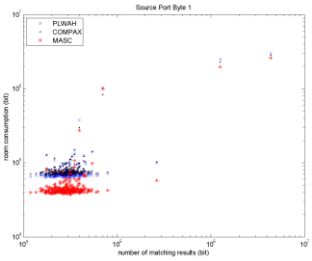

Fig. 30 Relationship between number of matching results and room consumption in byte 1 of source port

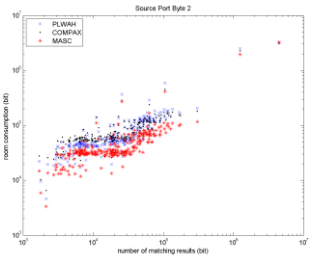

Fig. 31 Relationship between number of matching results and room consumption in byte 2 of source port

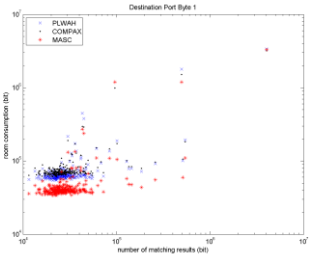

Fig. 32 Relationship between number of matching results and room consumption in byte 1 of destination port

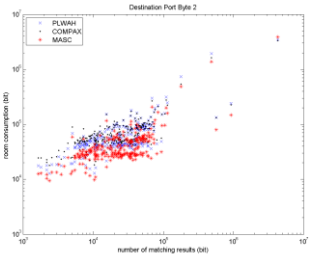

Fig. 33 Relationship between number of matching results and room consumption in byte 2 of destination port

C. *The Benefit of Query Table for Query Performance*

To pick out all records belong to a particular prerequisite (such as source IP = *166.*.*.**), the plain MASC has to make more than 2.15M bitwise sum operations and 1.43M logical sum operations on average. After complementing MASC with query table, now it can make 1.43M bitwise sum operations and 700K logical sum operations. That is because the type tag lessen the workload of bitwise sum and the position offset reduces the number of logical sum operations, since position offset frees MASC from calculating positions once and again.

Meanwhile, PLWAH has to carry out more than 1.74M bitwise sum and 870K logical sum operations, and COMPAX makes 2.05M bitwise sum and 850K logical sum operations. As a result, MASC with query table has comparable querying efficiency as PLWAH and COMPAX.

D. *GPU-MASC's Evaluation*

GPU based MASC algorithm is implemented with Thrust, a C++ library provided by the NVIDIA SDK designed to enhance code productivity and more importantly performance portability across NVIDIA GPUs. To evaluate the performance of implementation, we use similarly priced CPU and GPU: a 3.4 GHz Intel i7-2600K processor with 8 Mb of cache and a NVIDIA GTX-760 GPU fitted in a PCI-e Gen 2.0 slot.

The input data being used are anonymous Internet trace data set from CAIDA, totally 13,581,810 packets. The five tuples of Internet traces (source IP, destination IP, source port, destination port, protocol number) are picked out. For simulating circumstances in practice, packets are cut by 3,968 (128*31), and bitmap indexes are created for 14 (bytes) * 3,968 once at a time.

GPU-MASC vs. MASC Fisrtly we construct and compress bitmap index using GPU-MASC and compare its result with the encoding result of MASC. Their results of roon consumption is the same, and room consumption is shown in Fig. 34. However, GPU-MASC can build bitmap indexes for 128*31 packets in 8.057 milliseconds, while MASC takes 157.3 millisecond, because MASC cannot encoding in parallel on CPU. Thus, GPU-MASC improves encoding speed by 19.5 times.

Based on Fig. 35, the throughput of GPU-MASC is 492,491 packets per second. However, GPU-MASC constructs and encodes bitmap indexes for all 14 bytes in the five tuple for Internet trace packets, while other algorithms on GPU only construct 2 bytes in the five tuple one at a time. So the equivalent thoughput for GPU-MASC is 3,447,437 packets. per second.

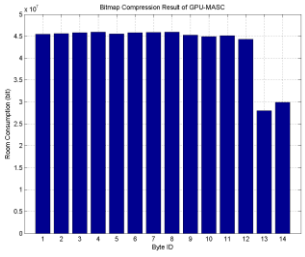

Fig. 34 Bitmap indexes room consumption of GPU-MASC

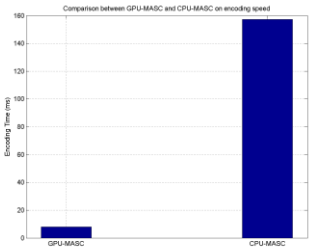

Fig. 35 Encoding speed comparsion between GPU-MASC and CPU-MASC

VI. Conclusion

In this paper, we propose a new bitmap index coding algorithm named MASC. Instead of 31 bits in PLWAH and COMPAX, MASC uses the stride size as long as possible to encode the consecutive zero bits and nonzero bits in a more compact way. As inspired by index format from Google, the query table is also introduced so that the query process can be far better than the plain MASC version without query table. In experiments based on real Internet data traces, MASC shows a better compression ratio than PLWAH and COMPAX schemes without impairing the query performance in practice.

In order to meet the need for dealing with high speed network, GPU-MASC is designed for running MASC on GPUs. The experiments show that GPU-MASC can improve coding speed by 19.5 times and reach the speed of about 3.5 million packets per second.

In the future, we will carry out more experiment on the compression ratio and query time to make all-round evaluation on the new algorithms.

References

[1] G. Navarro, and V. Mäkinen, "Compressed Full-text indexes", ACM Computing Surveys (CSUR) 39, no. 1 (2007): 2.
[2] F. Scholer et al., "Compression of Inverted Indexes for Fast Query Evaluation", In Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 222-229. ACM, 2002.
[3] V. N. Anh and A. Moffat, "Inverted Index Compression Using Word-aligned Binary Codes", Information Retrieval 8, no. 1 (2005): 151-166.
[4] J. Dean. "Challenges in Building Large-scale Information Retrieval Systems: Invited Talk", In Proceedings of the Second ACM International Conference on Web Search and Data Mining, pp. 1-1. ACM, 2009.
[5] M. Wu et al., “Encoded Bitmap Indexes and Their Use for Data Warehouse Optimization”, Shaker, 2001.
[6] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing Bitmap Indices with Efficient Compression", ACM Transactions on Database Systems (TODS) 31, no. 1 (2006): 1-38.
[7] FastBit. An Efficient Compressed Bitmap Index Technology. https://sdm.lbl.gov/fastbit/
[8] F. Deli`ege and T. B. Pedersen, “Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps”, In Proc. of the 13th Int. Conf. on Extending Database Technology, EDBT ‘10, 2010.
[9] F. Fusco, M. Stoecklin, and M. Vlachos, “NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic”, In Proceedings of the International Conference on Very Large DataBases (VLDB), pp. 1382–1393, 2010.
[10] CAIDA, www.caida.org.
[11] F. Fusco, M. Vlachos, and M. Stoecklin, "Real-time Creation of Bitmap Indexes on Streaming Network Data", The VLDB Journal-The International Journal on Very Large Data Bases 21, no. 3 (2012): 287-307.
[12] F. Fusco, M. Vlachos, X. Dimitropoulos, and L. Deri. "Indexing Million of Packets Per Second Using GPUs", In Proceedings of the 2013 Conference on Internet Measurement, pp. 327-332. ACM, 2013.
[13]W. Andrzejewski, and R. Wrembel. “GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid”, Database and Expert Systems Applications (pp. 315-329). Springer Berlin Heidelberg, January, 2010.
[14]W. Andrzejewski., and R. Wrembel, “GPU-PLWAH: GPU-based Implementation of the PLWAH Algorithm for Compressing Bitmaps”, Control and Cybernetics, 40(3), 2011.
[15] D. Lemire, O. Kaser, and K. Aouiche. “Sorting Improves Word-Aligned Bitmap Indexes”, CoRR, abs/0901.3751, 2009.
[16] Cisco Visual Networking Index Forecast (2011 - 2016). http://www.cisco.com/web/solutions/sp/vni/vni_forecast_highlights/index.html.
[17] L. Deri, and F. Fusco, “MicroCloud-based Network Traffic Monitoring”, IFIP/IEEE International Symposium on Integrated Network Management (IM), 2013.
[18] L. Deri, and F. Fusco, “Realtime MicroCloud-based flow aggregation for fixed and mobile networks”, In 9th IEEE International Wireless Communications and Mobile Computing Conference (IWCMC), pp.96-101, 2013.
[19] P. Giura and N. Memon, "NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring", In Recent Advances in Intrusion Detection, pp. 277-296. Springer Berlin Heidelberg, 2010.
[20] Z. Chen et al., “Cloud Computing-Based Forensic Analysis for Collaborative Network Security Management System”, Tsinghua Science and Technology, pp.40-50, vol.18, No.1, 2013.
[21] Z. Chen et al., “TIFAflow: Enhancing Traffic Archiving System with Flow Granularity for Forensic Analysis in Network Security”, Tsinghua Science and Technology, vol.18, No.4, 2013.
[22] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, “Enriching Network Security Analysis with Time Travel”, In Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, New York, USA, pp.183-194.
[23] J. Li et al., “TIFA: Enabling Real-Time Querying and Storage of Massive Stream Data”, In Proc. of International Conference on Networking and Distributed Computing (ICNDC), 2011.
[24] W. Huang, Z. Chen, W. Dong, and H. Li, “Mobile Internet big data platform in China Unicom”, Tsinghua Science and Technology, Volume 19, Issue 1, pp. 95-101, Feb. 2014.
[25] Z Chen, W Dong, H Li, P Zhang, X Chen, and J Cao, “Collaborative Network Security in Multi-tenant Data Center for Cloud Computing”, Tsinghua Science and Technology, 19 (1), pp.82-94, Feb. 2014.