The VLDB Journal (2012) 21:287–307
DOI 10.1007/s00778-011-0242-x

REGULAR PAPER

Real-time creation of bitmap indexes on streaming network data

Francesco Fusco · Michail Vlachos · Marc Ph. Stoecklin

Received: 9 December 2010 / Revised: 28 April 2011 / Accepted: 24 June 2011 / Published online: 30 July 2011

Abstract High-speed archival and indexing solutions of streaming traffic are growing in importance for applications such as monitoring, forensic analysis, and auditing. Many large institutions require fast solutions to support expedient analysis of historical network data, particularly in case of security breaches. However, "turning back the clock" is not a trivial task. The first major challenge is that such a technology needs to support data archiving under extremely high-speed insertion rates. Moreover, the archives created have to be stored in a compressed format that is still amenable to indexing and search. The above requirements make general-purpose databases unsuitable for this task and dedicated solutions are required. This work describes a solution for high-speed archival storage, indexing, and data querying on network flow information. We make the two following important contributions: (a) we propose a novel compressed bitmap index approach that significantly reduces both CPU load and disk consumption and, (b) we introduce an online stream reordering mechanism that further reduces space requirements and improves the time for data retrieval. The reordering methodology is based on the principles of locality-sensitive hashing (LSH) and also of interest for other bitmap creation techniques. Because of the synergy of these two components, our solution can sustain data insertion rates that reach 500,000–1 million records per second. To put these numbers into perspective, typical commercial network flow solutions can currently process 20,000–60,000 flows per second. In addition, our system offers interactive query response times that enable administrators to perform complex analysis tasks on the fly. Our technique is directly amenable to parallel execution, allowing its application in domains that are challenged by large volumes of historical measurement data, such as network auditing, traffic behavior analysis, and large-scale data visualization in service provider networks.

Keywords Bitmap index · Locality sensitive hashing · Data stream · Data archive

F. Fusco (✉) · M. Vlachos
IBM Research - Zurich, Rüschlikon, Switzerland
e-mail: ffu@zurich.ibm.com

M. Ph. Stoecklin
IBM Research - T. J. Watson Research Center, Hawthorne, NY, USA

1 Introduction

Corporate and service provider networks, financial institutions, and high-security data centers are increasingly interested in tools that allow them to archive network traffic information for postmortem analysis. Imagine for example, the case of an identified data breach at a financial institution: the system administrator would like to quickly pinpoint the accessed nodes from the list of suspected IP addresses to isolate additional compromised nodes. A similar scenario is also encountered during the outbreak of a computer worm, when one would like to identify the computer nodes that have been contacted by the compromised system.

To support the above functionality, all inbound and outbound traffic can be recorded in order to recreate the original breach or attack conditions. Archival solutions typically do not record the actual packet content of network traffic, something that would result in repositories of prohibitive size and severely compromise user privacy. The recorded information focuses on *network flow* data. In that way, one can still capture extended information on the graph connectivity, such as source and destination IP addresses, ports, protocols, and time.

However, even when only storing network flows, huge repositories can be accumulated over time. Currently, a typ-

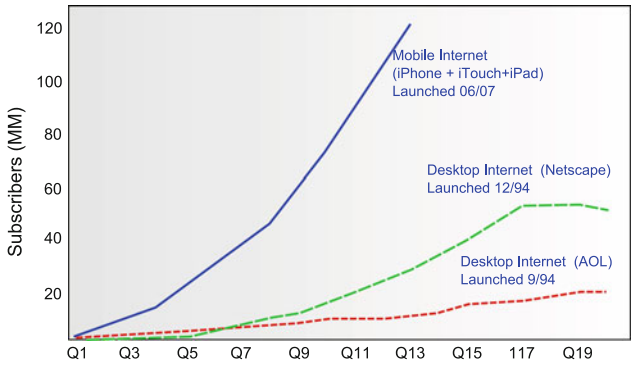

Fig. 1 Exponential growth in the number of mobile devices suggests that the number of network flows will increase at a similar rate in the near future. Compare, for example, the growth of desktop internet, which was not nearly as rapid (data source: Morgan Stanley)

ical service provider network may exhibit flow export rates as high as 50,000 flows per second; such rates amount to more than 8 GB of raw flow information per hour. This means that, even when using data compression, the resulting data repository could easily reach the order of Terabytes on a yearly basis. The task of capturing flow data at high speeds is expected to be aggravated in the near future. Recent survey data suggest an exponential increase in the number of mobile devices that generate network traffic (Fig. 1), which directly implies an equivalent growth in the captured network flows. Even sifting through such enormous databases is not trivial, particularly when one is interested in identifying "needles in the haystack".

To effectively address the above issues, two mechanisms need to be in place: (i) a high-performance data storage mechanism that will capture and archive all streaming network information of interest, and (ii) a search mechanism over the archival data, potentially with the help of an indexing scheme.

This work introduces **NET-FLi** (NETwork FLow Index), a highly optimized solution for real-time indexing and data retrieval in large-scale network flow repositories. By exploiting the nature of network flow data, we introduce adaptive indexing and data compression mechanisms that exploit the synergy of bitmap indexing and locality-sensitive hashing (LSH) methodologies. Our approach offers real-time record processing, with high compression rates and interactive query response times. Both data compression and indexing are performed *on the fly*. The low response time for data retrieval from the repository is attributed to our effective indexing and selective data block decompression strategies.

Our solution can be utilized in the following applications:

1. **Network forensics**, for auditing and compliance purposes. Compressed network flow archives capture digital evidence that can be used for retrospective analysis in cases of information abuse and cyber-security attacks. Such an analysis would require deep recursive exploration of the inbound/outbound traffic through access of the stored archive. Traditional approaches typically require expensive searches and data decompressions in large data repositories. Using our framework, answers can be retrieved in mere seconds.
2. **Network troubleshooting**, as an indispensable tool for system administrators and data analysts to better support network management. Consider the case of an operator requiring a visualization of the dependencies between a set of servers; this is a necessary step for preparing a server migration plan and requires the laborious retrieval of historical traffic usage and access patterns in one's domain. *Drill-down* capabilities for querying the archive system are also needed. Our approach can provide a viable solution for fast search over the archived data, thus assisting in the expedient resolution of tasks such as: identification of network anomalies, network performance issues, and bottlenecks. Resolving such issues can lead to better load balancing of the network.
3. **Behavior analysis**, with focus on traffic classification. It is of general interest to identify the application that generated a particular network communication. Nowadays, this is a challenging problem, because a significant portion of network traffic is transported through widely used ports, such as port 80 (e.g., Skype or torrent file transfers) [30]. Recent work suggests that network flow information, such as the cardinality of flows, packets, and bytes exchanged, can be exploited for characterizing applications [44].
4. **Streaming data indexing**. Even though the solution presented here has been created to primarily support archiving of network data, the indexing and compression strategies that we present can also be used for archiving and searching over any streaming numerical data, when high throughput and low latency are essential.

Our work makes several important **contributions**:

- We present a novel solution for *on-the-fly* archiving and indexing of network flow data based on the synergy of an alphabet-optimized bitmap index variant, along with an online LSH-based reordering scheme to further boost the space savings of our approach.
- The compressed columnar approach that we propose achieves compression ratios on par with that of a typical GZIP compression, with the added benefit of providing indexing functionality as well as partial and selective archival block decompression.
- Typical data insertion rates of our approach can reach **1.2 million flow records/s** on a commodity desktop PC. High-bandwidth networks currently experience bursts of up to 50,000 flow records/s traffic, so our approach offers

an order of magnitude higher processing rates than those required to capture all flows of typical networks.
- The combination of compressed bitmap indexes and compressed data columns enables the selective decompression of the desired data and therefore guarantees **interactive retrieval times** while searching through gigabytes of compressed flow repositories.
- The architecture actively exploits the parallelism offered by multi-core and multi-processor systems.

In the remainder of the paper, we go into more detail on the distinct advantages of our solution. We begin by surveying related work and elementary concepts in Sects. 2 and 3. Section 4 presents the archival and bitmap indexing approach of our solution, while Sect. 5 discusses the online stream reordering methodology. We evaluate our approach in Sect. 6 with respect to other extant approaches. Finally, Sect. 7 provides a case study and introduces the visualization layer of our prototype solution.

2 Related work

Network forensics involves topics that need to be addressed efficiently from both a network and a database perspective. Below, we review some relevant solutions, and when possible, highlight the differences to our approach:

1. **Network traffic recording systems** [15,37] are deployed by financial and security agencies interested in keeping a sliding window of the traffic, which can be replayed entirely for postmortem analysis. Filters for supporting on-the-fly queries can be enabled, but interactive queries over the archived data are not supported.
2. **Data stream management systems** (DSMS), such as Gigascope [12], TelegraphCQ [10], Borealis [2], Tribeca [47], and System S [4,51], have been introduced to perform online queries over data streams. The idea behind the stream database approaches is to support static queries over online data without storing the entire stream on disk. Only the query results (such as data aggregations) are kept on secondary storage. Those systems usually provide SQL-like languages augmented with stream-aware operators. Plagemann et al. provide examples of stream databases deployed in the context of traffic analysis [41].
3. **Flow-based systems** for network traffic attempt to lift some of the limitations of stream-based systems by archiving entire network streams. Silk [22], nfdump [25], and Flow-tools [43] are commonly used tools for storing flows. They all store flow records as flat files, with optional data compression using tools such as gzip, bzip2, and lzop [38]. Important shortcomings of those software suites are that (i) they do not provide any indexing facility and (ii) the entire data set must be scanned linearly (and decompressed) even for retrieving only a few records. In contrast, our flow storage solution has been designed to selectively decompress only the data blocks containing the required answer set.
   Reiss et al. [42] propose a flow storage solution implemented on top of FastBit [17]. FastBit is a library for developing column-oriented databases that provides built-in support for compressed indexes for accelerating queries over historical data. Contrary to NET-FLi, their flow storage solution does not provide any compression mechanism for the data, which is appended uncompressed (and unsorted) to the disk. In addition, indexes are built offline and in batch. NET-FLi can handle twice the incoming flow rate with compression rates equivalent to that of gzip, leading to 40% smaller indexes.
   Distributed architectures for storing flow records such as MIND [33] and DIPStorage [35] assign flow records to different peers for decreasing query response times and for achieving high insertion rates. Our work is complementary, as our flow storage solution can be used as a back-end system for building distributed storage architectures.
4. **Flow aggregation databases** such as AURORA [28] and nTop [36] use stream-based approaches to analyze high-speed networks and provide statistics (e.g., IP addresses with highest usage) at different timescales (i.e., hourly, daily, weekly, or yearly). However, unlike stream databases that are programmable with SQL-like languages, network aggregation databases provide a predefined set of analysis tasks, trading flexibility for performance. In addition, they do not offer efficient drill-down capabilities, which are essential for our application.

In our solution, we adopt a columnar data representation [3,26,27]. Related to our work are therefore column-oriented database systems, such as CStore [46], MonetDB [9], or BigTable [11]. Organizing the data in columns rather than in rows offers several advantages: (i) it provides more opportunities for compression; (ii) it allows operators to be implemented directly over the compressed data; and (iii) it reduces the I/O bandwidth for queries with high selectivity on the attributes.

This work represents an expanded version of [20], providing additionally (a) an in-depth comparison with PLWAH [14], a recently introduced bitmap indexing encoding; (b) the two-sided COMPAX variant; (c) an extended evaluation of the stream reordering component; and (d) the application layer of our methodology.

Connection with compression techniques: Compression is increasingly becoming a companion technology for columnar databases [1], and high-speed compression algorithms optimized for modern hardware, such as superscalar CPUs

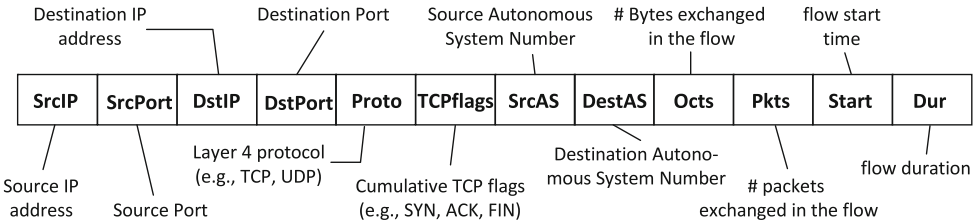

Fig. 2 Indicative attributes present in a flow record

[5,52] or graphics processing units (GPUs) [16], have been proposed in the literature. In this work, we use the fastest LZ-based compressor [38] for compressing flows, but we propose an extensible architecture where alternative compression algorithms can be plugged in.

Closely related to our work is NetStore [24], a columnar database optimized for flow records. NetStore provides built-in compression and supports multiple compression algorithms. Contrary to our work, NetStore chooses the compression algorithm to be used for each column segment (block in our terminology) at runtime and provides indexes solely for the set of IP addresses defined as local in a network. When deployed over a significantly less powerful machine (4 cores instead of 8, 2 Gb of RAM instead of 6 Gb), our storage repository provides two orders of magnitude higher insertion rates even when indexing IP addresses, transport ports, protocol, the timestamp, and the TCP flags of every flow record.

NET-FLi introduces a new compressed bitmap index variant, which we call COMPAX. Existing bitmap indexes, such as Word-Aligned-Hybrid (WAH) [49], Run-Length Huffman (RLH) [45], or Byte-Aligned Bitmap Codes (BBC) [6], have been shown to offer indexes that are many times smaller than traditional tree-based indexing data structures [48]. Our evaluations show that COMPAX is superior to the state-of-the-art index WAH in terms of compression rate, indexing throughput, and retrieval time.

Bethel et al. [8] adopted WAH-compressed bitmap indexes to reduce duty cycles of queries in the context of network traffic visualization. Their work focuses on index construction only; no mechanisms are provided to retrieve data from a flow archive to support detailed root-cause analysis. Moreover, the work does not address the challenge of compressing and storing flow records in real time.

Connection with sorting techniques: There are several studies [7,31,32,40] that propose sorting techniques to increase the average length of run-length-encoded sequences and thus the compression rate of compressed bitmap indexes. Contrary to previous works, we do not focus on finding an optimal sorting strategy. Instead, we propose a sorting strategy that can be performed *online*. This stream reordering approach leads to both reduced disk space and retrieval time. In addition, this approach can also be of interest for other bitmap index creation techniques.

3 Background

Before describing our solution in detail, we briefly revisit the concept of network flow, which is essential for our framework. The *network flow* structure has been introduced in the area of network monitoring to capture a variety of information related to network traffic. A flow is defined as a unidirectional sequence of network packets sharing source and destination IP address, source and destination ports and transport protocol. Network equipment, such as routers, provides built-in monitors (referred to as flow meters) that maintain statistics on the flows observed, such as the number of bytes, packets, or TCP flags. Once a flow is considered terminated, the statistics are exported as a flow record to a collector device.

Flow records consist of a predefined data structure used to represent the fixed-size attribute fields. Over the years, different export protocols (e.g., Netflow v5, Netflow v9, IPFIX) have been proposed. Netflow v5, the most widely used protocol, uses 48 Bytes (including three padding bytes) to encode 19 flow attributes of a single record. In Fig. 2, we depict the network flow attributes that we use in our setting, along with a simple description of the fields. By archiving flow records collected over time, a large repository documenting all end-to-end network communication patterns can be created. This repository provides a valuable source of information for many analysis tasks conducted on a daily basis by administrators, such as the examination of anomalous traffic patterns, forensic investigations, usage accounting, or traffic engineering.

4 Architecture

In this section, we describe our approach and explain our design choices. An overview of our solution is depicted in Fig. 3. Our solution comprises the following parts:

1. A data preprocessing phase reorganizes the records in the flow in order to achieve better locality and boost compression (Fig. 3b). This step is optional.
2. Separation of the stream in a columnar manner (Fig. 3c).
3. An *archiving backend* that compresses blocks of the incoming flow data (Fig. 3d).

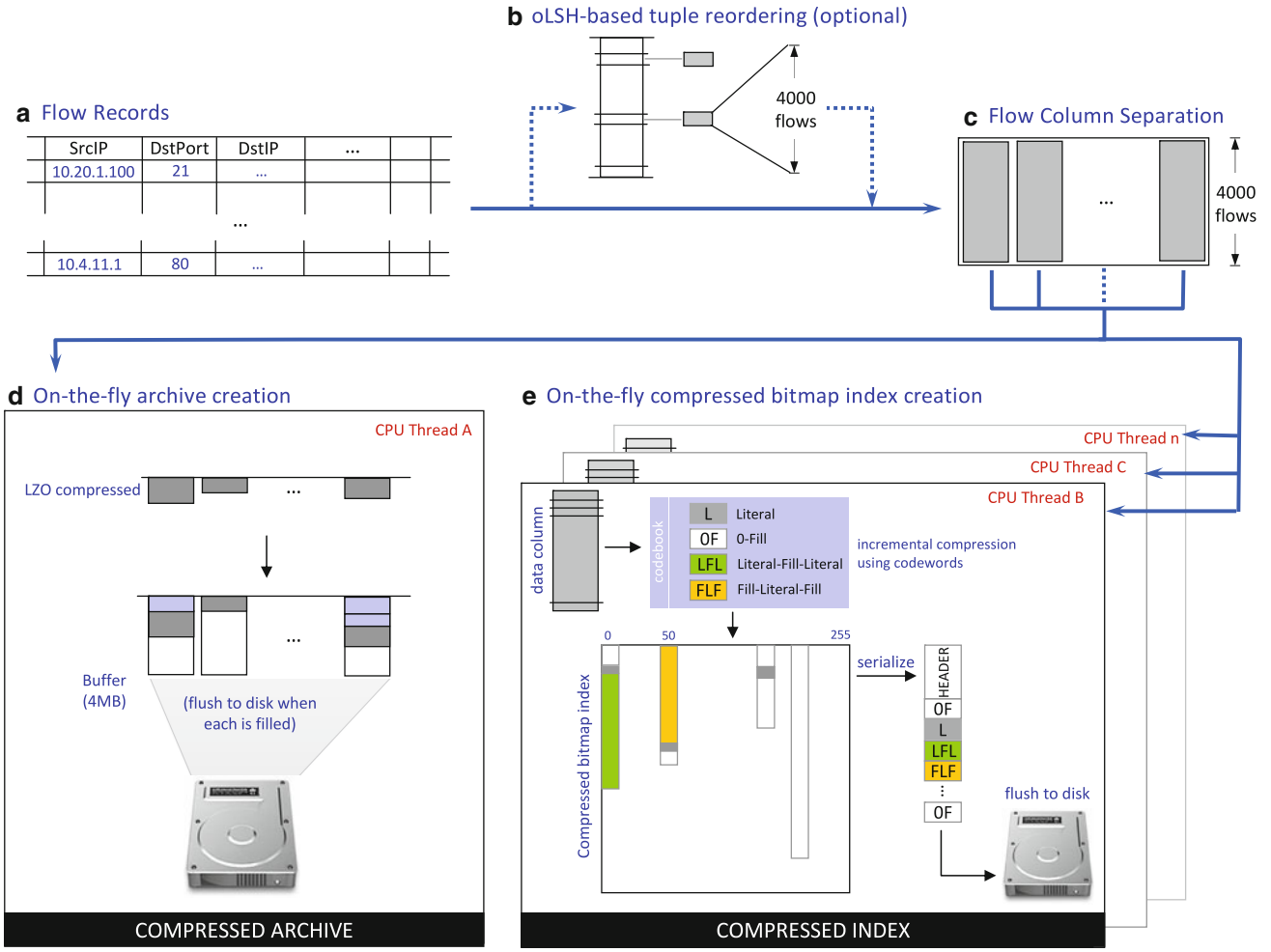

Fig. 3 NET-FLi consists of the following steps: **a** Streaming multi-attribute data are input in the system. **b** Optionally, a fast packet reordering based on locality-sensitive-hashing principles is performed to improve both compression and retrieval. **c** Attributes from the streaming records are separated. **d** The compressed columnar archives are created. **e** Compressed bitmap indexes are created on the fly, in parallel

4. A *compressed index* encoding the flow information into the novel COMPressed Adaptive indeX format (or COMPAX) (Fig. 3e).

The above tasks are executed on the fly and in parallel, which imparts the high performance to our solution. The optional online reordering step repacketizes the flows using an online implementation of a locality-sensitive hashing-based technique (online LSH or oLSH). This step results in significantly better compression ratios for both archived data and the data index. This approximate sorting is computationally light, thus supporting an on-the-fly execution over the streaming network data. Although this optional process slightly reduces the number of processed flows, it results in reduced archive and index storage and eventually leads to lower query response times.

Finally, a component outside the archival solution is the *query processor*. Given a search query and using the indexes created, the query processor identifies historical flows that match the criteria given and retrieves them by uncompressing only the relevant portions of the archive data.

Below, we provide more details on each of the components. We first cover the archival and indexing solution and later the stream reordering methodology.

4.1 Compressed archive of flow data

The input to our system consists of streaming network flow records. As mentioned, flow records can record a number of predefined communication attributes; in our system, we utilize 12 such attributes (cf. Fig. 2).

The incoming flow records are packetized and processed using a tumbling data window of M flows (Fig. 3c). Even though windowing the data reduces the effective compression rate, this allows us to provide selective decompression of archive blocks, hence significantly speeding up the query execution time.

Practical considerations at this stage are the size of the data window. Larger windows will offer a better compression rate, but least selective data retrieval during search time. In an effort to reduce cache misses, we use $M = 4{,}000$ records in our prototype implementation; this reflects the amount of processed data (12 attributes $\times$ 4,000 records) that fits into the L2 cache of the system. We also experimented with larger flow record block sizes, but noticed that changing the block size does not significantly affect the compression rate. The overall query response time of the system was best when using blocks of 4,000 records, because this provides a finer-grained access on the archived data. In addition, fewer data have to be decompressed at query time.

For all subsequent phases, the data are treated in a columnar manner, and each attribute (column) of the flow records is processed independently. Each of the blocks of conceptual data columns is compressed using a desirable compression scheme. Compression algorithms for data columns are interchangeable. We choose the Lempel-Ziv-Oberhumer (LZO) compressor [38] as our default compression algorithm, because it provides a nice compromise between compression speed and compression size. In particular, it has been empirically demonstrated that LZO is four to five times faster in decompression than *zlib* [21], even when using zlib at the fastest compression level [39].

To be able to exploit sequential writes to the disk, the compressed blocks of streaming data are not appended to the data archive as soon as they are created. Instead, the compressed columnar blocks are initially buffered and only flushed to disk when the buffer is full[1] (Fig. 3d).

4.2 COMPressed Adaptive IndeX: COMPAX

Concurrently with the creation of the compressed flow data archive, a compressed bitmap index is constructed, which facilitates a rapid location of relevant portions in the archived data during the querying process. We call this new bitmap index "COMPressed Adaptive indeX" or COMPAX. It is built using a codebook of words that significantly reduce the bitmap index size. The entire process is performed on the fly. We begin by elucidating the creation and usefulness of traditional bitmap indexes.

4.2.1 Bitmap indexes

The concept of *bitmap indexing* has been applied with great success in the areas of databases [17] and information retrieval [19]. The basic notion is to keep k bitmap vectors (columns), one for every possible value that an attribute can assume (k refers to the attribute cardinality), and to update them at every insertion by appending a "1" to the bitmap

[1] In our implementation, we allocate 4 MB for each column buffer.

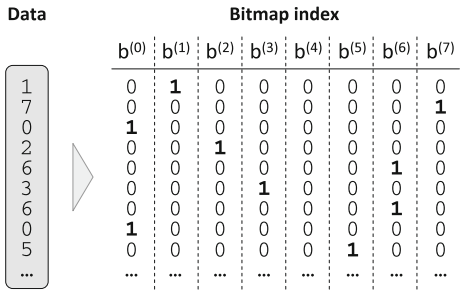

Fig. 4 An example of a bitmap index

i corresponding to the inserted value and "0" otherwise. An example of a bitmap index with $k = 8$ is illustrated in Fig. 4.

In addition to fast updates, bitmap indexes allow inexpensive bitwise AND/OR operations between columns. For example, given the bitmap index of Fig. 4, to retrieve the row numbers where the record was either 5 or 7, it is sufficient to perform the bitwise *OR* between columns 5 and 7. The desirable rows are indicated by the positions where the resulting bitmap is 1.

Another key property of bitmap indexes is that subsequent insertions do not require reorganization of the existing index, as new bit values can be appended easily to the end of the index.

Compressed variants of bitmap indexes have appeared in the literature [48], with the most popular variant being the World-Aligned-Hybrid (WAH) [49], which has been used for indexing and searching on a multitude of data sets, including scientific simulations results and network flow repositories. WAH uses run-length-encoding principles to compress homogeneous sequences of sets of the same symbol. Compression is not only beneficial for reducing space but also for improving the performance of boolean operations. Logical bitmap operations can be executed directly on compressed bitmaps by using run-length-encoded sequences to decrease the number of bitwise comparisons. More recently, the Position List Word-Aligned-Hybrid (PLWAH) [14] has been introduced. It represents a variant of WAH offering better compression performance for uniformly distributed data. In the experimental section, we compare COMPAX with both WAH and PLWAH.

4.2.2 COMPAX encoding

We compress each bitmap index in a column manner *on the fly* using our *COMPressed Adaptive indeX* (COMPAX). There are two variants of COMPAX: one-sided (or COMPAX) and two-sided (or COMPAX2). We adapt this terminology from [50], in which BBC bitmap index encoding offers a two-sided variant that compresses sequences of both zeros and ones, whereas the one-sided variant focuses on sequences of both zeros only. COMPAX2 offers an extended codebook,

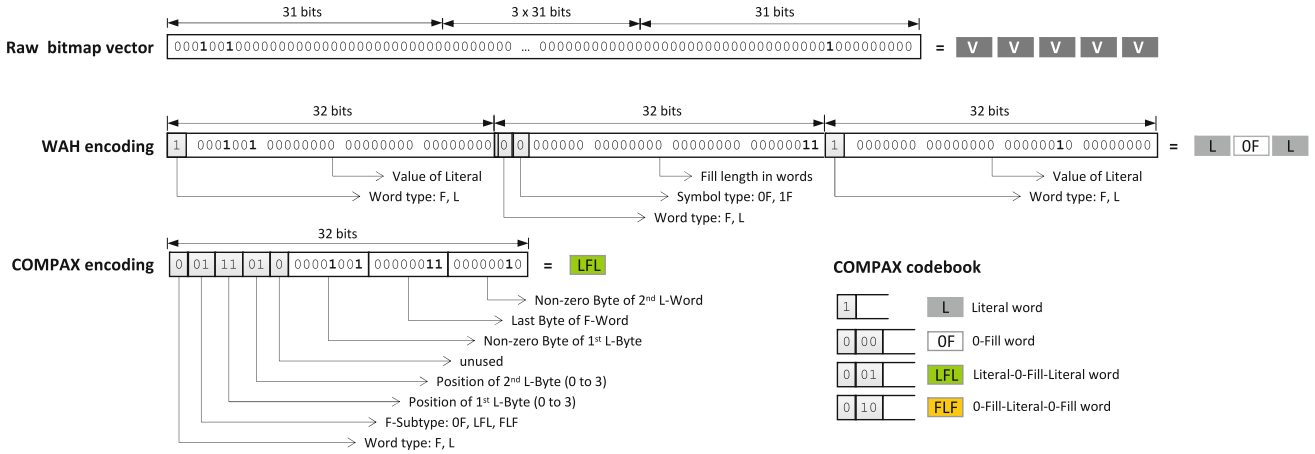

Fig. 5 Example encoding of a raw bitmap vector (*top*) corresponding to a single bitmap index column (as in Fig. 4) using the WAH (*middle*) and the one-sided COMPAX method (*bottom*). The one-sided COMPAX codebook is shown at the *bottom-right*

which allows better compression but introduces more complex bitwise operations. More details will be provided below.

COMPAX uses a codebook of four word types and compresses sequences of zeros with run-length encoding. The following five 32-bit wide word types are used to encode the incoming bit stream processed in *chunks* of 31 bits:

1. A Literal **[L]** represents a 31-bit-long non-zero sequence of bits; it is encoded as a 32-bit word having the first bit as one (1), e.g.,

1	0011100 11000010 00110000 00000000

2. A 0-Fill **[0F]** encodes a sequence of consecutive chunks of zero bits only by means of run-length encoding.
   For example, a portion of a bitmap index column consisting of a sequence of 3×31 zero bits is encoded within a single 32-bit word as

000	00000 00000000 00000000 00000011

   where the first 3 bits encode the codeword type (0-Fill) and the remaining 29 payload bits encode the number of 31-bit chunks from the original sequence.
3. An **[LFL]** word condenses sequences of [L]-[0F]-[L] words after applying null suppression. In particular, if in each of the two literal words in the sequence, only one of the payload bytes is non-zero ("dirty"), and if the 0F represents a sequence shorter than 256×31 bits, the 3 dirty bytes are packed into a single [LFL] word. An example of how the [LFL] codeword is formed is shown in Fig. 6. The 8-bit header encodes the word type (3 bits) and the positions from 0 to 3 of the literals' dirty bytes (4 bits in total). The last bit is unused.
   A real example of [LFL] encoding is depicted in Fig. 5. The raw uncompressed bitmap vector depicted on top consists of five 31 bits chunks, where only the first and the last one contain some non-zero bits. As mentioned, they can be represented as literals [L]. As for both literals, the non-zero bits are present in only one of the 4 bytes, they exactly have one "dirty" byte (in position 3, and 1 for the first and the last, respectively). The three remaining chunks do not contain any non-zero bit and can be encoded with a single [0F] word, where the payload encodes the number of 31-bit chunks (3 in the example, which can be represented as a single byte). Therefore, the technique depicted in Fig. 6 can be applied to encode the two literals and the FILL as a single [LFL] word. Its one-byte header specifies the word type and, in addition, encodes the positions (3 and 1, respectively).
4. The **[FLF]** word condenses a sequence of words following a [0F]-[L]-[0F] paradigm and having a single non-zero byte. As fill words are representing sequences shorter than 256×31 bits, the header includes the position of the literal non-zero byte only (2 bits).

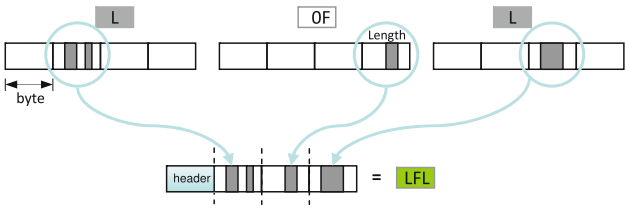

Fig. 6 Example of bit packetization in the [LFL] codeword

COMPAX2 extends the COMPAX codebook with a 1-Fill (1F) word that compresses sequences of ones. In addition, the FLF and LFL headers include bits specifying the fill type (0F or 1F):

1. The 1-Fill **[1F]** encodes a sequence of consecutive chunks of zero bits only by means of run-length encoding.
   For example, a portion of a bitmap index column consisting of a sequence of 3×31 one bits is encoded within a single 32-bit word as

011	00000 00000000 00000000 00000011

2. The [**LFL**] word is built as in COMPAX. However, both sequences of [L]-[0F]-[L] and [L]-[1F]-[L] can be compressed. The word carries an 8-bit header and the three payload bytes. The first three bits of the header encode the word type (001 for LFL), whereas the remaining 5 bits encode the two positions (2 bits each) and the fill type (1 bit representing 1F or 0F). For example, a sequence such as the one in Fig. 6, but with a 1F word instead of a 0F, is encoded as

001	01101	00000000 00000000 00000011

3. The [**FLF**] compresses sequences following the [F]-[L]-[F] pattern. Contrary to COMPAX, fill words can be both 0F or 1F. The 8-bit header encodes the word type (3 bits), the two fill types (2 bits) and the position of the non-zero byte (2 bits) within the literal. The last bit of the header is unused.

010	11000	00000000 00000000 00000011

4.2.3 *COMPAX compared with other bitmap indexing schemes*

Here, we briefly outline the main differences between COMPAX and the state-of-the-art bitmap indexing approaches.

Comparison with WAH: The WAH encoding uses three word types to encode bitmaps: Literal, 0-Fill, and 1-Fill. In our work, we introduce the [LFL] and [FLF] codewords because we observed that such patterns were predominant in network flow traffic. The one-sided version of COMPAX omits the [1-Fill] word from the codebook, because such patterns are quite uncommon. In addition, it provides a simpler implementation of bitwise operations because of the reduced codebook size. However, in both cases, our coding scheme leads to significant space savings. In the experimental section, we show that compared with WAH, even the simplest COMPAX encoding can result in a space reduction of more than 60%, particularly when combined with the optional tuple reordering phase (described in detail later on).

Figure 5 depicts the difference of the one-sided COMPAX encoding compared with WAH. The bits of a column are shown row-wise for presentation reason, and the original uncompressed sequence comprises 155 bits (indicated as 5 verbatim 31-bit words [V]). In this example, we illustrate COMPAX's ability to condense three WAH words into one. In general, the COMPAX encoding packs more information because of the carefully selected codebook and as such offers superior compression.

Comparison with PLWAH: Similar to COMPAX, the PLWAH encoding scheme attempts to achieve better compression rates than WAH in the presence of very sparse bitmaps. In the original paper [14], the authors of PLWAH show empirically and analytically the improved compression rates when encoding data that follow a uniform distribution. PLWAH boosts the compression rates by exploiting the fact that FILL words do not usually represent long sequences and literals themselves contain only few bits set to 1. Therefore, by storing the positions of those non-zero bits into the preceding FILL word, it is possible to pack a FILL word and the following sparse literal into a single codeword. When the word size is 32 bits, each position can be encoded with 5 bits (0 to 30). The number of positions that can be stored in the preceding FILL is a tunable parameter, and, in fact, increasing the number of positions kept corresponds to decreasing the maximum length of the sequence of homogeneous symbols encoded by the FILL word. By keeping a single position, the longest sequence of homogeneous symbols that PLWAH can represent is $(2^{25} - 1) \times 31$ instead of $(2^{30} - 1) \times 31$ as in WAH. In the example of Fig. 5, a PLWAH implementation that keeps a single position can encode the row bitmap vector in two words instead of three as WAH does. In fact, the last literal containing a single bit set to 1 and the preceding FILL word can be merged by storing the position (9 in the example of Fig. 5 when counting from the least significant bit) as a 5-bit number just after the two bits header of the FILL word.

In the experimental section, we show that the COMPAX encoding results in equivalent or, in many cases, better compression rates than PLWAH. More importantly, when coupled with the proposed tuple reordering phase (oLSH), COMPAX always outperforms PLWAH in terms of the index compression rate.

4.3 On-the-fly bitmap index creation

The COMPAX-encoded bitmap indexes are created on the fly to minimize the space consumption, without having any adverse impact on the index size (see also Fig. 3e).

For example, assume that already a 0-Fill codeword has been created on some column, encoding 3×31 bits of zeros, as in the previous example. Now, suppose that, for the same column, one more chunk of 31 zeros is observed, and then, another 0-Fill word of length 1 can be created. By looking at the previously produced codeword, we are able to merge them and produce a single 0-Fill word of length 4, which in binary format will be encoded as

000 00000 00000000 00000000 00000100

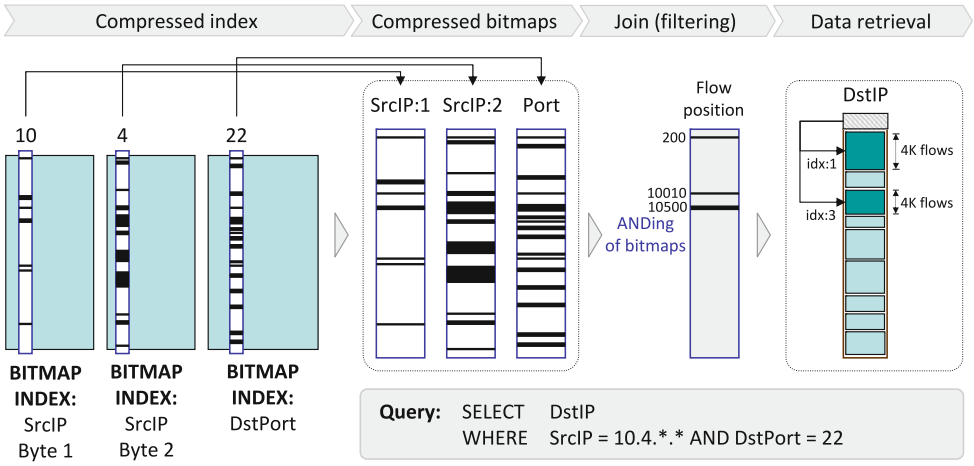

Fig. 7 Example of a query execution

In a similar manner, we maintain the codewords per column on the fly. We distinguish the following cases:

- [L]: Unlike the case for 0-Fill words, when a new 31-bit literal chunk is observed, it cannot be merged with a previously produced literal, so another literal [L] word is created.
- [LFL]: In order to form an [LFL] codeword, a look-back examines the two previously produced codewords. If they are [L] and [F] and the current word is a literal [L] and all three codewords have only one dirty byte (the remaining bytes are all zero), then these three codewords can be merged into a single [LFL] word.
- [FLF]: Treated similarly as [LFL].

The advantage of encoding the bitmaps on the fly is the memory consumption, which, during the entire creation phase, is almost the same as the compressed bitmap index size. An example of this encoding is shown at the bottom right of Fig. 3.

Bitmap index serialization: The COMPAX-encoded bitmap indexes are serialized to disk by sequentially appending the compressed columns. In addition, each index is prepended with a header that contains offsets corresponding to the beginning of each compressed column. In this way, random access to specific compressed column within a bitmap index is accommodated. Headers are also compressed. In fact, offsets are compressed using gap coding and the Simple9 algorithm [5], which is among the fastest available integer compression/decompressions schemes.

In our implementation, we create bitmap indexes for the most commonly queried attributes, such as *source and destination IP addresses*, *source and destination ports*, *protocol*, *tcpflags*, *duration* and *start time*. A particular indexing strategy is taken for IPv4 addresses: a separate index is maintained

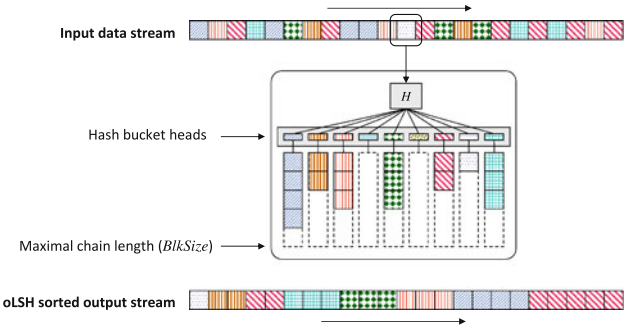

Fig. 8 Mechanism of the online-LSH (oLSH) for approximate sorting on a data stream

for each 8-bit block of a 32-bit IP address.[2] In this way, we accelerate wild card queries over networks (e.g., 10.1.*.*) by combining compressed bitmaps belonging to different byte indexes using boolean AND operations.

4.4 Querying the system

The proposed architecture can very efficiently answer common types of queries posed by system administrators and security analysts. The system users can use both equality and range queries on attributes contained in the index, such as source and destination IP addresses, ports, protocols, and the time-span of interest. The query execution consists of the following steps, which are also captured in Fig. 7:

1. Columns/attributes involved in the query are determined. Relevant columns from the appropriate compressed bitmap indexes are retrieved.
2. Boolean operations among compressed columns are performed directly *without explicit decompression*. Flow record positions on the compressed archive are resolved.

[2] The generalization to IPv6 addresses is straightforward.

3. The appropriate portions of the archive (relevant compressed data blocks) are decompressed, and the results are supplied to the user.

We will explain the above process with an example. Consider the case of an worm attack. A sample query that the system administrator can issue to discover all systems contacted for a set of compromised nodes is as follows:

> **Query:** "Find all destination IP addresses contacted by the source IP address range 10.4.*.* at destination port 22."

The various substeps are depicted in Fig. 7. As bitmap index files are created on an hourly basis, first the bitmap index within the time range of interest in the query is retrieved. The bitmap indices for *SrcIP.byte1*, *SrcIP.byte2* and *DstPort* are retrieved, and from those, the relevant *compressed* columns 10, 4, and 22, respectively, are fetched. Note that we do not need to uncompress the columns; an AND operation can be performed directly on the three compressed columns, as one does not need to AND the portions containing the 0-Fill codewords.

Suppose the join result of the three compressed columns indicates that there are matches at flow row numbers {200, 10010, 10500}. Because the user wants to retrieve the destination IP addresses, the query processor will need to uncompress the relevant blocks in the archive of column *DstIP*. Assume that each of the compressed blocks on the archive contains 4,000 flow records. Therefore, to access the 200th, 10010th, and 10500th flow record, we need to retrieve only the first and third compressed blocks in the archive. The start position of those blocks is provided in the header of the archive. Finally, the result set of the three destination IP addresses, contacted by the specified range of source IPs in the query, is returned to the user.

5 Online-LSH (oLSH) stream reordering

Here, we introduce an online stream reordering mechanism that is based on the principle of locality-sensitive hashing (LSH) [23,34]. We call this variant online-LSH or oLSH for short. It implements an intelligent buffering mechanism with the purpose of packing together similar records from a data stream. It has been shown that data sorting leads to smaller and faster bitmap indexes [40,32]. FastBit [17], the reference implementation of the WAH algorithm, accommodates an optional off-line column sorting to decrease index sizes. In this work, we rely on sorting to decrease the disk consumption of both indexes *and* data archives. Our approach is consonant with the *permuting*, *partitioning*, and *compression* (PPC) principle [18]: after permuting the order of incoming stream records, the records are partitioned into groups of similar records, with the goal of boosting the compression ratio of general-purpose compressors.

The characteristics and benefits of oLSH are as follows:

- It reorders the incoming data records in a fast and effective way, resulting in data blocks with lower entropy.
- It improves the compression rate, leading to smaller bitmap index and archive sizes.
- By placing similar records in close-by positions, it eventually leads to better query response times, because fewer data blocks need to be decompressed from the data archive.

The stream reordering is implemented by a hash-based buffer, which uses several LSH-based functions to group flow records by content. In fact, the basic premise of LSH is to use hash functions to direct vectors that are close according to a distance function into the same hash bucket with high probability. The sorted output stream is then built by constantly flushing similar records from the hash table (cf. Fig. 8).

We consider each flow record as a vector $\mathbf{r} \in \mathbf{N}^d$, where d is the number of the attributes relevant to the sorting process. The purpose of the LSH functions is to aggregate "similar" flows to the same hash bucket. Each hash bucket eventually contains a chain of "similar" records. We employ n LSH functions based on p-stable distributions, such as the ones proposed by Datar et al. [13]. Each LSH hash function $h_{\mathbf{a},b} : \mathbf{N}^d \rightarrow \mathbf{Z}$ maps a vector $\mathbf{r}$ into a single value ("bin") and is defined as

$$h_{\mathbf{a},b}(\mathbf{r}) = \left\lfloor \frac{\mathbf{a}^T\mathbf{r} + b}{W} \right\rfloor \tag{1}$$

where $\mathbf{a}$ is a d-dimensional random vector with each component chosen independently from a Gaussian distribution,[3] W is the width of a bin, and b is a real number chosen uniformly from the range $[0, W]$.

In our scenario, we wish to pack together flow records based on the attributes queried most often. Therefore, we chose $\mathbf{r}$ to be an 11-dimensional vector ($d = 11$) consisting of the attributes source and destination IP addresses (2×4 bytes, i.e., 8 dimensions) as well as source and destination ports and protocol numbers (3 dimensions).

We reduce the probability of collisions of unrelated flows by computing the hashtable index value as a sum of many LSH functions. In detail, $H_1(\mathbf{r}) = \sum_{i=1}^{n} h_{\mathbf{a}_i,b_i}(\mathbf{r}) \mod P$ from the n mappings of $\mathbf{r}$, where P is the hash size. In addition, as collisions of unrelated records may still occur within each hash chain, chains are kept ordered using an InsertionSort algorithm. The key used for sorting in the InsertionSort

[3] The Gaussian distribution is 2-stable; it can be shown that elements, which are close in the Euclidean distance sense will be mapped to the same value with high probability, and to distinct values otherwise [13].

Program 1 oLSH reordering of the streaming records

```
processElement(hashtable hash, flow record r){
   P = hash.length(); // size of hashtable

   h1 = sum( h(a[i], b[i], r) ) mod P; // hashtable index
   h2 = sum( h(a'[i], b'[i], r) ) mod Q; // used for
                                         // InsertionSort

   chain = hash[h1].InsertionSort(r, h2);
   if (chain.length() > maxBlockSize)
     emitBlock(chain, archive, index); // send to archive
                                       // and index
   maxCount = hash.totalNumBuckets();
   if (maxCount > MMax){ // memoryBudget
     do{
       chain = longest_chain(hash);
       emitBlock(chain, archive, index);
     } while(hash.totalNumBuckets() > MMin); // minimum
   }                                         // threshold
}
```

Fig. 9 Comparison of positions of a query without (*left*) and with (*right*) LSH reordering

is computed using a different combination of LSH functions H_2 that utilizes different projection spaces:

$$H_2(\mathbf{r}) = \sum_{i=1}^{n} h_{\mathbf{a}_i', b_i'}(\mathbf{r}) \mod Q$$

The stream reordering process consists of inserting incoming flow records into the hash and dispatching new blocks to the compression and indexing components. Whenever the length of a chain reaches a configurable maximum threshold (*maxBlockSize*), the chain is removed from the hash and its content used to fill a block (or several blocks in case of collisions). We also employ two thresholds, *MMax* and *MMin*, to limit the number of records to be buffered (i.e., the memory budget). When the total number of flows stored by the hash reaches *MMax*, blocks are created by packing (and purging) the longest chains. The process stops when *cnt* reaches a value lower than *MMin*. A pseudocode of the online reordering process is shown in Program 1.

Figure 9 illustrates the benefits of online record reordering. We compare the row positions (matching records) returned by an IP query lookup when executed over two NET-FLi flow repositories storing exactly the same traffic and built without and with oLSH reordering enabled. In the unsorted version, matching positions are spread all over the column, whereas in the version using the online-LSH matching, records are concentrated in the first half. Data retrieval from compressed columns benefits from this because fewer blocks from the archive must be accessed and decompressed. In addition, as the blocks store sets of homogeneous records, the average block decompression times are substantially reduced and therefore, as we show in the evaluation section, faster decompression speeds can be achieved.

Finally, Fig. 10 displays an instance of the actual bitmap index capturing the first byte of the Source IP field (256 columns) with and without the reordering component. The color coding on the bitmap index indicates which COMPAX codeword is used (0-Fill is shown in white). At the bottom of each bitmap index, we show the number of codewords for each column. It is easy to see that the oLSH reordering results in a significantly lower number of codewords used for encoding the same amount of information.

6 Evaluation

In this section, we evaluate the implementation of our approach. We investigate critical performance metrics of the archival and data retrieval process. In addition, we compare it with other prevalent bitmap indexing approaches.

We use two real data sets in the evaluation:

- Six days of NetFlow traces of access traffic from a large *hosting environment* (HE).
- Two months of NetFlow traces of internal and external traffic in an average-sized enterprise *production network* (PN).

Each data set is stored as a set of flat files, each corresponding to an hour of traffic. Details of the two data sets are listed in Table 1. The traffic of the two networks differs significantly in terms of the distribution of IP addresses and service ports. The flow attributes included in the index and archived data columns are presented in Table 2.

We have implemented the indexing, storage, and querying techniques as a C++ library of 25,000 lines of code. The library comes with three similarly tuned implementations of WAH, PLWAH and COMPAX compressed bitmap indexing algorithms. In all cases, boolean operations do not require any explicit bitmap decompression. For portability reasons,

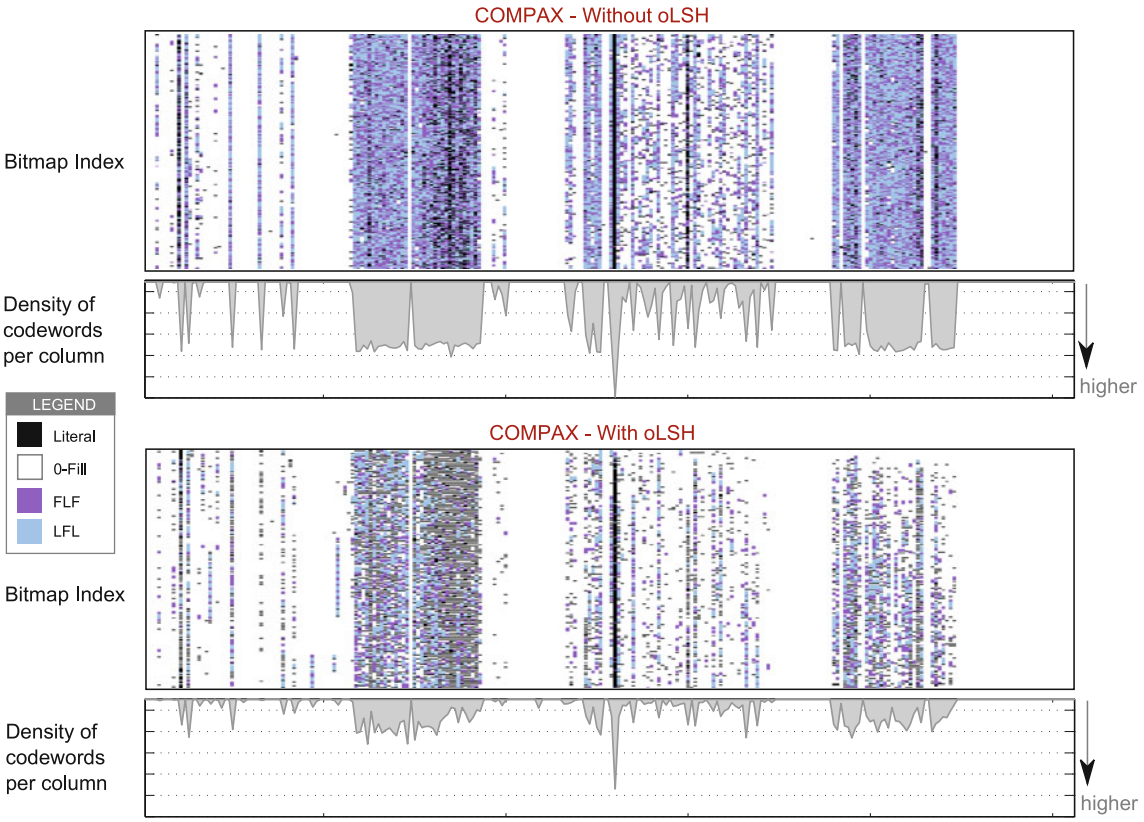

Fig. 10 Visualizing the bitmap index with and without the oLSH component. This bitmap index captures the first byte of the Source IP field (256 columns). The different COMPAX codewords indicated in different colors

Table 1 Data sets used

Data set	# Flows	Length	Size
Hosting environment (HE)	231.9 million	6 days	6.9 GB
Production network (PN)	1.2 billion	62 days	37 GB

Table 2 Flow attributes present in the index and archived data columns

Attribute	Size	Index	Archive
Source IP address	4 bytes	✓	✓
Destination IP address	4 bytes	✓	✓
Layer 4 protocol	2 bytes	✓	✓
TCP/UDP source port	2 bytes	✓	✓
TCP/UDP destination port	2 bytes	✓	✓
Number of packets	4 bytes	–	✓
Number of bytes	4 bytes	–	✓
First time stamp	4 bytes	✓	✓
Duration	4 bytes	–	✓
TCP flags	1 byte	✓	✓
Source AS number	2 bytes	–	✓
Destination AS number	2 bytes	–	✓

we decide not to exploit any specific instruction set, such as the SSE4.2 [29], which provides the *POPCNT* instruction for counting the number of bits set to 1 in an unsigned word.[4] Instead, we rely on precomputed lookup tables for implementing bit counting.

The software does not require any external library except for *lzo* [38], which provides us the LZO1X-1 [39] algorithm implementation that we use for compressing columns. LZO1X-1 does not offer the best compression rate, but instead is designed for achieving high compression and decompression speeds.

All experiments have been executed on a commodity desktop machine equipped with 2 GB of DDR3 memory and an Intel Core 2 Quad processor (Q9400) running GNU/Linux (2.6.28 kernel) in 32-bit mode. The processor has four cores running at 2.66 GHz and 6 MB of L2 cache. We store flows on a 320-GB desktop hard drive[5] formatted with a single Ext3 partition.

[4] The GNU compiler (GCC) provides the built-in function *built-in_ popcount* for exploiting such an instruction, if present.

[5] The hard drive is a 7200 rpms Hitachi HDP725032GLA380 equipped with 8 MB of cache. The system is capable of performing cached reading at 2400 MB/s and unbuffered disk reads at 80 MB/s (measured with `hdparm`).

Table 3 Disk space requirements when compressing the two data sets with general purpose compressors (gzip and lzop). We report the disk consumption when attribute columns are compressed independently and when the compression is done on the entire flow trace

		Row-wise		Column-wise	
Data set	Raw	GZIP	LZOP	GZIP	LZOP
HE	6.9 GB	2.5 GB	3.5 GB	2.5 GB	3.6 GB
PN	37 GB	8.1 GB	13.2 GB	8.8 GB	13.4 GB

Table 4 Disk space requirements when compressing flow data with NET-FLi with and without oLSH component enabled

		NET-FLi archive	
Data set	Raw	LZO	LZO+oLSH
HE	6.9 GB	3.7 GB	2.6 GB
PN	37 GB	13.8 GB	8.2 GB

6.1 Disk utilization

We measure the disk consumption of our methodology for both the archive and the compressed bitmap indexes.

Archive size: We compare the disk space requirements when storing the flow data using different storage approaches:

- The flow data are stored as uncompressed flat files.
- The flow data files are compressed with popular compression utilities: *gzip* and *lzop*. This represents the typical scenario for flow archive systems [25].
- The flow data files are split attribute by attribute. Each attribute column is independently compressed using *gzip* and *lzop*.
- Our proposed archival method that relies on a compressed columnar approach with small compression blocks. We evaluate this approach with and without oLSH reordering.

In Table 3, we report the disk consumption when flow data traces are compressed using the general-purpose compression utilities *gzip* and *lzop*. For both utilities, we measure the disk consumption when compressing entire flow files (row-wise) and when columns corresponding to different attributes within a flow file are compressed independently (column-wise).

When compressing the flow records, the disk space is lower when using $gzip$ compression than when using $lzop$. This is because the lzo algorithm implemented by the $lzop$ utility is optimized for speed rather than compression efficiency. However, this is exactly the reason for adapting lzo in a streaming setting, where flow records have to be compressed on the fly. When compressing each attribute column independently, the disk consumption slightly increases.

Table 5 Comparison of index sizes built using different bitmap encodings without the oLSH component enabled

Data set	WAH	PLWAH	COMPAX	COMPAX2
HE	8.1 GB	5.1 GB	4.9 GB	4.9 GB
PN	26.3 GB	17.4 GB	18.6 GB	18.0 GB

Table 6 Comparison of index sizes built using different bitmap encodings with the oLSH component enabled

Data set	WAH oLSH	PLWAH oLSH	COMPAX oLSH	COMPAX2 oLSH
HE	4.8 GB	3.4 GB	3.3 GB	3.2 GB
PN	15.1 GB	11.7 GB	12.8 GB	11.6 GB

In Table 4, we report the disk consumption when NET-FLi stores the data sets with and without enabling oLSH reordering of flows.

We observe that the column-oriented NET-FLi archive, without the oLSH component enabled, is up to 6% bigger than the flat files compressed with *lzo*. This marginal space overhead is attributed to the fact that we compress data in small blocks (up to 4,000 records each). This leads to a reduction in the compression rate. In addition, our approach requires some additional space for keeping the required header at the beginning of every block. However, the above additions in our approach allow the selective decompression of any block in the archive.

The most interesting results of these measurements are the disk savings introduced by the oLSH component. In fact, the on-the-fly reordering allows the disk consumption of our storage architecture to be reduced by as much as 40%. Therefore, the combination of the oLSH and lzo leads to similar compression rates as $gzip$ when applied to the flat files. Thus, our storage architecture consumes as much space as standard flow-based storage solutions, but can leverage the substantially higher decompression speed provided by lzo during data retrieval operations. As we show later in the evaluation, the oLSH component further boosts the decompression speed.

Index Size: We now focus on the disk consumption of the indexes built using different compressed bitmap indexing technologies. We compare the disk requirements when indexing the seven (most queried) flow attributes reported in Table 2 using our encodings COMPAX and COMPAX2 and the existing WAH and PLWAH.

Table 5 reports the disk consumption used by the WAH, PLWAH, COMPAX and COMPAX2 indexes without oLSH-based sorting. Compared with WAH, COMPAX and COMPAX2 indexes are up to 40% smaller, whereas PLWAH offers comparable compression rates.

As shown in Table 6, enabling the oLSH component allows the index size to be further reduced not only for

COMPAX but also for other bitmap indexing techniques, making our contribution of independent interest. For example, WAH indexes are more than 42% smaller when oLSH is enabled. COMPAX2 is the bitmap indexing technology providing the lowest disk consumption (up to 30% lower than WAH), but the gain is only marginal compared with COMPAX and PLWAH, which provide similar compression rates when oLSH is enabled.

Visualizing the oLSH improvements: One can also communicate the improvements attributed to the oLSH record reordering process visually. In Fig. 11, we plot the bitmap index as encoded by COMPAX and COMPAX with oLSH by depicting an *uncompressed* version of the bitmap. The various codewords are indicated with different colors provided in the legend. We display the bitmap indexes for a number of attributes, namely, all four bytes of source IPs and destination IPs, source and destination ports (only partially), and protocol.

As an example, the bitmap index on the top-left-hand side of the figure captures the first byte of the source IP field of the network flow records. There are 256 columns on this bitmap, one for each of the 256 values possible. Each of the bitmap indices displayed reflects the relevant field for a traffic duration of one hour. Next to the bitmap encoded using COMPAX, we position the resulting bitmap index when oLSH is enabled. On the far-right-hand side, we display the size ratio of the oLSH-encoded bitmap index versus that of the unsorted traffic.

The visual comparisons cogently show the power of the oLSH process in producing compact bitmap indexes.

6.2 Stream record processing rates: archive and index

NET-FLi has been designed to handle high-speed streams of flow records, so in this section, we measure the average sustainable insertion rate, expressed in flows per second (f/s). We test our storage solution by feeding it with uncompressed flow traces stored on a mainstream solid-state drive.[6] The drive provides a sustained reading speed of 170 MB/s corresponding to more than 5 million f/s. The indexing software has been configured to fetch flows sequentially from the solid-state drive and to store indexes and compressed columns to the mechanical hard drive. This simple setup allows us to reproduce flow rates that can only be observed in very large ISP networks.

We measure the insertion rate of our solution with and without enabling oLSH flow reordering and compare it with the insertion rates of WAH and PLWAH-based indexes. In Table 7, we report record processing rates for building **both the index and the archive**. We compare three variants: 1) indexes encoded with WAH, 2) with PLWAH, 3) with COMPAX , and 4) with COMPAX and additional oLSH reordering.

[6] Intel X-25M G1, 80 GB model.

Table 7 Record processing rates of our system when building both the index and the archive. We compare building the index using WAH, COMPAX and COMPAX+oLSH

Data set	WAH	PLWAH	COMPAX	COMPAXoLSH
HE	768K f/s	924K f/s	936K f/s	474K f/s
PN	1150K f/s	1220K f/s	1255K f/s	513K f/s

Table 8 Disk consumption (in KB) of WAH, PLWAH, and COMPAX when indexing IP addresses and ports with a uniform distribution

Field	WAH	COMPAX	PLWAH
Source IP address	52296	**21046**	29145
Destination IP address	52285	**21049**	29144
Protocol	435	449	396
TCP/UDP source port	14941	13446	**7602**
TCP/UDP destination port	14941	13444	**7602**
First time stamp	2247	1958	1870

Without oLSH reordering enabled, COMPAX offers insertion rates that are higher than WAH and comparable to PLWAH. The advantages in terms of insertion rate are not only related to the lower disk consumption. In fact, we can measure similar differences (in percentage) when index serialization is disabled. By using a profiler, we realized that COMPAX is much more cache-friendly than WAH. Indeed, COMPAX can perform its online compression step that can pack three WAH words into a single word (`FLF` or `LFL`) without requiring many additional L2 cache misses on average. The test is confirmed by the higher insertion rate on the data set HE, which, because of its entropy in many attribute fields, taxes the CPU cache more than the PN data set. Similar results can be observed for PLWAH, which offers similar indexing rates as COMPAX.

Without enabling oLSH reordering, our storage solution can handle as many as 1.2 million f/s. To put these numbers into perspective, medium–large networks exhibit peak rates of 50,000 flows/sec or more. When the oLSH component is enabled, the insertion rate is reduced to half. The drop is substantial, but this eventually results in a disk space reduction of as much as 55% and in a significant improvement in response time (details in the following section). Nonetheless, it is worth considering that 500K f/s is still *more than double* the maximum insertion rate reported in [42], where no data compression was performed.

6.3 Extended comparison with PLWAH

As discussed in Section 4, PLWAH and the COMPAX have been introduced for achieving better compression rates than WAH in the case of sparse bitmaps. On real traffic traces, the

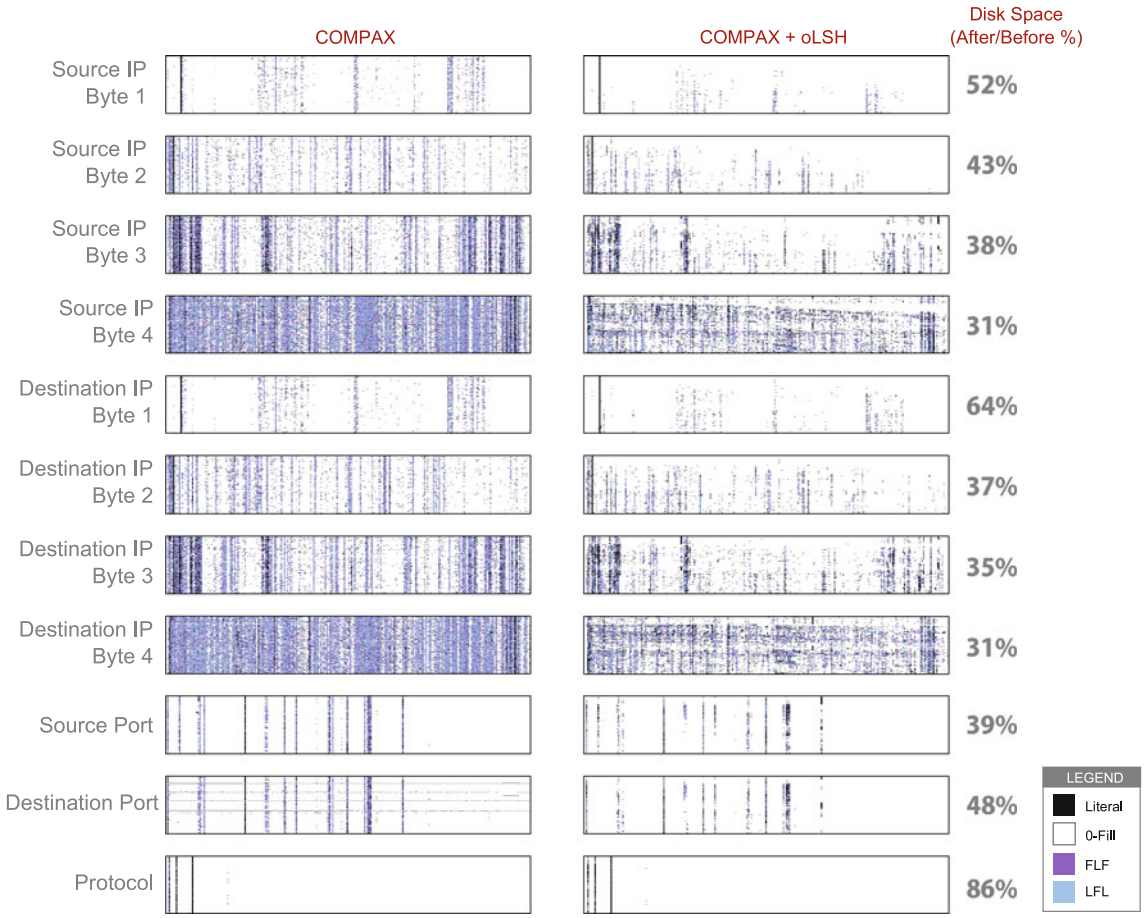

Fig. 11 Visual representation of the savings induced by the oLSH component. We display the bitmap index with and without the online reordering

two techniques provide equivalent compression rates. Here, we offer a more extended comparison with PLWAH under the more challenging condition of uniformly distributed data. Even though such traffic patterns are quite uncommon during regular traffic conditions, anomalous events such as these caused by port scans and worms may lead to distributions that are close to uniform. It is worth noting that providing higher compression rates under such circumstances is desirable because it renders the storage architecture more resilient to attacks.

To evaluate the different encoding techniques under this scenario, starting from a real flow trace of 2 million records, we synthetically create a new flow trace in which IP addresses and ports are uniformly distributed. We index the synthetically generated trace using WAH, COMPAX and PLWAH. In Table 8, we report the disk consumption, expressed in KB, when indexing some of the most queried fields.

When indexing IP addresses, COMPAX offers significantly smaller bitmap index size than WAH and PLWAH. In particular, COMPAX-encoded indexes are just 40% of the size of WAH-encoded indexes, whereas PLWAH-encoded indexes are 56%.

On the other hand, when indexing uniformly distributed Port numbers, PLWAH offers the best compression rate. This is the case, because the FLF and LFL codewords introduced by COMPAX cannot be applied frequently, as the length of the fill is often longer than 256.

To better illustrate why PLWAH provides better compression rates than COMPAX when indexing the port fields, we analyze the COMPAX and PLWAH-compressed bitmap indexes. In particular, we are interested in understanding for which bitmap index columns PLWAH compresses better than COMPAX. In Fig. 12, we report the disk consumption (expressed in numbers of 32-bit words) for every column of the bitmap index. Columns have been ordered by number of matching results, i.e., the number of *ones* they carry, so that sparse columns are on the left and dense columns on the right. The figure shows a clear separation between the two encodings: PLWAH performs better when compressing sparse columns, whereas COMPAX offers better compression rates for denser columns, which usually require more space. However, the difference in space consumption between COMPAX and PLWAH is higher for dense columns and lower otherwise. This means that when perform-

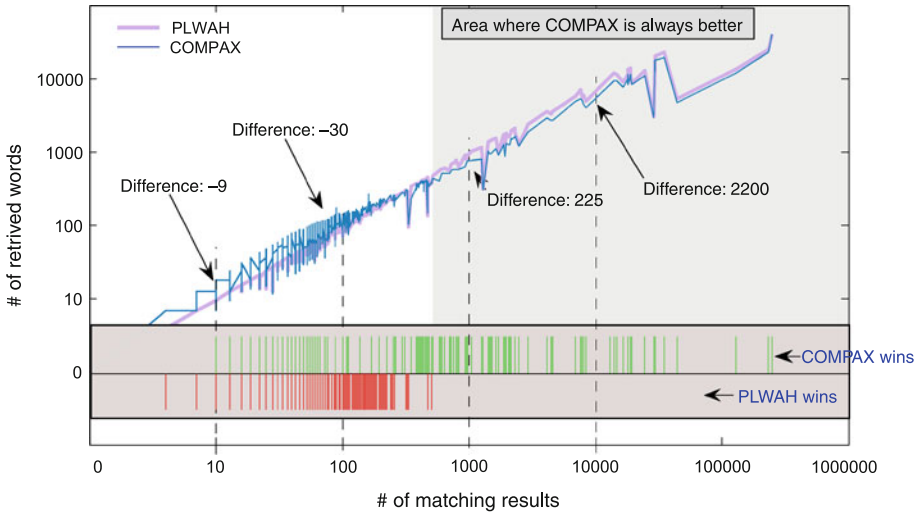

Fig. 12 Comparison between COMPAX and PLWAH. The x-axis shows the number of results (number of ones) contained in a column. The y-axis shows the number of 32-bit words (codewords) that encode the particular bitmap index column. At the *bottom* of the figure we also juxtapose a 'win–lose' graph that illustrates when each technique is better. Evidently, COMPAX compresses data using fewer words, when the data contain large number of matches (dense columns). Sparse columns holding few matches are encoded more efficiently using PLWAH

ing queries for common ports (dense columns), the COMPAX index requires fewer words to be fetched from the disk.

The behavior of the two bitmap encodings when indexing ports and IP addresses suggests that when one is interested in achieving the best possible compression rate, different compression mechanisms can be used for different data attributes. However, the adoption of heterogeneous compressed bitmaps requires boolean operations to be implemented across different encodings. While this approach is feasible, in practice, it will substantially increase the complexity of the software, making it harder to test, debug and maintain.

In general, the reasons for selecting COMPAX over PLWAH are the following: First, as IP addresses typically are the most commonly queried attributes, COMPAX will lead to better compression for these attributes and hence increase the operating system cache effectiveness. Second, when deploying the system on larger networks, the number of distinct IP addresses (and therefore the entropy) is expected to increase significantly, whereas the distribution of ports under normal traffic conditions is unlikely to approach a uniform distribution. Third, the disk consumption advantage of PLWAH over COMPAX when indexing the port fields is significant only for uncommon ports that, in any case, do not require substantial indexing space.

6.4 Index performance

Now, we evaluate the overall index performance with respect to other currently commercial available solutions, such as WAH, which is used in many databases (e.g., Oracle). WAH has already been extensively used by the research community, and there is a well tested and freely available implementation of WAH in FastBit [17]. We compare the search performance on just the bitmap index, when encoded

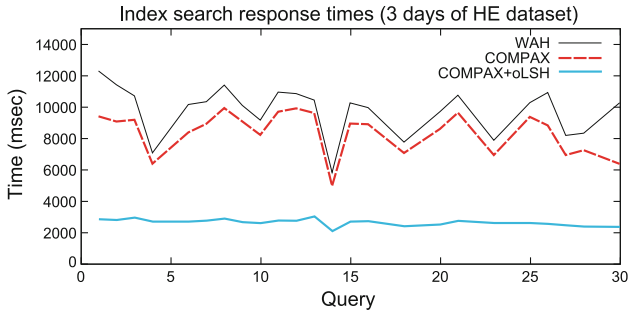

Fig. 13 Comparison of index search time for 3,000 random IP addresses

with WAH, COMPAX, or COMPAX+oLSH. We pose queries using random IP addresses, since IP addresses are typically the most queried attribute. By querying exact IP addresses, we are evaluating the performance of boolean operations over compressed bitmaps as each IP lookup requires the bitwise AND-ing of four different bitmaps, each corresponding to the four bytes of an IPv4 address. We conduct the experiment over the more heterogeneous 'Hosting Environment' (HE) dataset.

We choose 3,000 random distinct IP addresses and group them into 30 sets of 100 addresses each. For every set, we report the time for executing the 100 independent IP address lookups executed sequentially. To put into perspective the performance offered by our system, we also uses WAH indexes in addition to COMPAX-based indexes. Because we are interested in measuring the performance of the index in terms of the CPU efficiency, we built an index over 3 days from the HE data set, which can be completely be cached by the system. The size of the index is 1.5 GB for WAH, 845 MB for COMPAX and 314 MB for COMPAX+oLSH. The measurements are reported in Fig. 13.

The response times shown depend entirely on the performance on the boolean operations among the IP address byte indexes. The index query on the COMPAX-based index is on average 15% faster than the WAH-based index. This result shows that adopting a more complex bitmap encoding does not necessary result in performance penalties when implementing boolean operations directly on the compressed bitmaps (i.e., without explicit decompression). In fact, for COMPAX bitmaps, the more complex decoding is compensated by increased cache locality, which is a direct consequence of the space savings (smaller working set).

COMPAX+oLSH is four times faster, on average, than WAH to complete the IP address lookups. In this case, the improvement is not just due to increased cache locality, but rather to the sorting itself. The sorting results in literal words that are more dense; so, fill words can represent longer sequences. In this way, the number of bitwise instructions is substantially reduced [31].

6.5 Query performance: index and archive

Finally, we measure the complete system performance during query time, which corresponds to evaluating the retrieval time of both index (with COMPAX encoding) and archive. Index and archive are created with and without oLSH reordering. We measure the cumulative response time required when executing queries over the 1.2 billion flows of the data set PN. The index+archive without the oLSH option amounts to approximately 31 GB. With oLSH enabled, the resulting repository size is 20 GB. Naturally, we expect that the oLSH variant results in a faster query response time.

Low selectivity: We pose 10,000 random IP address queries with an increasing number of wild cards such as

```
srcIP = 10.4.5.*, dstPort=X, dstIP = 10.5.5.*
srcIP = 10.4.5.*, dstPort=X, dstIP = 10.5.*.*
srcIP = 10.4.*.*, dstPort=X, dstIP = 10.5.*.*
...
```

to create queries that would have to retrieve an increasing number of results. We sort and 'bin' the number of results into a histogram format. Each bin represents the average number of results returned by all queries in the same bin. In Fig. 14, we report three measures:

Index time:	The time needed to search the index and join the resulting indexes.
Archive time:	The time needed to retrieve the data from the archive.
Total query time:	The sum of these two measures.

Each of the graphs also illustrates the runtime with and without the oLSH option. One can observe that by using oLSH reordering the total query time improves by more than 2 times. This result is attributed to the smaller sizes of both

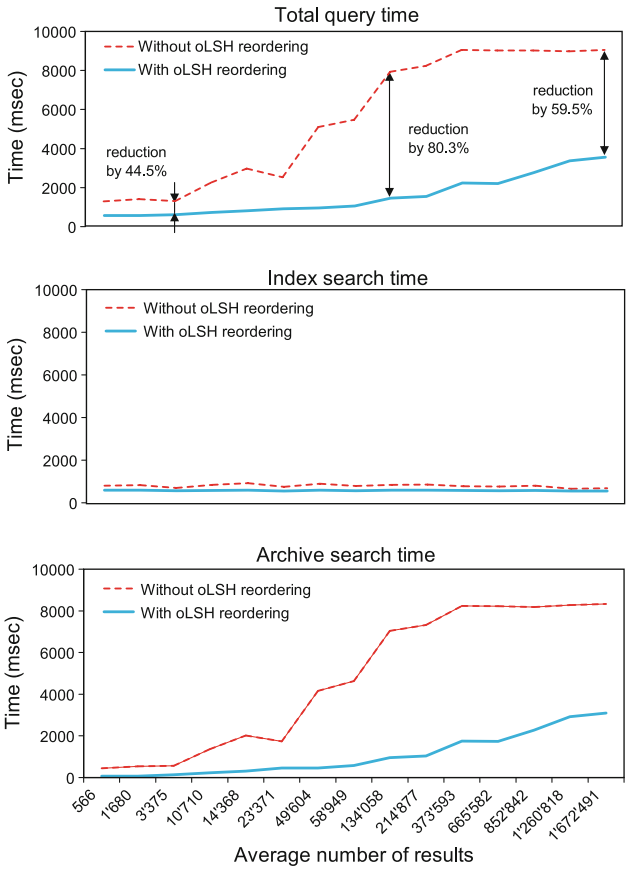

Fig. 14 Query time versus number of results: comparison of the total query time (*top*) and its separation into time on the index access (*middle*) and data retrieval time from the archive (*bottom*), with and without oLSH reordering

index and archive after oLSH reordering. For the queries with the highest selectivity, the response time can reach 3 s, whereas for the queries with lower selectivity, the results can be returned in less than a second. This outcome is very encouraging because we do not make the query be selective with respect to time (i.e., "find only matches during day A"), but queried the entire flow repository for results.

On the same figure, in the bottom graph, we segregate the total query time into its components. The time to retrieve results from the index and perform a join on them (Fig. 14 middle) is nearly constant and approximately on the order of 500 ms. The majority of the search query is spent on the retrieval from the archive, which is depicted on the lower part of the figure. One can easily observe the great performance boost attributed to oLSH reordering. The careful packaging of flow records in both the index and the archive eventually leads to a significant improvement in the query performance. In particular, the gains from flow reordering in the archive can reach the 400% range. This is attributed to the better packing of similar records. In fact, the reordering reduces the number of blocks that must be decompressed at the data archive level, and in addition, because of the reduction of the over-

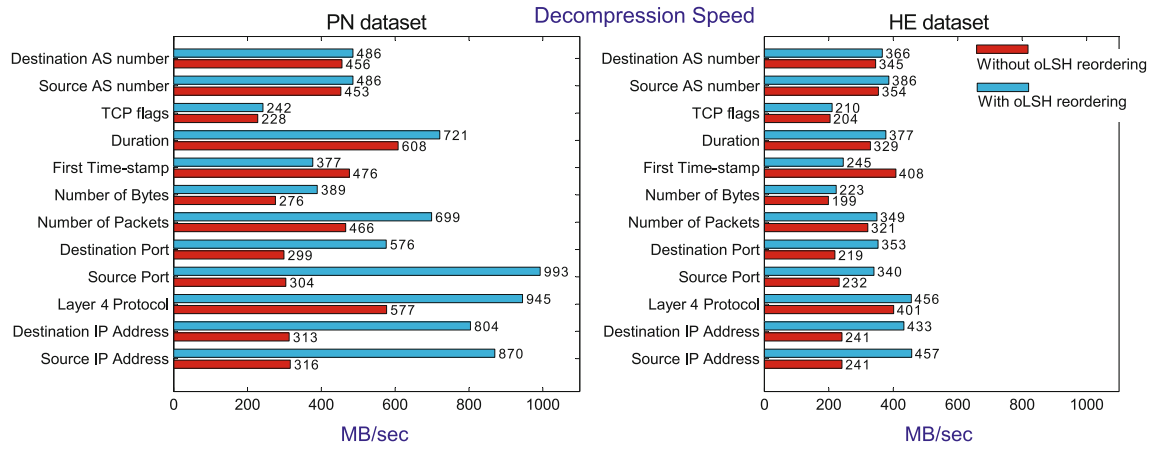

Fig. 15 Decompression speed in MB/s. Note that for the most commonly queried attributes, such as source/destination IP addresses and ports, the decompression speed is more than double when oLSH reordering has been applied

all entropy within each block, it boosts the decompression performance.

High selectivity: In many cases, decompressing the entire data archive or entire data columns might be required; this is the case, for example, when one has to compute the set of distinct IP addresses. For such a scenario, the bitmap index does not help, as the entire archive on one or more attributes needs to be decompressed. To evaluate such cases, we measure the decompression speed when the *entire* data archive is unpacked. We measure both variants: with and without oLSH tuple reordering.

For each network flow attribute, we measure the CPU time spent on the execution of the decompression routines only. In this way, we do not take into account the time spent on I/O, which is higher for the non-ordered data because of the higher disk consumption. Therefore, the decompression speeds reported in Fig. 15 (expressed in Megabytes/second) represent a lower bound for the decompression performance improvement.

We observe that the oLSH reordering significantly boosts the decompression speed of the *lzo* compressor by as much as 3.6 times. In both data sets, the decompression speed boost is substantial for IP addresses and ports, which are the most commonly queried attributes.

It is worth noting that, with the only exception of the timestamp attribute archive, the reordering has a positive impact on the decompression speed. This is indeed expected even if the system uses IP addresses, ports and protocol attributes (the so called 5-tuple) when performing the oLSH reordering. In fact, there is an high correlation between the 5-tuple and the remaining fields, with the only exception begin the timestamp attribute. This can be intuitively explained as a combination of ports and protocol identifies in the majority of the cases the networked application (e.g., HTTP usually runs on port 80/TCP), which, in turn, tends to manifest specific behaviors in terms of session duration and number of packets and bytes transferred. Similarly, the source and destination Autonomous System numbers are related to the IP addresses. The timestamp attribute, on the contrary, is not strongly related to any other 5-tuple field, and in addition, it is the only attribute having partially ordered values.[7] For these reasons, timestamp is the only attribute where oLSH reordering has a negative impact on the compression rate and, consequently, on the decompression speed.

7 Application of NET-FLi

In this section, we demonstrate the utility of NET-FLi for typical scenarios in network monitoring: (1) detection of worm propagation and (2) interactive exploration of the network traffic history.

Worm propagation: Consider the scenario of an outbreak of a computer worm (cf. Fig. 16) that exploits a vulnerability of an application protocol. All network nodes being contacted by the infected host need to be discovered to (a) quarantine any other possibly infected machine, and (b) better understand the propagation pattern of the worm.

Filtering a large flow repository for a small subset of flows is a tedious task, as the entire repository needs to be scanned linearly. Using the index capabilities of NET-FLi, we expect to achieve a significant reduction of the time needed to identify possibly infected machines.

We perform an analysis on the 2-month PN data set consisting of 1.2 billion flows and focus on machine M, which is suspected to spread a worm using a vulnerability on service port 123. We query the system for all machines to whom

[7] Flow meters export a flow record for a monitored network flow once it has expired or terminated. The timestamp of the first packet belonging to the flow is not used by flow meters to impose an order on the exported flow records.

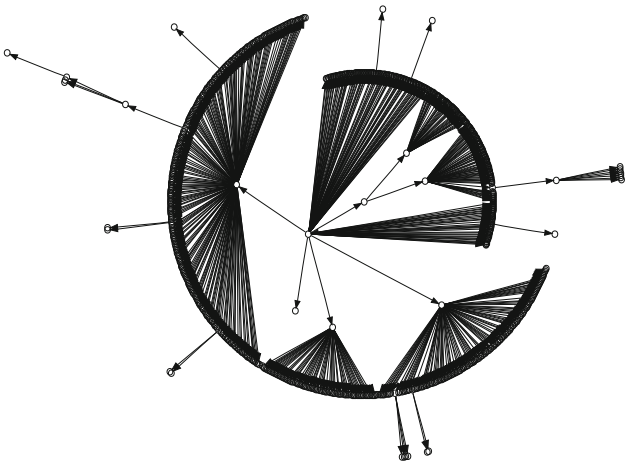

Fig. 16 Propagation graph of a worm epidemic used to identify potentially compromised nodes in a network

the machine connected to on service port 123. Therefore, the query has the form:

```
SELECT DstIP
WHERE SrcIP = M AND DstPort = 123
```

Figure 16 displays an example of a worm propagation pattern: the center node represents a suspicious machine, whereas an edge signifies a communication link on a vulnerable port between the two machines.

The connectivity graph to a node can be created with NET-FLi by recursively querying for flows emitted by suspected infected machines in the network. We measure the time needed to find all 2,225 relevant graph connections and to retrieve the corresponding *dstIP* addresses using COMPAX+oLSH, with a completely empty cache (unmounting the disk) as well as with a 'warm' cache at the operating system level, when other queries have previously been posed. We repeat the experiment 100 times. We discover that the uncached query response time is 62.314 s on average (with a standard deviation σ of 0.798). When reissuing the same query with a previously 'warm' cache, the response time drops down to 2.345 s on average ($\sigma = 0.051$). In comparison, the same query executed on a conventional flat flow file repository using linear scanning over all records takes as much as 6,062 s, i.e., more than two orders of magnitudes longer than the identical NET-FLi query.

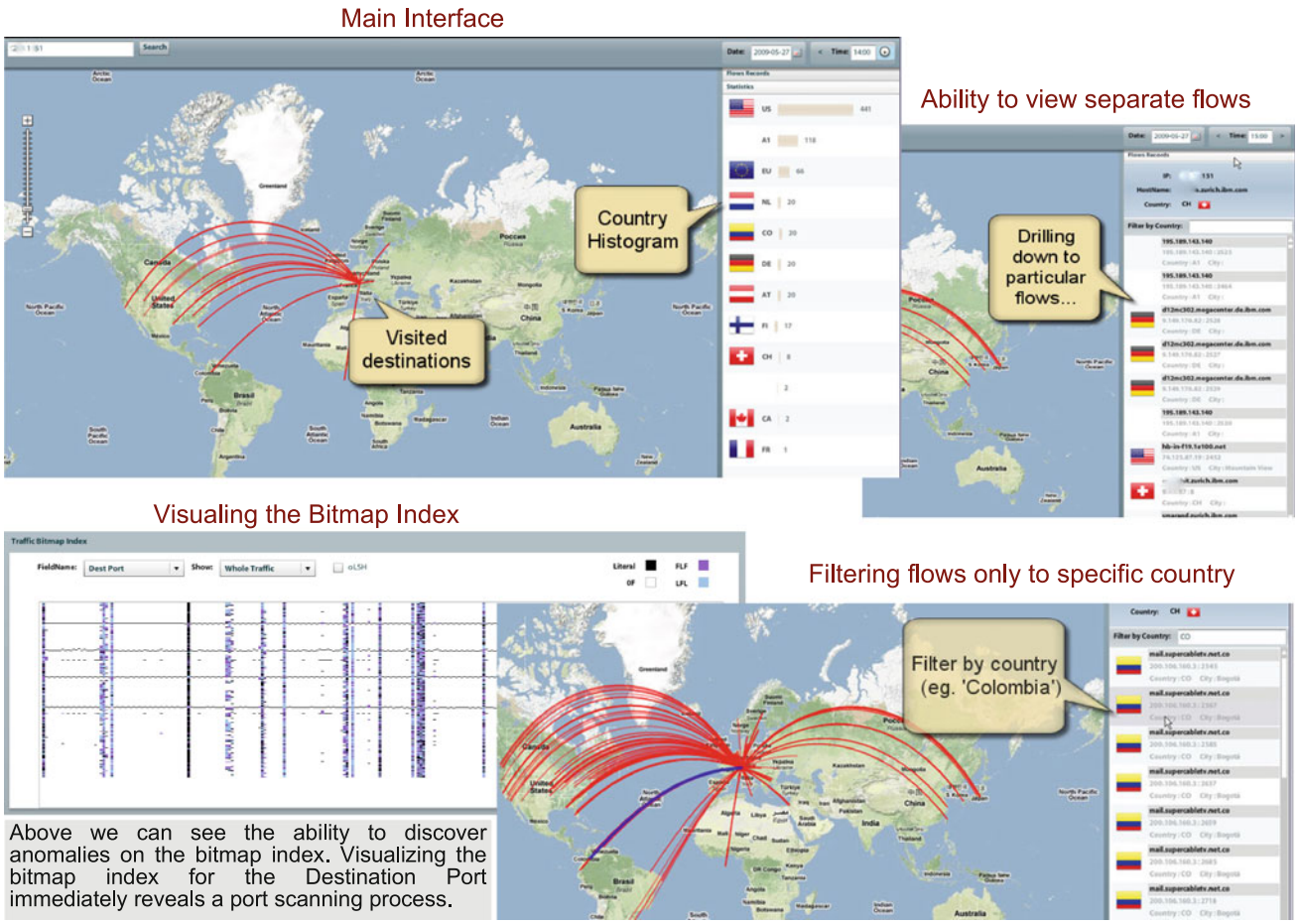

Fig. 17 Application for interactive exploration of flows and anomalies that is built on top of our technology

Our results show that the NET-FLi approach exhibits low response times for locating the candidate records and returning the flow data. In addition, NET-FLi can exploit the cache capabilities offered by the operating system, without requiring a custom-made cache manager. The last observation is particularly attractive for interactive query refinement: for example, in a network investigation, typically a number of queries are used to narrow down the root cause. Subsequently, refined queries on previously cached indexes can be answered almost for free. On the contrary, linear scan approaches cannot benefit significantly from the LRU-oriented cache system of the OS.

Interactive exploration: On top of NET-FLi, we built an interactive graphical interface that enables easy exploration of network flows. Given a specific IP (port, protocol, etc) and a time frame, the interface discovers all flows involved, which are then overlaid on a map of the globe. Results are also aggregated by country. An example of this query interface is shown in Fig. 17. For the IP address, date, and time (which can be of a compromised node), we can quickly identify which other Internet addresses are contacted by this host. Contacted destinations are aggregated by country on the right-hand side. However, the user can also drill-down on the individual IP addresses for more information (top-right graph). In addition, basic flow filtering by country is also incorporated in the visual interface (bottom-right graph).

Finally, we provide the capability to visualize the actual bitmap index for any of the attributes indexed. This is a particularly useful functionality, because it facilitates the visual identification of anomalies. Consider, for example, the snapshot of the bitmap index shown on the bottom left-hand side of Fig. 17. This captures a portion of the bitmap index for the attribute 'Destination Port' of the network flows. From the horizontal lines (columns represent the different ports), it is immediately apparent that some resource is initiating a port scanning process. If this belongs to an unscheduled activity, it is something worth investigating by the data administrator.

Such cases can easily be identified by visualizing the bitmap indexes constructed. In addition, recording an 'average' or expected bitmap index profile and monitoring the deviation of the current traffic from that profile can provide an interesting avenue for discovering network anomalies. These possibilities can be potentially accommodated on top of the technology and visualization tools provided, but reside outside the scope of the current work.

8 Conclusion

We introduced NET-FLi, a high-performance solution for high-speed data archiving and retrieval of network traffic flow information. Our solution achieves the following:

- Data record insertion rates in the range of **0.5M to 1.2M flows per second** depending on the desired compression level.
- A novel adaptive, compressed bitmap indexing technique, called COMPAX.
- An on-the-fly stream reordering approach based on locality-sensitive hashing (LSH) that renders the data compression rate of flow records **equivalent to that of gzip**. This fast record reordering scheme is of general interest for other bitmap encoding methodologies.

Our solution can be used to drive a wide spectrum of applications including iterative hypothesis testing in network anomaly investigation, on demand traffic profiling, and customized reporting and visualization. Moreover, we believe that it can be applied to many other domains challenged by high data volumes and exceptionally high insertion rates.

References

1. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 671–682 (2006)
2. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the borealis stream processing engine. In: Second Biennial Conference on Innovative Data Systems Research (CIDR) (2005)
3. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores versus row-stores: how different are they really? In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 967–980 (2008)
4. Andrade, H., Gedik, B., Wu, K.-L., Yu, P.S.: Scale-up strategies for processing high-rate data streams in system S. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE), pp. 1375–1378 (2009)
5. Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Inf. Retr. **8**(1), 151–166 (2005)
6. Antoshenkov, G., Ziauddin, M.: Query processing and optimization in Oracle Rdb. Very Large Data Bases J. **5**(4), 229–237 (1996)
7. Apaydin, T., Ferhatosmanoglu, H., Canahuate, G., Tosun, A.C.: Dynamic data organization for bitmap indices. In: Proceedings of International Conference on Scalable Information Systems (INFOSCALE), pp. 30:1–30:10 (2008)
8. Bethel, E.W., Campbell, S., Dart, E., Stockinger, K., Wu, K.: Accelerating network traffic analysis using query-driven visualization. In: Proceedings of IEEE Symposium on Visual Analytics Science and Technology (IEEE VAST), pp. 115–122 (2006)
9. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the memory wall in MonetDB. Commun. ACM **51**(12), 77–85 (2008)
10. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: TelegraphCQ: continuous dataflow processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 668–668 (2003)
11. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a

distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 1–26 (2008)
12. Cranor, C.D., Johnson, T., Spatscheck, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 647–651 (2003)
13. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedingds of the Symposium on Computational Geometry, pp. 253–262 (2004)
14. Deliége, F., Pedersen, T.B.: Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 228–239 (2010)
15. Endace. Endace Measurement Systems, NinjaProbe Appliances. http://www.endace.com
16. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 670–680 (2010)
17. FastBit. An Efficient Compressed Bitmap Index Technology. https://sdm.lbl.gov/fastbit/
18. Ferragina, P.: Data structures: time, I/Os, entropy, joules! In: Proceedings of 18th Annual European Conference on Algorithms: part II, pp. 1–16 (2010)
19. Fujioka, K., Uematsu, Y., Onizuka, M.: Application of bitmap index to information retrieval. In: Proceedings of the international World Wide Web conference (WWW), pp. 1109–1110 (2008)
20. Fusco, F., Stoecklin, M., Vlachos, M.: 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)
21. Gailly, J.-L., Adler, M.: The ZLIB library. http://www.zlib.org/
22. Gates, C., Collins, M., Duggan, M., Kompanek, A., Thomas, M.: More netflow tools for performance and security. In: Proceedings of USENIX Conference on System Administration, pp. 121–132 (2004)
23. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 518–529 (1999)
24. Giura, P., Memon, N.: Netstore: an efficient storage infrastructure for network forensics and monitoring. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID), pp. 277–296 (2010)
25. Haag, P.: Nfdump. http://nfdump.sourceforge.net/
26. Harizopoulos, S., Liang, V., Abadi, D.J., Madden, S.: Performance tradeoffs in read-optimized databases. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 487–498 (2006)
27. Holloway, A.L., DeWitt, D.J.: Read-optimized databases, in depth. Proc. VLDB Endow. **1**, 502–513 (2008)
28. IBM Corp., AURORA—Traffic Analysis and Visualization. http://www.zurich.ibm.com/aurora/
29. Intel. Intel. SSE4 Programming Reference (2007)
30. Karagiannis, T., Papagiannaki, K., Faloutsos, M.: BLINC: multilevel traffic classification in the dark. In: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), pp. 229–240 (2005)
31. Kaser, O., Lemire, D., Aouiche, K.: Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes. In: Proceedings of International Workshop on Data Warehousing and OLAP (DOLAP), pp. 1–8 (2008)
32. Lemire, D., Kaser, O., Aouiche, K.: Sorting improves word-aligned bitmap indexes. Data Knowl. Eng. **69**(1), 3–28 (2010)
33. Li, X., Bian, F., Zhang, H., Diot, C., Govindan, R., Hong, W., Iannaccone, G.: MIND: a distributed multi-dimensional indexing system for network diagnosis. In: Proceedings of the IEEE International Conference on Computer Communications (INFOCOM) (2006)
34. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 950–961 (2007)
35. Morariu, C., Kramis, T., Stiller, B.: DIPStorage: Distributed storage of IP flow records. In: Proceedings of the 16th Workshop on Local and Metropolitan Area Networks (LANMAN) (2008)
36. Network Top. http://www.ntop.org/
37. Niksun. Niksun NetDetector. http://niksun.com
38. Oberhumer, M.F.: The Lempel-Ziv-Oberhumer Packer. http://www.lzop.org/
39. Oberhumer, M.F.: Lzo documentation. http://www.oberhumer.com/opensource/lzo/lzodoc.php
40. Pinar, A., Tao, T., Ferhatosmanoglu, H.: Compressing bitmap indices by data reorganization. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE), pp. 310–321 (2005)
41. Plagemann, T., Goebel, V., Bergamini, A., Tolu, G., Urvoy-Keller, G., Biersack, E.W.: Using data stream management systems for traffic analysis—a case study. In: Proceedings of the Passive and Active Measurement Conference (PAM), pp. 215–226 (2004)
42. Reiss, F., Stockinger, K., Wu, K., Shoshani, A., Hellerstein, J.M.: Enabling real-time querying of live and historical stream data. In: Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM), pp. 28 (2007)
43. Romig, S., Fullmer, M., Luman, R.: The OSU flow-tools package and CISCO NetFlow logs. In: Proceedings of USENIX Conference on System Administration, pp. 291–304 (2000)
44. Schatzmann, D., Mühlbauer, W., Spyropoulos, T., Dimitropoulos, X.: Digging into https: flow-based classification of webmail traffic. In: IMC '10: Proceedings of the 10th Internet Measurement Conference. Melbourne, Australia, Nov (2010)
45. Stabno, M., Wrembel, R.: RLH: bitmap compression technique based on run-length and Huffman encoding. In: Proceedings of ACM International Workshop on Data Warehousing and OLAP (DOLAP), pp. 41–48 (2007)
46. Stonebraker, M., et al.: C-Store: a column-oriented DBMS. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 553–564 (2005)
47. Sullivan, M., Heybey, A.: Tribeca: a system for managing large databases of network traffic. In: Proceedings of USENIX Annual Technical Conference, p. 2 (1998)
48. Wu, K., Otoo, E., Shoshani, A.: On the performance of bitmap indices for high cardinality attributes. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 24–35 (2004)
49. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Trans. Database Syst. **31**(1), 1–38 (2006)
50. Wu, K., Otoo, E.J., Shoshani, A., Nordberg, H.: Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA (USA)
51. Wu, K.-L., et al.: Challenges and experience in prototyping a multimodal stream analytic and monitoring application on System S. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 1185–1196 (2007)
52. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE), p. 59 (2006)