An efficient storage technique for network monitoring data

Giuseppe Aceto, Alessio Botta, Antonio Pescapé
University of Napoli Federico II, Italy
Email: {giuseppe.aceto, a.botta, pescape}@unina.it

Cedric Westphal
Docomo Innovations
Email: cwestphal@docomoinnovations.com

***Abstract*—Monitoring modern networks involves storing and transferring huge amounts of data. For this reason, compression techniques are typically used in order to reduce the space and time needed for these operations. The main drawback of this approach is that, when data has to be processed, a preliminary decompression is necessary, which increases the time and computational power needed. To cope with this problem, in this paper we propose a technique that allows to transform the measurement data in a representation format meeting two main objectives at the same time. Firstly, it allows to perform a number of operations directly on the transformed data with a controlled loss of accuracy, thanks to the mathematical framework it is based on. Secondly, the new representation has a small memory footprint, allowing to reduce the space needed for data storage and the time needed for data transfer. To validate our technique, we perform an analysis of its performance in terms of accuracy and memory footprint. The results show that the transformed data closely approximates the original data (within 5% relative error) while achieving the compression ratio of 20%; storage footprint can be gradually made close to that of the state-of-the-art compression tools, such as bzip2, if higher approximation is allowed.**

I. Introduction

Telecom operators have to constantly monitor the network for a number of tasks, such as billing, management, provisioning, dimensioning, etc.. Some of these tasks such as billing require the analysis of the data in near real-time, while others are performed a posteriori on historical data sets. For instance, network provisioning is performed on a set of statistical indicators calculated over the last months or years. For these tasks monitoring infrastructures have been presented such as [1] that rely on data reduction techniques to keep the amount of data manageable. Similar issues were early addressed by the networking research community, e.g. [2] but the evolution of technology has worsened them, requiring more "aggressive" lossy approaches such as adaptive shedding of input data [3]. On the other hand, the availability of network monitoring data has allowed for more complex analyses such as behavioral pattern mining [4], high valuable for both content providers and communication infrastructure managers.

For all of these cases, it is often required to save the monitoring data; this implies storing for a long time a huge amount of information. For instance, a telecom operator willing to analyze service access patterns could easily face data sizes in the order of terabytes per day, to be transmitted from monitoring points to processing sites, and accumulated for over several months. One possible solution resorts to state-of-the-art compression algorithms, in order to keep the storage footprint - and the transmission burden - manageable; but this approach has the drawback of needing a decompression stage (and then space for the uncompressed data) before any further processing.

In order to cope with this issue, we propose a technique that i) is efficient in space utilization, and ii) provides the possibility to perform a class of operations directly on the compressed data. The technique we present, described in Sect. II, trades off accuracy for reduced memory footprint and computational complexity for postprocessing, allowing for different levels of approximation or compression, according to the needs of the intended application. To validate our technique, in Sect. III we perform an analysis of its performance in terms of accuracy and memory footprint. The results show that the transformed data closely approximates the original data while the compression ratio is close to that of the state-of-the-art compression tools, such as bzip2.

II. Our technique

The technique we propose produces a spatially efficient data representation scheme that allows approximated computations and pattern recognition in network monitoring logs, with no need for decompression stage. In order to ease the description of the technique, of its characteristics, and of the challenges it must deal with, we consider a specific format of monitoring log, but the technique can be applied to more complex monitoring data. The considered monitoring log format, analogous to flow-based traffic traces, is constituted of records, each comprising four fields: *timestamp*, *source IP*, *destination URL*, *load*; each record represents a single HTTP conversation originated by a source IP to retrieve the given URL, and the amount of data that has been exchanged. Even if we focus on this type of traffic for ease of exposition - it represents a reasonable example of data commonly logged by operators - our technique applies to other types of data as well. To be able to perform computation directly on the representation, we apply a technique which identifies patterns in matrices where each row/column is a vector of real numbers. Thus

Part of the research activities presented in this paper have been performed during a summer intership by Mr. Giuseppe Aceto at Docomo Innovations. The research has been partially funded by LINCE project of the FARO programme jointly financed by the Compagnia di San Paolo and by the Polo delle Scienze e delle Tecnologie of the University of Napoli "Federico II".

978-1-4577-0457-4/11/$26.00 ©2011 IEEE

our technique will take two phases: in the first phase (the preprocessing phase), we will first convert the IP addresses and URLs into numbers in a Euclidean space, and then we will scale all fields so that each of them ranges between 0 and 1. As a result of this preprocessing phase we obtain a fully numeric matrix representation of the original data. In the second phase (the factorization phase), we transfor the numeric matrix into a couple of sparse matrices which approximate our original data, while having a smaller memory footprint.

In the following sections, we detail the two phases of our technique.

A. Mapping and Normalizing

1) URL filtering: As in some applications singular events may be of no importance, the proposed technique provides the possibility to control the presence of rare connections to servers; this is done by means of a *URL filtering threshold* parameter, defined as the minimum number of occurrences a URL must exhibit in the log to be considered. Such kind of filtering reduces the number of uniques URLs, but also affects some sources (all sources communicating with an under-the-threshold URL are discarded).

2) Label Coding: Even if source IPs could be represented as their numerical value, this value conveys little meaning, and presents a strongly uneven distribution, so we treat IP as a non-numerical value to be processed similarly to URL field. The values in the field *URL* (destination URLs[1]) are stored as a list of unique elements; URLs are ranked by number of occurrences, and mapped to an integer equal to their rank order: the higher the frequency, the higher the numerical label assigned to them. In a typical encoding, a shorter code is assigned to more frequent items. However, in our case, small values might be changed to zero when attempting to find a sparse representation and we would like to preserve the values of the most frequently accessed items. The values in the field *source IP* are stored as a list with unique elements: the label corresponding to a source IP is given by its position in the list, and referred to as *srcID*.

3) Normalization: The fields *srcID*, *Bytes* and *URLcode* are scaled to $]0, 1]$. For *srcID* this is done by dividing the code (ranking) by the total number of *srcID*s. The same is done for URLs. Due to the high variance of values in *Bytes* field, base-2 logarithm of the value is taken, and divided by its maximum over the trace.

4) Timestamp: The timestamps are considered implicitly in the ordering of the transactions, and they are not included in the processing. Due to the large number of independent transactions in a window of time, one can assume that these transactions are regularly spaced in time with an acceptable error. Define by Δ the average inter-arrival time within the observed time window τ. Thus, $\Delta = \tau/N$ where N is the number of entries in the log. τ ranges from a few minutes to a few hours. The timestamp of entry k is replaced by:

$$\tilde{t}_k = t_1 + (k-1) * \Delta$$

This allows to only keep the ordering of the entries.

In the following we calculate the error made with this approximation. Define a_i to be the sequence of inter-arrival times in between connection requests i and $i+1$. a_i is a Poisson distributed random variable with mean $E[a] = \Delta$ (and thus variance Δ as well). The actual arrival time for the k-th connection is thus $t_1 + \sum_{i=1}^{k-1} a_i$. The error in the time stamp is: $e_k = \sum_{i=1}^{k-1} a_i - (k-1)\Delta$. By the central limit theorem, e_k is distributed according to a normal distribution $\mathcal{N}(0, \sqrt{k}\Delta)$. Since $\Delta = \tau/N$, where N is the number of connections, and since $k \leq N$, the error is upper bounded by $\mathcal{N}(0, \tau/\sqrt{N})$, and it goes to zero as the number of connection grows within a time window τ. Considering as a typical scenario the case of a cellular operator, the system under consideration has a number of users in order of tens to hundreds of millions, thus the assumption that N is large is reasonable.

5) Matrix format: As the last step of the preprocessing phase we build P, which is an $N \times M$ matrix, representing N connections, each of which with M dimensions (we described three fields in the input data format above, but we could consider other flow parameters as well). In this representation, each column of the input matrix is a vector $(srcID; URLcode; Bytes)^T$ where the time is implicitly represented by column index, namely the i^{th} column holds values of the i^{th} entry of the input file.

B. Factorization

In order to store P efficiently, we want to use a matrix $\hat{P}$ such that: $\hat{P}$ approximates P, and $\hat{P} = TC$, where $T(N \times K)$ and $C(K \times M)$ are two matrices with high sparsity and thus requiring a small amount of memory for their storage. $K \leq M$ is a parameter that is optimized at the same time as when T and C are derived. As our objective is to trade off in computational complexity vs accuracy, we model our approximation as the minimization problem:

$$\min_{T,C} ||P - TC||_2^2 + \lambda(||T||_0 + ||C||_0) \quad (1)$$

where $||.||_0$ is the ℓ_0 norm that counts the number of non-zero elements of its argument, and λ is a Lagrangian multiplier which is specified by the user. T is the pattern basis and C a matrix of coefficients. Namely, a column vector $\hat{p}_k$ from $\hat{P}$ can be expressed as a decomposition along the basis vectors t_i in T:

$$\hat{p}_k = \sum_{i=1}^{K} c_k(i) t_i \quad (2)$$

T can thus be used to identify patterns in P and answer queries about patterns in the logged data. Note that since the optimization problem (1) takes into account the number of non-zero elements of T and C, it yields sparse results, thus reducing the storage requirement and also the computational complexity required by operations on the compressed representation. In

[1]We consider a general domain as a URL, without including php arguments or subpaths within a domain.

more details, to answer a specific query that can be put in the form $Y = PX$, one can use $\hat{P}$ instead of P and solve $\hat{Y} = \hat{P}X = TCX$. Remember that $||T||_0$ and $||C||_0$ are minimized by construction. Therefore, computing the product CX requires at most $||C||_0$ multiplications of coefficients. Similarly, computing $Y = T(CX)$ requires no more than $||T||_0$ multiplications. Thus, the complexity of answering a query that can be put in the form $Y = PX$ is equal to $||T||_0 + ||C||_0$ operations.

Using this factorized format allows to answer a range of queries. For instance, one can answer any max-k transaction query to find the k largest transactions in the log file. This can be solved by finding the k largest value of the $N \times 1$ row of $\hat{P}$ that corresponds to the load. One can similarly find the total usage of a specific *srcID*, by summing all bytes value $\hat{P}(3, i)$ for which $\hat{P}(1, i)$ = *srcID*. The matrix C points to which patterns in T the user calls upon. Thus similar users will have similar coefficient in the C matrix, and can be identified by observing this sparse matrix. Conversely, the underlying matrix of patterns T embeds some overall behavior of the system and can be used to identify abnormal usage. In particular, if after computing T over some period of time Δ at regular intervals, one sees dramatic changes in the composition of T, say $\min_\pi ||T(t_2) - \pi T(t_1)||_2 > \gamma$ where π is a column permutation and γ a threshold, then it might point to some abnormal behavior in the system and call for some investigation.

In order to compute T and C, we use the technique proposed by Zujovic et al [5] in the context of pattern matching algorithms (applied to query-by-example image retrieval).

III. Experimental evaluation

To evaluate the performance of our technique, we have considered its efficiency in terms of memory footprint reduction (*Compression Ratio*) and different consequences of the approximation (*Bytes error*, *ID error*, *URL error*). For the evaluation of the memory footprint, the compression ratio is compared with the output of the general purpose compression utility `bzip2`, employing the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding. The reported results are obtained by varying the value of λ, that controls the amount of distortion allowed in the approximated factorization algorithm (the higher the value of λ, the higher the tolerated approximation, but also the more sparsity induced in the representation matrix); preprocessing stages are identical, with a fixed URL threshold[2], while λ is varied in the set $\{0.001, 0.0025, 0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1\}$.

A. Data set

For a proof of concept of the effectiveness of our technique, we use a real traffic trace: the considered data cover a time span of 1 hour, presenting 26965 sessions, with 110 different *source ID*s exchanging 907.024 MB of data with 1771 different *destination URL*s. The data set has the format of a log file, each record of which represents a single HTTP session, and is constituted by four fields: *timestamp* (in UNIX epoch time, μs precision), *source ID*, *destination URL*, *load* (in bytes).

[2]We performed an analysis of the effects of the filtering threshold, and found that 5 is the maximum value for which the accuracy is not significantly affected: all presented results refer to this value of URL threshold.

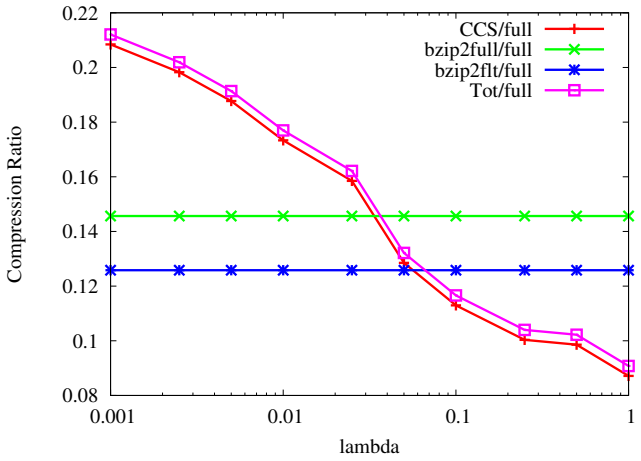

Fig. 1. Compression Ratios

B. Results

1) Compression Ratio: The total size of the compressed version, as well as the size of specific components, is compared against the size of the original data. The quantities whose ratio is considered are: **CCS** - size in bytes of Compressed Column Sparse representation of `C` matrix alone; **Tot** - sum of the size in bytes of CCS, `T` matrix, bzip2-compressed ordered list of URLs, bzip2-compressed ordered list of source IDs (this, with a few metadata, is all is needed to rebuild the original data); **bzip2flt** - size in bytes of the URL-filtered and bzip2-compressed version of the original data; **bzip2full** - size in bytes of the bzip2-compressed version of the original data.

The CCS component is calculated in bits as:

$$nnz \cdot (basesize + \lceil log_2(cols) \rceil) + \lceil log_2(nnz) \rceil \cdot (rows + 1)$$

where *nnz* is the number of non-zero elements of the matrix, *rows* and *cols* are the dimensions of the matrix, and $\lceil \cdot \rceil$ is the *ceiling* function. This value corresponds to the size occupancy of a sparse matrix, represented as Compressed Column Sparse (or *Compressed Sparse Column* [6]), where indexes are binary coded, and each element is represented with *basesize* bits. In the considered case, *basesize* is 32, *rows* and *cols* are respectively 3 and 23516. The matrix `T` is represented as $N \cdot K$ values of length *basesize* bits each. Fig. 1 shows the compression ratio as a function of the granularity parameter λ. As expected, by increasing the approximation granularity, more sparsity is found in the factor matrix `C`, and therefore the size occupancy for CCS representation decreases, causing the size of total representation to gracefully decrease for growing values of λ.

2) Error on URL decoding: Error on URL decoding is calculated as the ratio of entries with mistaken URLs versus

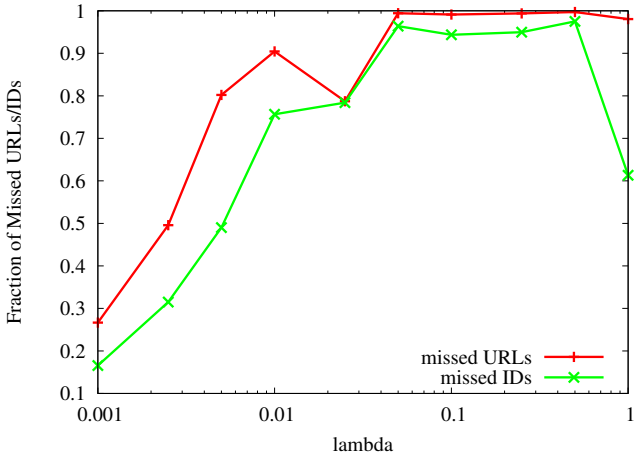

Fig. 2. Fraction of Incorrect URLs and IDs.

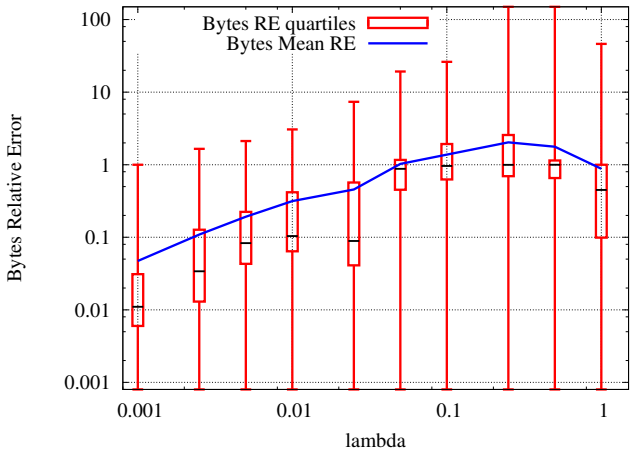

Fig. 3. Relative Error on *Bytes* field: box plot shows minimum, 1st quartile, median, and 3rd quartile; the line connects mean values.

the total number of entries. In order to calculate this value, a reconstruction of the original log is performed, using the approximated matrices and the index files. An URL is "mistaken" when the approximated index value, after the decoding, is closer to an index different from the original one. The same procedure is performed on source IDs, with analogous definitions.

The percentage of mistaken URLs and IDs is shown in Fig. 2. It can be seen that the errors increase until $\lambda = 0.01$, then for $\lambda = 0.025$ there is a plateau, and then the error reaches almost 1. This is due to the rescaling phase that precedes the approximated matrix representation: in order to uniform the value span among data of different nature, the indexes are "compacted" so that for high values of approximation granularity the distance between consecutive ones becomes smaller than the allowed approximation error, eventually causing the decoding fault.

3) Error on Bytes field: We consider two kinds of errors on the *Bytes* field: the *Relative Error*, defined as $\|\frac{x-\hat{x}}{x}\|$, and the *Signed Relative Error*, defined as $\frac{\hat{x}-x}{x}$, where x is the true value, and $\hat{x}$ is the approximated value, affected by approximation error. The approximated value $\hat{x}$ is reconstructed, for each considered value of λ, by multiplying the matrices, rescaling by the inverse of the normalization factors, and considering the *Byte* rows.

For the *Relative Error* the extreme values, the quartiles and the mean are shown in Fig. 3; due to the wide variations of the Relative Error, the plot is in log-log scale; for y-axis a minimum of 10^{-3} has been set, as minimum values of relative error are always zero. We notice that the 75-percentile is always close to mean relative error (blue solid line), which increases with increasing λ, approximatively varying as 10λ for values of λ in the decades $[0.001, 0.1]$ and showing a decrease for λ in $[0.1, 1]$.

For the *Signed Relative Error* the mean and a confidence interval as wide as the standard deviation are shown in Fig. 4; as the y values are also negative, the graph is semilogarithmic. The plot shows that the mean signed relative error is always

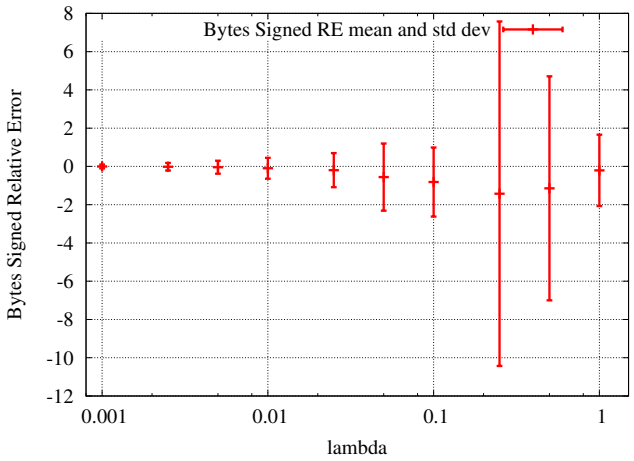

Fig. 4. Signed Relative Error: error bars are centered in the mean value, and are as wide as standard deviation.

negative, and that the standard deviation increases less than linearly for values of λ in the decades $[0.001, 0.1]$, presenting a peak of about 8 for $\lambda = 0.25$.

C. Discussion

From the experimental evaluation we find that the technique is suitable for efficient space usage, as can be seen in Fig. 1, where for fine-grained approximation ($\lambda = 10^{-3}$) the space occupation becomes 21% of the original, and compression level equivalent to the one of `bzip2` is reached, for slightly coarser approximation ($\lambda = 0.06$); this result is of notable value, as our format does not require decoding in order to perform a whole class of data mining and processing operations, as opposed to `bzip2` outcomes, that always need a decompression phase (and space for the decompressed data).

Our ongoing work is focused on researching optimizations aimed at improving the compression performance, e.g. reducing the memory footprint of CCS representation by lowering the precision of values in the C matrix from 32 to 16 bit.

In Fig. 3 we can see that the average relative error increases less than linearly for $\lambda \in \{10^{-3}, 25 \cdot 10^{-3}\}$, always close to the 75-percentile, while the median is notably lower, and less or close to 10% of the true value even for coarse grained approximation. This shows how, although some values can undergo a substantial and increasing approximation, the large part is barely affected. As can be seen in Fig. 2, the fields that are mostly affected by approximation error are *IDs* and *URLs*, due to their nature of identifiers: in this case the decoding is either hit or miss, and "approximation" to the closer IDs is still a miss. A way to reduce this impact is to exploit the pre-processing phase that precedes the approximation algorithm to extend the separation between coded identifiers. This results in a trade-off with accuracy on the other fields, and can be simply implemented by choosing weighting factors for all fields according to the relative sensitivity to approximation error, so that the difference between adjacent elements is more than double the allowed approximation error. This choice is to be done according to the intended application.

In order to analyze the effect of the proposed scheme on the *Byte* field, that represents the amount of data that has been exchanged in a single connection, the empirical distribution function has been calculated for the original file and for the approximated versions corresponding to a subset of λ values, namely $0.001, 0.01, 0.1$. Fig. 5 shows the results obtained with $\lambda = 0.01$: using a bin width of $10kB$, the histogram of the relative frequencies is plotted. It can be noted that the first bin ($0 - 10kB$) accounts for more than 80% of the total and the count of occurrences quickly falls under 0.001 times the total, reached at bin $120 - 130kB$. The same overall behavior is shown for the other values of λ and for the original data; thus the statistical characteristic of the *Byte* field are not significantly affected by the application of the scheme.

In general, the choice of allowed error, and thus, the setting of the control parameter λ, is dependent on the application. Values of λ exceeding 0.1, even if introduce significant approximation error, with higher variance (see Fig. 2,3 and 4), achieve compression levels better than the state-of-the-art reference (Fig. 1), still retaining the possibility for direct processing without decompression. In applications where the exact storage of the value is not needed (e.g. in clustering), the graceful and controlled degrading of accuracy, distributed among different monitoring fields, could be further exploited to gain even higher efficiency: ongoing work is also focused on these aspects.

IV. Related work

The issues related to storing and processing huge amounts of data coming from network monitoring activities is a strongly felt problem, on which several different approaches have been proposed. An information theoretic framework is presented in [7], that within a network model analyzes the information content of flow-level captures (using NetFlow-like format, which is analogous to the log file format we considered); by means of this modeling the authors derive the bounds for lossless compression of network traffic traces, resulting

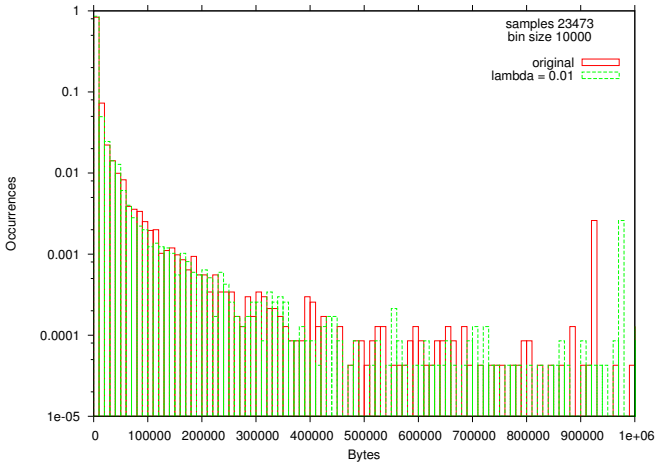

Fig. 5. Frequency distribution of Bytes of original data and processed with $\lambda = 0.01$, cut at $x = 10^6$.

in around 20% theoretical compression ratio for the format they considered.

Facing huge amounts of monitoring data, a possible strategy to enhance the performance of storing and processing is by preprocessing the data and then apply lossless compression algorithms to the obtained data representation. An example of this approach can be found in [8], where a technique is presented that aims at compression improvement by reordering on high dimension data. By organizing data in *data matrices* in which each row is an occurrence of a multi-field values, it applies an optimization algorithm that changes row ordering to maximize the compression by *differential predictive coding*. A similar approach, implying a preprocessing stage whose output is fed to a general-purpose compressor, is adopted in [9], where the input data show a common basic format: ASCII text encoding, structured as a Character Separated Values database (records separated by NL, tokens/fields separated by SPACE or other delimitator character); the authors present a multi-layered approach, where the general-purpose compression algorithm (third stage) is prepended with a *string substitution stage* on subsequent couples of lines. Another example of compression gain obtained by preprocessing input data can be found in [10], where the input data is constituted by log files of mail server; a dictionary-based word replacement phase is performed before applying different lossless compression algorithms and the compression gain is evaluated, reporting improvements of up to 56 percent in compression time and up to 32 percent in compression ratio.

When input data has know structure, an ad-hoc preprocessing can be done: for URL storage and retrieval, in [11] a compression scheme is proposed based on *AVL trees*, a balanced binary tree algorithm for lookup-intensive applications. The paper is focused on URL compression and retrieval algorithms for web caches, search engines ad webcrawlers. The scheme gains a reduction in space occupancy of 50% (both store and retrieval online) or 64% (just retrieval online, the case of search engines). When on the one hand the huge amount

of data to be stored and processed is an issue, and on the other hand, some loss of information is acceptable from the point of view of the subsequent use of data, a trade-off can be applied using lossy compression schemes. In [12] a multi-scale approach is presented, that divides the input time-series in 3 components, with different time and frequency characteristics; each is separately represented with approximations that introduce error (low-pass filtering and sampling, below-threshold clipping, Johnson-Lindenstrauss compressive random projection), achieving data reduction larger than 91%. The authors also demonstrate that histograms and correlations can be approximated by using the "compressed" data (this is done with known error bounds under some assumptions on the input signal); the paper does not address multi-dimensional data, and so does not consider correlations among different dimensions.

It is worth noting that all the cited works but [12] either propose a compression algorithm/scheme or a preprocessing stage aimed at improving the compression of the original data: in both categories they gain space efficiency but still have to decompress the data in order to perform any computation on them. In the case of [9], [10], a comparison is possible with our experimental results for the space occupancy, as the input data format they consider can be used in principle also for our input data, and they both refer to plain bzip2 outcome as we do. The other works only refer to the original size, and consider data types that constitute a subset of ours, therefore a quantitative comparison of the results is not directly possible.

V. Conclusion

With the growth of the telecommunication networks and of their user base, the need for efficiently collecting, transferring, processing and storing network monitoring data continuously poses new challenges. In this paper, we focused on the storage and subsequent data mining phases, and presented a technique that stores network measurement data in a representation format that satisfies two main objectives at the same time: the efficient utilization of storage space, and the possibility to perform a number of operations in a computationally convenient way and directly on the transformed data; these high valuable properties are traded-off with a controlled loss of accuracy. The experimental evaluation on data derived from a real traffic trace shows that a compression down to about 20 percent of the original size can be achieved with relative error in the order of 5 percent of the true value (for the numerical field), and with a more aggressive approximation, compression level greater of *bzip2*'s is reached while preserving the property of being directly searchable and processable.

The experiments showed also space for further improvements: given the accuracy requirements of an application for the different data fields, the accuracy loss could be optimally distributed on the different fields (specially non-numerical ones, such as URLs or IDs). On the other side, an optimized choice of the precision of the matrix elements (in terms of amount of bits assigned for each field) would easily improve the compression ratio. Current work is focused on investigating the exploitation of the preprocessing-normalization stage in order to meet different accuracy constraints (according to application needs), as well as on assessing the sensitiveness of the performances to the variation of the URL filtering threshold. In future works we plan to evaluate the trade-off in accuracy vs compression in relation to different precision of matrix element representation, and also to evaluate the accuracy of the technique after the data-mining operations, in relation to different use cases.

References

[1] S. Agrawal, C. N. Kanthi, K. V. M. Naidu, J. Ramamirtham, R. Rastogi, S. Satkin, and A. Srinivasan, "Monitoring infrastructure for converged networks and services," *Bell Labs Technical Journal*, vol. 12, no. 2, pp. 63–77, 2007.

[2] M. Peuhkuri, "A Method to Compress and Anonymize Packet Traces," in *In Proceedings of the First ACM Internet Measurement Workshop*, 2001, pp. 257–261.

[3] P. B. Ros, G. Iannaccone, J. S. Cuxart, D. A. López, and J. S. Pareta, "Load shedding in network monitoring applications," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2007.

[4] M. Munk, J. Kapusta, and P. Svec, "Data preprocessing evaluation for web log mining: reconstruction of activities of a web visitor," *Procedia Computer Science*, vol. 1, no. 1, pp. 2273–2280, 2010.

[5] J. Zujovic and O. G. Guleryuz, "Complexity regularized pattern matching," in *Proceedings of the 16th IEEE international conference on Image processing*, ser. ICIP'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 1869–1872.

[6] Y. Saad, "SPARSKIT: A basic tool kit for sparse matrix computations," Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, Tech. Rep. RIACS-90-20, 1990.

[7] Y. Liu, D. Towsley, T. Ye, and J. Bolot, "An information-theoretic approach to network monitoring and measurement," in *In Proc. of IMC*, 2005.

[8] S. Vucetic, "A Fast Algorithm for Lossless Compression of Data Tables by Reordering," *Data Compression Conference*, vol. 0, pp. 469+, 2006.

[9] P. Skibiński and J. Swacha, "Fast and efficient log file compression," in *CEUR Workshop Proceedings of 11th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, 2007.

[10] F. Otten, B. Irwin, and H. Thinyane, "Evaluating text preprocessing to improve compression on maillogs," in *SAICSIT '09: Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*. New York, NY, USA: ACM, 2009, pp. 44–53.

[11] K. K. Arsa and S. Sanguanpong, "In-memory URL Compression," in *National Computer Science and Engineering Conference, Chiang Mai, Thailand*, Nov. 2001, pp. 425–428.

[12] G. Reeves, J. Liu, S. Nath, and F. Zhao, "Managing massive time series streams with multi-scale compressed trickles," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 97–108, 2009.