AUTOMATYKA • 2007 • Tom 11 • Zeszyt 3

Szymon Grabowski*, Sebastian Deorowicz**

Web Log Compression

1. Introduction

Surprisingly perhaps, in recent years plain text has become a prominent medium for data conveyance and storage. It is enough to mention the XML format and web languages (HTML, XHTML, CSS, web scripts etc.) to easily support this claim, but a more complete list should also include DNA and protein sequence databases, mail folders, plain text newsgroup archives, IRC archives, and so on. Human-readable textual data are easy to analyze, edit, extract snippets from, etc. It is also easier to find and fix occasional errors in textual rather than in binary form. An interesting feature of "texts" of the mentioned kinds, however, is their redundancy, typically much greater than the redundancy of natural language texts, e.g. fiction books with no markup. Redundancy is obviously harmful as it increases the costs of data transmission and storage; what is less obvious perhaps is that it can also slow down query handling. Dealing with redundant data may require substantial amounts of main memory, which can pose trouble in the notoriously multitasking and multiuser systems.

To overcome the verbosity of textual data, compression techniques can, of course, be applied. In fact, the number of published papers dedicated to specialized XML compression only up to this moment (May 2007) is about 50 (according to a thorough bibliography listed at http://www.ucalgary.ca/~grleight/research/xml-comp.html), and compression of some other data-oriented text formats has also been considered in the literature. It should be stressed that specialized methods, even if limited to text preprocessing before running a general-purpose compressor, can achieve compression ratios significantly better than universal compression algorithms, at more or less retained (and sometimes decreased) computational requirements for the process of data encoding and decoding [5].

In this paper we point out for the need of compressing log data: a rather vague category of files documenting human and machine activity. Many log types can be met in everyday practice: database operation logs, file system access logs, installation logs, etc. Among the most important ones we should definitely mention *web logs*, storing page requests at a given web server. Logging the activity at popular sites can easily add even hundreds of megabytes a day, which needs disk space, makes log data analysis and searches slow and cumbersome etc. Here is where, we believe, compression should enter the stage.

* Katedra Informatyki Stosowanej, Politechnika Łódzka w Łodzi
** Instytut Informatyki, Politechnika Śląska w Gliwicach

We assume that in many scenarios queries or log data analyses are not performed often enough to make *queriable* compression necessary. Our compression techniques are devised for succinct storage and efficient backuping. Prior to handling any queries, the log archive must be decompressed. This is a disadvantage of course, but on the other hand, non-queriable compression algorithms enable reaching better compression ratios and are simpler. We show that it is often possible to compress log data 40 or even 80 times, preserving very fast decompression. A side goal of the current work is to stress on how inappropriate the widely used (also in log storage and analysis systems) Deflate method is, if the data to compress are typical large log files.

2. Redundancy sources in web logs

Typically, web logs have regular structure. Even across different web server log formats (Apache, IIS, etc.) we can easily track down common characteristics. First, we assume a single event (page request) is recorded in one line and each line corresponds to a single event only. Second, several pattern types are very frequent: IP addresses, timestamps (in some chosen format), URL's. Third, there are (long) text sequences which occur many times, e.g. clients' web browser ID strings, clients' OS platform names, names of frequently accessed files, IP's of those users who frequently visit a given site, or request many files in a single session (which is almost always the case). Fourth, there is a strong spatial correlation of log entries: successive lines tend to store requests from the same user, and thus their IP addresses and the client machine information will repeat. Also, the timestamps of the successive lines are often very similar, which suggests differential encoding as an effective means to squeeze out the redundancy. Fifth, web log files are similar to tables in a relational database: lines are composed of fields (attributes) in a fixed order, typically separated with blank spaces. The knowledge of a given field domain (built into a specialized compressor or inferred during the compression process) is certainly beneficial for both the compression effectiveness and the compression efficiency. Sixth, like in tables of real-world databases, there exist strong correlations across fields, e.g. between user's IP and his web browser (a subsequent request from the same IP, even if thousands of lines farther in the log file, is very likely to be followed by the "old" web browser ID string). Seventh, logs are usually in plain ASCII, i.e. the character values do not exceed 127 (also, most of the symbols with ASCII codes below 32 are unused). The unused symbols could be utilized for cheap substitution of frequent sequences.

3. Related work

As mentioned in Section 1, most open-source and commercial utilities for archiving and analyzing log data use gzip (Deflate) compression, while some make use of a newer and stronger compressor bzip2 (Web Log Mixer is an example). We know about only one non-research application, SafeLog (http://www.solution-soft.com/safelog.shtml), incorporating a proprietary compression format, which is claimed to produce up to twice smaller log archives than gzip. No details on the algorithm are disclosed.

Differentiated Semantic Log Compression (DSLC) presented by Rácz and Lukács in [4] probably bears significant similarity to the algorithm we present in the current paper, but unfortunately the authors were not explicit about some details. DSLC works on the level of web log lines, uses specific treatment for each individual field, replaces frequent field values with references to a semi-static dictionary, and at the end runs a general-purpose compressor. The results cited in the original work are quite impressive, but, according to [7], the Rácz and Lukács scheme "works well only on huge log files (over 1 GB) and it requires human assistance before the compression, on average about two weeks for a specific log file". Moreover, it is unclear from the original paper[1] which of the mentioned ideas have already been implemented and which are only planned, also at times the authors direct a reader to an extended version of their paper (which however is not available as of May 23, 2007), which we found very confusing.

A compression scheme for encoding the user activity logs in their client-side monitoring system was employed by Kulpa et al [3]. The algorithm had to be simple and fast (it is implemented in JavaScript) and is intended to work on small log chunks; it comprises string substitution and differential date/time encoding techniques. From those reasons, the obtained compression is mediocre.

Very recently, Skibiński and Swacha [7] proposed a couple of simple preprocessing variants intended to facilitate further compression of log files from various applications. Since their goal was broader than ours, they used more general means of transforming data. Namely, they proposed five variants, where the simplest one merely encodes each line with reference to the previous line, storing the length of the longest match on a single byte (with aid of symbols over 127 in ASCII), followed by the mismatching subsequence copied verbatim, until the nearest field end, where again the longest match in the previous line for the corresponding field is sought for. The next two variants are more flexible in choosing the reference line which helps especially for log types where not all lines have identical structure (e.g. MySQL database logs in the experiments in the cited work). Fourth variant adds a dictionary substitution for words found in a prepass (an idea used earlier, e.g. in [6], for plain text compression), and the fifth variant extends the previous one with compact encoding of the following patterns: numbers, dates, times and IP addresses. In their experiments, the transformed log files compressed then with the default zip algorithm, i.e. Deflate, were on average shorter by 37% than the non-preprocessed files submitted to zip. Significant improvements (on the order of 20%) have also been noticed when stronger back-end compression algorithms (LZMA, PPMVC) were used.

4. Apache Web log format

The default order of fields in a Apache web log is fixed. We list them below[2]. The field numbers are added only for reference in the latter sections).

[1] We mean the 10-page version, obtained via personal communication, not the 1-page DCC conference poster.

[2] http://www.jafsoft.com/searchengines/log_sample.html

#0 – visitor's IP address,
#1, #2 – username etc. Set to "– –", unless accessing password-protected content,
#3 – timestamp of the visit (date, time, time zone),
#4 – access request (e.g. "GET /full/j35.jpg HTTP/1.0"),
#5 – result status code (200 – success, a number of error codes exist as well),
#6 – byte transferred (usually the requested file size; less means a failed or partial download),
#7 – referrer URL (e.g. "http://www.fighter-planes.com/data6070.htm"). This is the page the visitor was on when he clicked to move to the current location,
#8 – user agent ID string (e.g., "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"). Usually a web browser, but could be a web robot, a link checker etc.

An Apache server administrator may configure the log format with an entry of the conf/httpd.conf file. For example, it may happen that fields #7 and #8 are missing, and it was the case of our Access.log file used for the experiments.

5. Our algorithms

We start with a simple algorithm which reorders the data in a field-by-field manner, without taking any cross-redundancy between fields into account, and then we examine the issue how to find out which fields should be grouped together to improve compression.

5.1. Every man for himself

Transposing a relational database table is a well-known idea attempting to increase compression [2]. The successive attribute values are then located adjacently, and each field can then be compressed separately, as (we assume) different fields are unlikely to share statistical properties. If this is indeed the case, such a disentanglement of data should bring significant benefits: it is easy to perform dictionary substitution on individual fields, limiting the scope of the compression model to what is relevant only results in lower computational and memory requirements, recency effects (e.g. runs of occurrences of the same field value) can be conveniently exploited, and so on. An alternative approach – compressing the fields, each with its separate model and using appropriate semantic knowledge – has the benefit of being on-line but requires housing several models at the same time, i.e. needs more memory to work.

We decided to apply this idea for web log compression. More concretely, what we do in this basic file preprocessing variant is presented in the following list:

- We group the file content field by field, in the order of field occurrences in a row.
- A field containing timestamps (e.g. 03/Feb/2003:03:07:23 +0100) is identified and the differences between its successive values are calculated and encoded. In practice, those differences are often in 0.254 range, and store one byte in such a case. The value 255 for the timestamp field serves as an escape for a larger (or negative, although it never happens in real, non-modified logs) difference, which is encoded on the following four bytes.

- A field containing IP addresses only is identified; the next step will not be applied for this field.
- If the number of distinct prefixes for some field (except for the IP field) is not more than 16, and also the number of common suffixes in this field is not more than 16, they are chopped off and sent to two extra prefix streams and two extra suffix streams: one of a pair is merely the prefix (suffix) vocabulary, the other holds the prefix (suffix) indexes, item by item. By prefixes (suffixes), we understand the starting (ending) characters up to the first (last) whitespace in a field. It often happens that the prefix/suffix vocabularies are empty. For example, they are empty if a given field contains no spaces. The prefix and suffix index streams are order-1 arithmetically compressed, and no back-end compression will later be applied to them.
- The *move-to-front* transform [1] is applied on the remaining factors of the fields; the idea is to explore a recency effect typical for many fields, which means, in plain words, that recently occurring values are more likely to occur again than novel values. The move-to-front transform encodes a given value v as the number of *unique* values between the previous and the current occurrence of v. In our solution, for each field value v we send into the first stream the either 0 (which means v occurred just in the previous row), or 1 (v appeared before), or 2 (v is new and never appeared before). Then, if we encoded 1, we put into the second stream the MTF code, i.e. the number of unique values since last occurrence of v. If we encoded 2, we put into the third stream the value v as is. We found experimentally that high MTF values make the compression ratio worse, so if the number of unique values since last appearance of v is larger than 256, we treat v as a never-appeared-before value and encode both 2 and v. MTF codes and the stream of ternary flags are order-1 arithmetically compressed.
- Each value in the IP field is encoded on 4 bytes, no separators used.

5.2. Merging correlated fields

The algorithm from the previous section is simple but ignores the fact that some fields may be correlated. In fact, some strong correlations between fields are typical in Apache web logs. We identified the correlation between a file name (with its path) and its size, and the client's IP and his web agent ID string. Identical values in one of those fields are likely to be followed by identical values in the other fields in the corresponding rows, hence improving greatly the overall compression of those fields. Please note, however, that the word "likely" should not rather be replaced by "certain": logs usually store visits to a server's web site over quite a period of time, and thus some users might have upgraded their local platform and browser between successive visits, some files might have been edited and their sizes changed, and so on.

The variant we propose makes use of the log file format knowledge and explicitly assumes that the pairs of fields: (#0, #8) and (#4, #6) should be merged just after truncating the affixes and before any further processing step. Nothing else from the previous variant is changed. Of course, a compression-directed correlation analysis for all field pairs (or, even better, all field subsets) would be much more desirable, but our preliminary efforts suggest this is not an easy task. We therefore postpone it as a future work subject.

6. Experimental results

We implemented our algorithms in Python 2.5, all tests were run on an Athlon64 3000+ machine, equipped with 512 MB RAM and running under Windows XP SP2 operating system. Due to a script nature of our implementation, we do not provide any timings. Still, if implemented in a compiled language (e.g., C++), the algorithms should be fast enough in practice, especially in the decompression. For order-1 statistical encoding, applied in some stages of our transform, we used arhangel, v1.40a2, an archiver which can be downloaded from http://www.geocities.com/SiliconValley/Lab/6606/arhangel.htm.

For comparison, we were able to get only one specialized log compressor, logpack [7]. It works on arbitrary logs (not only web logs). Logpack is able to make use of built-in back-end compression libraries (zlib and others), but for test compatibility, we ran it with the -l0 switch for preprocessing only. Its output was then submitted to an external compressor, exactly like we did when testing our algorithms.

To measure how well our algorithms compete in their domain with respected universal compression methods, we chose a few well-known compressors for a comparison:

- gzip, v1.2.3, implementing the Deflate method from the LZ77 family,
- 7z, v4.45 beta, using its default algorithm, LZMA, a modern representative of the LZ77 family,
- bzip2, v1.0.2, a compressor based on the Burrows-Wheeler transform,
- PPMd, var. J, a efficient implementation of the PPM algorithm.

Table 1
Compression results in bits per character (bpc). Second top row holds the original file sizes in bytes

Log file →	Access	FP	Latexeditor	Netaccess	average
raw file (in bytes)	17 517 060	20 617 071	30 381 282	3 105 150	–
gzip	0.417	0.564	0.390	0.307	0.420
bzip2	0.256	0.281	0.212	0.168	0.229
LZMA	0.357	0.360	0.274	0.294	0.321
PPMd -o6 -m192	0.201	0.254	0.227	0.162	0.211
PPMd -o16 -m192	0.173	0.226	0.175	0.131	0.176
logpack + gzip	0.270	0.334	0.236	0.150	0.248
logpack + bzip2	0.185	0.244	0.157	0.124	0.178
logpack + LZMA	0.222	0.252	0.169	0.127	0.193
logpack + PPMd -o6	0.140	0.210	0.139	0.118	0.152
logpack + PPMd -o16	0.131	0.204	0.128	0.109	0.143
our, v1 + gzip	0.238	0.164	0.070	0.111	0.146
our, v1 + bzip2	0.178	0.150	0.067	0.102	0.124
our, v1 + LZMA	0.212	0.155	0.069	0.110	0.137
our, v1 + PPMd -o6	0.145	0.147	0.066	0.102	0.115
our, v1 + PPMd -o16	0.135	0.145	0.066	0.102	0.112
our, v2 + gzip	0.239	0.144	0.064	0.141	0.147
our, v2 + bzip2	0.169	0.115	0.053	0.102	0.110
our, v2 + LZMA	0.209	0.129	0.059	0.122	0.130
our, v2 + PPMd -o6	0.138	0.115	0.052	0.100	0.101
our, v2 + PPMd -o16	0.127	0.110	0.051	0.098	0.097

Default settings of those compressors were used, with the exception of PPMd, which was tested twice: with -o6 -m192 and -o16 -m192 switches, respectively. The -o parameter sets the maximum PPM model order. References to all the general-purpose compressors listed above can be found at http://www.maximumcompression.com.

The test set comprises four files: alas, web site administrators are reluctant to make the logs public, due to obvious reasons, therefore it is really hard to find on the web such kind of material, of reasonable quality (large real web logs). We obtained privately three files for the collection (Access.log, Latexeditor.log, Netaccess.log), while FP.log can be downloaded from http://www.maximumcompression.com/data/files/log-test.rar, and – interestingly – is part of a corpus for measuring compression performance of many compressors and archivers. The file sizes span from 3 MB to 30 MB, and are given in detail in Table 1.

As can be seen, our transform (variant 2) shortens gzip archives by 65% on average, and bzip2 archives by 52% on average. Also with the other compressors the achieved improvements are very significant.

7. Conclusions and future work

We presented two relatively simple off-line preprocessing schemes for web log compression. Our implementation works with the nowadays most popular web log format, Apache, but the entry fields occurring there are typical for other formats (e.g., IIS) too. The first variant treats each field separately, and – quite surprisingly – even this approach helps a lot. The biggest improvement, as expected, was achieved in combination with gzip, the weakest (but also most widely used) among the tested compressors: 3 out of 4 files were shrunk to about one third (or less) of the plain gzip archive size! Our advantage over logpack, a specialized log compressor, is also impressive. Still, sometimes it disappears when the strongest of the tested compressors, PPMd, comes into play.

Our experiments show how redundant log files are. The average ratio for gzip backend is 0.147 bpc, which means that the log files are reduced over 54 times! When the backend is PPMd, the logs are reduced even 82 times. Bzip2 as the backend is not much worse, but its speed is clearly inferior, especially in the compression. When high decoding speed has priority, the best choice may be LZMA (unfortunately, it is quite slow in the compression phase).

It is clear from the results that web logs vary significantly in redundancy. Interestingly, after the preprocessing and backend compression, the difference in compression ratio gets even larger. This can be explained by some kinds of redundancy in logs which general-purpose compressors cannot effectively cope with. An example of such redundancy can be the similar (but not well handled by, e.g., gzip) timestamps in the successive lines.

Definitely the main weakness of our algorithms in their current form is their rigid expectations about the input file format. This is one of the main things we are going to improve in the future work: make the scheme flexible enough to work with several, freely configured, log formats, or even better, assume as little as possible about the input and thus be able to efficiently process arbitrary log files, not only web log ones.

Along these lines, we are going to look for a practical heuristic for merging fields before further processing. Our preliminary experiments were unfortunately unfruitful.

Log files tend to contain long repeating sequences, which may be successfully replaced with short tokens. We, however, chose an alternative to replacing words with a semi-static dictionary index: instead, we applied the well-known move-to-front heuristic for whole field values. Still, we are aware we have not fully exploited the dictionary-based approach (large dictionary size, spaceless model etc.), so we may get back to this idea later. Also, it seems that the traditional notion of a "word" in dictionary-based schemes is inappropriate for some web log fields: the set of word separators may be expanded with (e.g.) the symbols '/', '&' and '?'. Some other, minor, improvements are possible as well. Finally, we are going to implement the application from scratch in C++ and then perform also speed measurements.

We thank Przemysław Skibiński for sending us his log compressor logpack and comments to the manuscript.

References

[1] Bentley J.L., Sleator D.D., Tarjan R.E., Wei V.K.: *A locally adaptive data compression scheme.* Communications of ACM, 29(4), 1986, 320–330

[2] Graefe G., Shapiro L.: *Data Compression and Database Performance*. Proceedings of ACM/IEEE-CS Symposium on Applied Computing, Kansas City, MO, 1991

[3] Kulpa A., Swacha J., Budzowski R.: *Script-based system for monitoring client-side activity*. [in:] Abramowicz, W., Mayr, H. (eds.), Technologies for Business Information Systems. Springer, 2007

[4] Rácz B., Lukács A.: *High density compression of log files*. Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, 2004, 557

[5] Skibiński P., Grabowski Sz., Swacha J.: *Effective asymmetric XML compression*. Submitted to Software–Practice and Experience, 2007

[6] Skibiński P., Grabowski Sz., Deorowicz S.: *Revisiting dictionary-based compression*. Software–Practice and Experience, 35(15), 2005, 1455–1476

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