Fast and efficient log file compression

Przemysław Skibiński[1] and Jakub Swacha[2]

[1] University of Wrocław, Institute of Computer Science,
Joliot-Curie 15, 50-383 Wrocław, Poland
inikep@ii.uni.wroc.pl
[2] The Szczecin University, Institute of Information Technology in Management,
Mickiewicza 64, 71-101 Szczecin, Poland
jakubs@uoo.univ.szczecin.pl

Abstract. Contemporary information systems are replete with log files, created in multiple places (e.g., network servers, database management systems, user monitoring applications, system services and utilities) for multiple purposes (e.g., maintenance, security issues, traffic analysis, legal requirements, software debugging, customer management, user interface usability studies).
Log files in complex systems may quickly grow to huge sizes. Often, they must be kept for long periods of time. For reasons of convenience and storage economy, log files should be compressed. However, most of the available log file compression tools use general-purpose algorithms (e.g., Deflate) which do not take advantage of redundancy specific for log files.
In this paper a specialized log file compression scheme is described in five variants, differing in complexity and attained compression ratios. The proposed scheme introduces a log file transform whose output is much better compressible with general-purpose algorithms than original data. Using the fast Deflate algorithm, the transformed log files were, on average, 36.6% shorter than the original files compressed with gzip (employing the same algorithm). Using the slower PPMVC algorithm, the transformed files were 62% shorter than the original files compressed with gzip, and 41% shorter than the original files compressed with bzip2.

Keywords: log files, log file storage, text compression, compression algorithms

1 Introduction

1.1 Motivation

Many of the actions performed in computer-based information systems leave a track. Users may not like it at all, but there are plenty of reasons for which such records are valuable for system administrators or vendors – such as maintenance, security issues,

Y. Ioannidis, B. Novikov and B. Rachev (Eds.): Local Proceedings of ADBIS 2007, pp. 56-69

traffic analysis, legal requirements, software debugging, customer management, or user interface usability studies.

Relevant data can be collected in multiple places (e.g., network servers, database management systems, user monitoring applications, system services and utilities). In many environments, tracked events can happen very often. As a result, there is a huge amount of data produced this way every day. And often it is necessary to store them for a long period of time.

Regardless of the type of recorded events, for reasons of simplicity and convenience, they are usually stored in plain text log files. Both the content type and the storage format suggest that it is possible to significantly reduce the size of log files through lossless data compression, especially if specialized algorithm was used. The smaller, compressed files have the advantages of being easier to handle and saving storage space.

1.2 Problem setting

There is plenty of redundancy to be exploited in typical log files. Log files are composed of lines, and the lines are composed of tokens. The lines are separated by end-of-line marks, whereas the tokens are separated by spaces.

Some tokens repeat frequently, whereas in case of others (e.g., dates, times, URL or IP addresses) only the token format is globally repetitive. However, even in this case there is high correlation between tokens in adjoining lines (e.g., the same date appears in multiple lines).

In most log files, lines have fixed structure, a repeatable sequence of token types and separators. Some line structure elements may be in relationship with same or different elements of another line (e.g., increasing index number).

1.3 Approach

We propose a multi-tiered log file compression solution. Every of the three tiers addresses one notion of redundancy. The first tier handles the resemblance between neighboring lines. The second tier handles the global repetitiveness of tokens and token formats. The third tier is general-purpose compressor which handles all the redundancy left after the previous stages.

The tiers are not only optional, but each of them is designed in several variants differing in required processing time and obtained compression ratio. This way users with different requirements can find combinations which suit them best. We propose five processing schemes for reasonable ratios of compression time to log file size reduction.

1.4 Contributions

Our main contribution is the log file transform aimed at producing output which is much better compressible with general-purpose algorithms than the original data. The transform can be used to compress any kind of textual log files, regardless of their

size and source. The core transform supports streamed compression. It is extended with an adaptation of our word replacement transform for additional improvement of compression ratio. The transform algorithm has low computational requirements. We investigate several variants of log compression schemes looking for the best trade-off between attained compression ratios and required processing time.

2 Related work

As with the growth of log file sizes their management and storage has become burdensome, many commercial and open-source utilities were developed to fix this issue. Most of them were aimed at compression of web server logs, e.g., logrotate, rotatelogs, httplog, IIS Log Archiver, Web Log Mixer, SafeLog**,** and almost each of them **(**with the exception of SafeLog) uses general-purpose compression algorithm (mostly gzip, rarely bzip2).

In 2004 Rácz and Lukács presented DSLC (Differentiated Semantic Log Compression), a generalized scheme for web log files compression. The scheme features parsing log lines into separate fields, and a set of pipelined transformations chosen for every kind of data field, including calculation of computed fields, stripping unnecessary information, replacement of strings with dictionary indexes, semantic-specific encoding, and, finally, general-purpose compression [7]. The DSLC is claimed to improve general-purpose compression algorithms efficiency on web logs up to a factor of ten. Still, it 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 [8].

In 2006 Balakrishnan and Sahoo proposed a scheme for compression of system logs of the IBM Blue Gene /L supercomputer. The scheme consists of preprocessing stage (featuring differential coding) and general-purpose compression [1]. The measured improvement of compression ratio was 28.3%. Of course, the application of this scheme is very limited.

Recently, Kulpa, Swacha, and Budzowski developed a scheme for encoding the user activity logs generated by monitoring system [4]. It features string substitution and differential date and time encoding. On average, a 71.9% gain in log size has been measured. This scheme is intended for small log chunks, it has very low complexity and it is implemented client-side in JavaScript, as its sole purpose is to limit the network bandwidth required to transmit user logs from client to server. Therefore, its results are much inferior to those of complex server-side log compressors.

3 Log file transform for improved compression

3.1 The core transform

Log files are plain text files in which every line corresponds to a single logged event description. The lines are separated by end-of-line marks. Each event description consists of at least several tokens, separated by spaces. A token may be a date, hour, IP address, or any string of characters being a part of event description.

In typical log files the neighboring lines are very similar, not only in their structure, but also in their content. The proposed transform takes advantage of this fact by replacing the tokens of a new line with references to the previous line. There may be a row of more than one token that appears in two successive lines, and the tokens in the neighboring lines can have common prefixes but different suffixes (the opposite is possible, but far less frequent in typical log files). For this two reasons, the replacement is done on byte level, not token level.

The actual algorithm works as follows: starting from the beginning of new line, its contents are compared to those of the previous line. The sequence of matching characters is replaced with a single value denoting the length of the sequence. Then, starting with the first unmatched character, until the next space (or an end-of-line mark, if there are no more spaces in the line), the characters are simply copied to the output. The match position in the previous line is also moved to the next space. The matching/replacement is repeated as long as there are characters in the line.

The length l of the matching sequence of characters is encoded as a single byte with value 128+l, for every l smaller than 127, or a sequence of m bytes with value 255 followed by a single byte with value 128+n, for every l not smaller than 127, where $l = 127*m + n$. The byte value range of 128-255 is often unused in logs, however, if such a value is encountered, it is simply preceded with an escape flag (127). This way the transform is fully reversible.

Consider the following input example:

12.222.17.217 - - [03/Feb/2003:03:08:13 +0100] "GET /jettop.htm HTTP/1.1"
12.222.17.217 - - [03/Feb/2003:03:08:14 +0100] "GET /jetstart.htm HTTP/1.1"

Actually, these are beginnings of two neighboring lines from the *fp.log* test file. Assume the upper line is the previously compressed line, and the lower line is the one to be compressed now. The algorithm starts matching characters from the line beginning, there are 38 consecutive characters which appear in both lines, so they are replaced with value 166 (128+38). Then we look for the next space, storing the single unmatched character that precedes it (‘4’). Now the data to process and reference line looks as follows:

+0100] "GET /jettop.htm HTTP/1.1"
+0100] "GET /jetstart.htm HTTP/1.1"

This time we have 16 consecutive characters which appear in both lines, so they are replaced with value 144 (128+16). Again, we look for the next space, storing the string of unmatched characters that precede it (‘start.htm’, notice that matching suffixes are not exploited at this stage). Now the data to process and reference line looks as follows:

HTTP/1.1"
HTTP/1.1"

This time we have 9 consecutive characters which appear in both lines, so they are replaced with value 137 (128+9). The full line has been processed. The input was:

12.222.17.217 - - [03/Feb/2003:03:08:14 +0100] "GET /jetstart.htm HTTP/1.1"

And the output is (round brackets represent bytes with specified values):

(166)4(144)start.htm(137)

Notice that single spaces are not existent in the output as they can be automatically reinserted on decompression.

The processed output can then be passed to a back-end general-purpose compressor, such as gzip, LZMA, or PPMVC.

We shall refer to the aforedescribed algorithm as the variant 1 of the proposed transform, the one which is the simplest and fastest, but also the least effective.

3.2 Transform variant 2

In practice, a single log often records events generated by different actors (users, services, clients). Also, log lines may belong to more than one structural type. As a result, similar lines are not always blocked, but they are intermixed with lines differing in content or structure.

The second variant of the transform fixes this issue by using as a reference not a single previous line, but a block of them (16 lines by default; smaller value hurts the compression ratio, higher value hurts the compression time). For a new line, the block is searched for the line that returns the longest initial match (i.e., starting from the line beginning). This line is then used a reference line instead of the previous one. The search affects the compression time, but the decompression time is almost unaffected. The index of the line selected from the block is encoded as a single byte at the beginning of every line (128 for the previous line, 129 for the line before previous, and so on).

Consider the following input example:

12.222.17.217 - - [03/Feb/2003:03:08:52 +0100] "GET /thumbn/f8f_r.jpg
172.159.188.78 - - [03/Feb/2003:03:08:52 +0100] "GET /favicon.ico
12.222.17.217 - - [03/Feb/2003:03:08:52 +0100] "GET /thumbn/mig15_r.jpg

Again, these are beginnings of three neighboring lines from the *fp.log* test file. Assume the two upper lines are the previously compressed lines, and the lower line is the one to be compressed now.

Notice that the middle line is an 'intruder' which precludes replacement of IP address and directory name using variant 1 of the transform, which would produce following output (round brackets represent bytes with specified values):

12.222.17.217(167)thumbn/mig15_r.jpg

But variant 2 handles the line better, producing following output:

(129)(188)mig15_r.jpg

3.3 Transform variant 3

Sometimes a buffer of 16 lines may be not enough, i.e., the same pattern may reappear after a long period of different lines. As similar lines tend to appear together, it is possible that only few tokens appear at beginning of all the 'intruder' lines. Transform variant 3 makes use of this fact by storing the recent 16 lines with different beginning tokens. If a line starting with a token already on the list is encountered, it is appended, but the old line with the same token has to be removed from the list. This way, a line separated by thousands others can be referenced only provided the other lines have no more than 15 types of tokens at their beginnings.

Compared to variant 2, instead of a block of previous lines there is a *move-to-front* list of lines with different initial tokens. Therefore, the selected line index addresses the list, not the input log file. As a result, the list has to be managed by both the encoder and decoder. In case of compression, the list management time is very close to the variant 2's time of searching the block for the best match. But in case of decompression, it noticeably slows down the processing.

3.4 Transform variant 4

The three variants described so far are *on-line* schemes, i.e., they do not require the log to be complete before the start of its compression, and they accept a stream as an input. The following two variants are *off-line* schemes, i.e., they require the log to be complete before the start of its compression, and they only accept a file as an input. This drawback is compensated with significantly improved compression ratio.

Variants 1/2/3 addressed the local redundancy, whereas variant 4 handles words which repeat frequently throughout the entire log. It features word dictionary containing the most frequent words in the log. As it is impossible to have such dictionary predefined for any kind of log file (though it is possible to create a single dictionary for a set of files of the same type), the dictionary has to be formed in an additional pass. This is what makes the variant off-line.

The first stage of variant 4 consists of performing transform variant 3, parsing its output into words, and calculating the frequency of each word. Notice that the words

are not the same as the tokens of variants 1/2/3, as a word can only be either a single byte or a sequence of bytes from value range 65..90, 97..122, 128..255 (all ASCII letters and non-7-bit ASCII characters). Only the words whose frequency exceeds a threshold value f_{min} are included in the dictionary. The optimal value of f_{min} depends on the back-end general-purpose compressor used. Based on many experiments, it has been estimated at 64 for the highly effective PPMVC algorithm, and 6 for the remaining, less effective algorithms (including gzip and LZMA).

The dictionary has an upper limit of size (by default, 2 MB of memory). If the dictionary reaches that limit, it is frozen, i.e., the counters of already included words can be incremented but new words cannot be added (though this is a rare case in practice).

During the second pass, the words included in the dictionary are replaced with their respective indexes. The dictionary is sorted in descending order based on word frequency, therefore frequent words have smaller index values than the rare ones. Every index is encoded on 1, 2, 3 or 4 bytes. The indexes are encoded using byte values unused in the original input file. The related work of the first author on natural language text compression [10] shows that different index encoding schemes should be used for different back-end compression algorithms. Variant 4 uses following encodings:

1. For gzip, three disjoint byte value ranges are used: x, y, and z. The first byte of the codeword belongs either to x, y, or z depending on the codeword length (respectively 1, 2, or 3 bytes). The remaining bytes (if any) may have any value (from 0 to 255). Thus the are $x + y*256 + z*256*256$ available codewords.
2. For LZMA, two disjoint byte value ranges are used: x and y. A byte belonging to x denotes codeword beginning, and a byte belonging to y denotes codeword continuation. Thus the are $x + x*y + x*y*y$ available codewords.
3. For PPMVC, four disjoint byte value ranges are used: w, x, y, and z. A byte belonging to w denotes codeword beginning, x – first continuation, y – second continuation, z – third continuation. Thus there are $w + w*x + w*x*y + w*x*y*z$ available codewords.

3.5 Transform variant 5

There are types of tokens which can be encoded shorter using binary instead of text encoding typical for log files. These include numbers, dates, times, and IP addresses. Variant 5 is variant 4 extended with special handling of this kind of tokens. They are replaced with flags denoting their type, whereas their actual value is encoded densely in a separate container. Every data type has its own container. The containers are appended to the main output file at the end of compression or when their size exceeds upper memory usage limit.

Numbers are replaced with a byte denoting the length of the number (1, 2, 3, or 4 bytes) encoded in binary in respective container. As only 256^4 integers can be stored on four bytes, digit sequences representing larger numbers are simply split into several shorter ones (a rare case in practice). If the digit sequence starts with one or more zeroes, the initial zeroes are left intact.

Dates in YYYY-MM-DD (e.g. "2007-03-31", Y for year, M for month, and D for day) and DD/MMM/YYYY (e.g. "31/Mar/2007") formats are replaced with a byte denoting date format and encoded as a two bytes long integer whose value is the difference in days from 1977-01-01. To simplify the calculations we assume each month to have 31 days. If the difference with the previous date is smaller than 256, another flag is used and the date is encoded as a single byte whose value is the difference in days from the previous date.

Times in HH:MM:SS (e.g. "23:30:59", H for hour, M for minute, and S for second) format are replaced with a flag and encoded as sequence of three bytes representing respectively: hours, minutes, and seconds.

IP addresses are replaced with a flag and encoded as sequence of four bytes representing IP octets. Non-standard IP addresses (with initial zeroes, or spaces between octets) are not replaced.

4 Experimental evaluation

4.1 Tested compression schemes

The main object of the evaluation is the proposed transform implemented by the first author in the LogPack program, written in C++ and compiled with Microsoft Visual C++ 6.0. LogPack offers five modes of operation corresponding to five transform variants; we shall denote them as LP*x*, where *x* is the transform variant, e.g., LP1 stands for the first (simplest) variant.

LogPack has embedded three back-end compression algorithms: gzip, LZMA, and PPMVC. Of these, only gzip is aimed at on-line compression, as it is the simplest and fastest scheme [2]. The remaining two are for highly effective off-line compression.

LZMA (better known as the default mode of the 7zip utility [6]) features improved parsing and larger dictionary buffer. It offers high compression ratio, but the cost is very slow compression (decompression is only slightly slower than gzip's).

PPMVC is a sophisticated prediction-based algorithm [11]. It offers compression ratio even higher than LZMA, and faster compression time. The PPMVC's drawback is that its decompression time is very close to its compression time, which means it is several times longer than gzip's or LZMA's decompression times. In the tests, PPMVC was used with prediction model order 8, and 64 MB of model size.

The back-end compression algorithms were tested with default compression options which generally give a reasonable trade-off between obtained compression ratio and required compression time.

All the five transform variants were tested with gzip. LZMA and PPMVC were tested only with the most complex, fifth variant. The log files were also compressed with the three back-end algorithms alone to show the improvement from applying the proposed transform. For a comparison, results for popular compression tools lrzip [3] and bzip2 [9] were also included.

4.2 Test files and environment

The proposed transform was not designed for a specific kind of log file, therefore different log files were included in the test suite. The only well-known file is *FP*, an Apache server log from the Maximum Compression corpus [5]. Three files were downloaded from public servers: *2005access*, *RMAccess* (web logs) and *Cernlib2005* (build log). The remaining three files were obtained from the first author's personal computer: *Mail* (from the Exim program), *Syslog* (Linux system log) and *MYSQL* (database log). File sizes vary from 3 MB (*Mail*) to 111 MB (*RMAccess*). Snippets of the contents of each test file can be found in the appendix.

The test machine was an AMD Sempron 2200+ system with 512 MB memory and Seagate 160 GB ATA drive, running Microsoft Windows 98.

4.3 Experimental results and their discussion

Tables 1 and 2 show the measured experimental results. Table 1 contains compression results of the test files using gzip only, and the five variants of the proposed transform combined with gzip. For each program and file a bitrate is given in output bits per input character, hence the smaller the values, the better. Below there is an average bitrate computed for all the seven test files, and the average improvement compared to the general purpose algorithm result (gzip in Table 1). At the bottom, the measured compression and decompression times are given in seconds.

Table 1. Log file compression using gzip as back-end algorithm

	gzip	LP1+gzip	LP2+gzip	LP3+gzip	LP4+gzip	LP5+gzip
2005access	0.445	0.209	0.215	0.215	0.212	0.171
Cernlib2005	0.276	0.218	0.172	0.171	0.185	0.182
FP	0.563	0.409	0.358	0.352	0.340	0.332
Mail	1.065	0.853	0.859	0.859	0.741	0.720
MYSQL	0.639	0.641	0.435	0.434	0.402	0.298
RMAccess	0.960	0.829	0.819	0.818	0.819	0.819
Syslog	0.386	0.317	0.318	0.318	0.275	0.225
Average	0.619	0.497	0.454	0.452	0.425	0.392
Improvement	–	19.79%	26.71%	26.92%	31.38%	36.61%
Comp. time (s)	16.7	17.1	21.5	22.2	42.1	41.5
Decomp. time (s)	5.8	9.8	11.3	14.3	18.0	18.8

Log files are quite well compressible with gzip. The attained bitrates vary between 1.065 bpc for *Mail* (about 1:8 compression ratio) and 0.276 bpc for *Cernlib2005* (about 1:29 compression ratio).

The effect of applying the first variant of the transform is positive (up to 53% improvement in case of *2005access*) with the sole exception of *MYSQL* (very slight loss). The second variant brings improvement for four test files, the highest for *MYSQL* (32% vs. variant 1). The measured third variant improvement is very small. It may be so that the test suite lacks a file with characteristics allowing LP3 to shine. These three variants allow streamed compression, and they are suited for on-line compression of current log files. One of them can be chosen, depending on processing

time requirements and the type of log file – sometimes LP1 may be the best, but LP2 (or LP3) is suggested as a better scheme on average.

Variants 4 and 5 were designed for efficient long-term storage of log archives. Table 1 shows the results of combining them with gzip: compared to variant 3, the average improvement is 6% for LP4 and 13% for LP5. Two results were not improved: *Cernlib2005* and *RMAccess*. The highest improvement was measured for *Syslog* (about 14% for LP4, and 29% for LP5) and *Mail* (about 14% for LP4, and 16% for LP5). LP5 was better than LP4 for 6 of 7 files (the results for *RMAccess* were the same).

As log rotation and archiving is usually done when user activity in the system is low, the time requirements are not so tight as for on-line log compression. Therefore, slower but more effective back-end compression algorithms can be used instead of gzip. Table 2 shows compression results of the test files using variant 5 of the transform combined with LZMA and PPMVC. For comparison, the results of using only the general-purpose algorithms were included. Besides LZMA and PPMVC, bzip2 and lrzip were also tested. For the latter, the times are not listed, as lrzip had to be tested on a different system platform.

Table 2. Log file compression using other back-end algorithms

	lrzip	bzip2	LZMA	LP5+LZMA	PPMVC	LP5+PPMVC
2005access	0.272	0.246	0.215	0.119	0.191	0.104
Cernlib2005	0.171	0.181	0.182	0.136	0.139	0.133
FP	0.397	0.281	0.317	0.246	0.222	0.163
Mail	0.686	0.645	0.543	0.499	0.447	0.354
MYSQL	0.425	0.463	0.326	0.207	0.325	0.199
RMccess	0.734	0.736	0.626	0.577	0.588	0.534
Syslog	0.315	0.245	0.279	0.186	0.191	0.158
Average	0.428	0.400	0.355	0.281	0.300	0.235
Improvement	–	–	–	20.82%	–	21.77%
Comp. time (s)	–	246.8	660.4	194.1	79.3	95.6
Decomp. time (s)	–	27.2	6.4	15.1	76.4	79.5

The proposed transform improves the effectiveness of baseline algorithms in both cases (i.e., LZMA and PPMVC). The result of LP5+PPMVC is on average over 41% better than bzip2's. The only drawback of LP5+PPMVC is the relatively long decompression time. In cases where log archive has to be decompressed often (e.g., for data mining purposes), LP5+LZMA should be used instead, as it is over five times faster in decompression, though more than two times slower in compression. Notice that LP5+LZMA is more than three times faster in compression than LZMA alone. In case when both compression and decompression must be fast, LP5+gzip remains the best solution.

To ease the comparison, Figure 1 shows the compression ratio (as original file length to compressed file length) attained by the tested schemes.

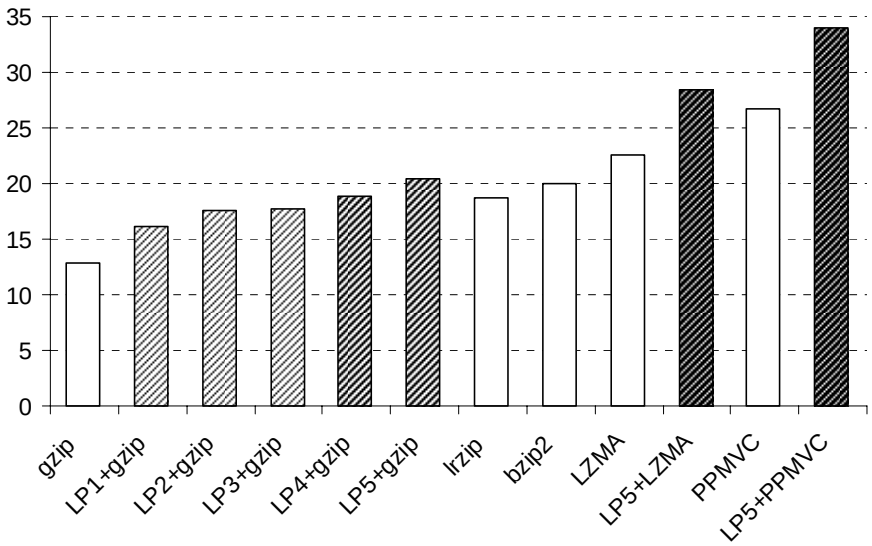

Fig. 1. Comparison of compression ratio

Conclusions

Contemporary information systems are replete with log files, often taking a considerable amount of storage space. It is reasonable to have them compressed, yet the general-purpose algorithms do not take full advantage of log files redundancy.

The existing specialized log compression schemes are either focused on very narrow applications, or require a lot of human assistance making them impractical for the general use.

In this paper we have described a fully reversible log file transform capable of significantly reducing the amount of space required to store the compressed log. The transform has been presented in five variants aimed at a wide range of possible applications, starting from a fast variant for on-line compression of current logs (allowing incremental extension of the compressed file) to a highly effective variant for off-line compression of archival logs.

The transform was not tuned for any particular type of logs, its set of features was designed for different types of logs, and the obtained test results show it manages to improve compression of different types of log files. It is lossless, fully automatic (it requires no human assistance before or during the compression process), and it does not impose any constraints on the log file size.

The transform definitely does not exploit all the redundancy of log files, so there is an open space for future work aimed at further improvement of the transform effectiveness.

References

1. Balakrishnan, R., Sahoo R. K.: *Lossless Compression for Large Scale Cluster Logs*. IBM Research Report RC23902 (W0603-038), March 3, (2006).
2. Gailly, J.-L.: *gzip* 1.2.4 compression utility. http://www.gzip.org. (1993).
3. Kolivas, C.: *lrzip* 0.18 compression utility. http://ck.kolivas.org/apps/lrzip. (2006).
4. 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).
5. *Maximum Compression (lossless data compression software benchmarks)*. http://www.maximumcompression.com. (2007).
6. Pavlov, I.: *7-zip* 4.42 compression utility. http://www.7-zip.org. (2006).
7. Rácz, B., Lukács, A.: *High density compression of log files*. In: Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, (2004), p. 557.
8. Rácz, B., private correspondence, (2007).
9. Seward, J.: *bzip2* 1.0.2 compression utility. http://sources.redhat.com/bzip2. (2002).
10. Skibiński, P., Grabowski, Sz., and Deorowicz, S.: *Revisiting dictionary-based compression*. Software – Practice and Experience, 35(15), (2005), pp. 1455–1476.
11. Skibiński, P., Grabowski, Sz.: *Variable-length contexts for PPM*. In: Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, (2004), pp. 409–418.

Appendix

Example snippets of the contents of the files used in the tests:

2005access

```
65.65.89.58 - - [24/Jan/2005:07:58:47 -0700] "GET
        /dods/fsl/01hr/2005024/20050241400_01hr.nc HTTP/1.0" 200 376472
209.153.165.34 - - [24/Jan/2005:08:04:44 -0700] "GET
        /dods/fsl/01hr/2005024/20050241500_01hr.nc HTTP/1.0" 404 311
128.117.15.119 - - [24/Jan/2005:08:08:42 -0700] "GET /dods/uniCat-model.xml HTTP/1.0"
        404 291
128.117.15.119 - - [24/Jan/2005:08:08:50 -0700] "GET /dods/model/ HTTP/1.0" 200 31307
204.62.251.81 - - [24/Jan/2005:08:08:51 -0700] "GET /cgi-bin/dods/nph-nc/dods/model/
        HTTP/1.1" 200 -
```

Cernlib2005

```
g77 -c -O -funroll-loops -fomit-frame-pointer -fno-second-underscore -fno-automatic
                                -fno-f90 -fugly-complex -fno-globals
     -fugly-init -Wno-globals    -I/Volumes/SandBox/fink/sw/src/fink.build/cernlib2005-
    2005-11/2005/src/mclibs/pdf -I/Volumes/SandBox/fink/sw/src/fink.build/cernlib2005-
    2005-11/2005/src/mclibs/pdf -I/Volumes/SandBox/fink/sw/src/fink.build/cernlib2005-
    2005-11/2005/src/mclibs/pdf/spdf -
    I/Volumes/SandBox/fink/sw/src/fink.build/cernlib2005-2005-11/2005/src/include  -
    DCERNLIB_LINUX -DCERNLIB_UNIX -DCERNLIB_LNX
```

```
                    -DCERNLIB_PPC -DCERNLIB_QMGLIBC
          -DCERNLIB_MACOSX   -o archive/sfkbmr2.o sfkbmr2.F
rm -f archive/sfkbmr5.o
```

FP

```
172.159.188.78 - - [03/Feb/2003:03:07:44 +0100] "GET /info/f4u.htm HTTP/1.1" 200 41011
        "http://www.fighter-planes.com/data4050.htm" "Mozilla/4.0 (compatible; MSIE 6.0;
        Windows 98)"
202.156.2.170 - - [03/Feb/2003:03:07:45 +0100] "GET /thumbn/jsf35_r.jpg HTTP/1.1" 304 -
        "http://www.martiworkshop.com/lf2forum/viewtopic.php?t=5832&highlight=firzen"
        "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.0.1) Gecko/20020823
        Netscape/7.0"
152.163.188.167 - - [03/Feb/2003:03:08:00 +0100] "GET /full/buccan.jpg HTTP/1.0" 200
        32618 "http://www.fighter-planes.com/data6070.htm" "Mozilla/4.0 (compatible; MSIE
        6.0; AOL 7.0; Windows NT 5.1; .NET CLR 1.0.3705)"
```

Mail

```
2007-01-14 06:26:01 cwd=/root 5 args: /usr/sbin/sendmail -i -FCronDaemon -oem root
2007-01-14 06:26:03 1H5xsX-0005PU-QT SA: Debug: SAEximRunCond expand returned: '0'
2007-01-14 06:26:03 1H5xsX-0005PU-QT SA: Action: Not running SA because
        SAEximRunCond expanded to false (Message-Id: 1H5xsX-0005PU-QT). From
        <root@research.pl> (local) for root@research.pl
2007-01-14 06:26:05 1H5xsX-0005PU-QT <= root@research.pl U=root P=local S=632
        T="Cron <root@server> php /home/www/research.pl/query/query.php" from
        <root@research.pl> for root
```

MYSQL

```
# at 818
#070105 11:53:30 server id 1  end_log_pos 975  Query      thread_id=4533      exec_time=0
           error_code=0
SET TIMESTAMP=1167994410;
INSERT INTO LC1003421.lc_eventlog ( client_id, op_id, event_id, value) VALUES( '', '',
        '207', '0');
# at 975
#070105 11:53:31 server id 1  end_log_pos 1131              Query      thread_id=4533
           exec_time=0          error_code=0
```

RMAccess

```
207.46.98.113 - - [21/Mar/2006:10:28:35 -0500]  "GET robots.txt HTTP/1.0" 404 213
        [msnbot/1.0 (+http://search.msn.com/msnbot.htm)] [] [UNKNOWN] 0 0 0 0 0 9001
207.46.98.113 - - [21/Mar/2006:10:28:35 -0500]  "GET ramgen/ferc/120905/ferc120905.smi
        HTTP/1.0" 200 359 [msnbot/1.0 (+http://search.msn.com/msnbot.htm)] []
        [UNKNOWN] 162 0 0 0 0 9002
```

65.216.119.102 - - [21/Mar/2006:10:49:48 -0500] "GET wm/revnine.wmv HTTP/1.0" 404 213 [contype] [] [UNKNOWN] 0 0 0 0 0 9003
198.40.41.250 - - [21/Mar/2006:10:55:05 -0500] "GET ramgen/ferc/031606/ferc031606.smi HTTP/1.0" 200 359 [Mozilla/4.0 (compatible;MSIE 6.0;Windows NT 5.1)] [] [UNKNOWN] 162 0 0 0 0 9005

Syslog

Jan 19 06:30:01 server /USR/SBIN/CRON[11013]: (root) CMD (lynx --source http://www.research.pl/index1.html > /home/www/research.pl/index.html)
Jan 19 06:30:02 server kernel: IN-world:IN=eth0 OUT= MAC=ff:ff:ff:ff:ff:ff:00:e0:81:27:d1:87:08:00 SRC=64.34.174.198 DST=64.34.174.255 LEN=78 TOS=0x00 PREC=0x00 TTL=128 ID=12552 PROTO=UDP SPT=137 DPT=137 LEN=58
Jan 19 06:30:02 server apache: PHP Warning: mysql_num_rows(): supplied argument is not a valid MySQL result resource in /home/www/research.pl/ktokto/index.phtml on line 11
Jan 19 06:30:07 server kernel: IN-world:IN=eth0 OUT= MAC=ff:ff:ff:ff:ff:ff:00:e0:81:27:d1:87:08:00 SRC=64.34.174.198 DST=64.34.174.255 LEN=78 TOS=0x00 PREC=0x00 TTL=128 ID=12566 PROTO=UDP SPT=137 DPT=137 LEN=58