computing@computingonline.net
www.computingonline.net

ISSN 1727-6209
International Journal of Computing

EFFICIENT PREPROCESSING FOR WEB LOG COMPRESSION

Sebastian Deorowicz [1], Szymon Grabowski [2]

[1] Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, POLAND,
sebastian.deorowicz@polsl.pl
[2] Politechnika Łódzka, Katedra Informatyki Stosowanej, al. Politechniki 11, 90-924 Łódź, POLAND,
sgrabow@kis.p.lodz.pl

Abstract: *Web log files, storing user activity on a server, may grow at the pace of hundreds of megabytes a day, or even more, on popular sites. They are usually archived, as it enables further analysis, e.g., for detecting attacks or other server abuse patterns. In this work we present a specialized lossless Apache web log preprocessor and test it with combination of several popular general-purpose compressors. Our method works on individual fields of log data (each storing such information like the client's IP, date/time, requested file or query, download size in bytes, etc.), and utilizes such compression techniques like finding and extracting common prefixes and suffixes, dictionary-based phrase sequence substitution, move-to-front coding, and more. The test results show the proposed transform improves the average compression ratios 2.70 times in case of gzip and 1.86 times in case of bzip2.*

Keywords: *web logs, text compression, table compression.*

1. INTRODUCTION

Plain text, as a medium for data conveyance and storage, is living its second youth. 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 (e.g., in order to track bugs in serialized objects or detect suspicious user behavior in a web traffic analyzer or an OS activity log), edit, and extract snippets from. 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 not only increases the costs of data transmission and storage, but can also slow down query handling. Another issue concerning redundant data are increased memory requirements, which may pose trouble in the notoriously multitasking and multi-user systems.

A natural approach to overcome the verbosity of textual data is, of course, to apply data compression. In fact, the current (April 2008) number of published papers dedicated to specialized XML compression exceeds 50 (see a bibliography at http://www.ucalgary.ca/~grleight/research/xml-comp.html), not counting works dedicated to specialized compression of some other structured text formats. 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 (or even decreased) computational requirements for the process of data encoding and decoding [1].

So far, most research on structured text compression focused on XML. Log data – e.g., database operation logs, file system access logs, installation logs – have rarely been subject of specialized text compression. A possible explanation of this little interest in the subject could be that log data form a rather vague category of files documenting human and machine activity: they may have same structure format in each line, but not necessarily so; they may have a fixed number of fields on a line, but not necessarily so; their fields may be whitespace separated, but other separators are possible too, and so on.

Among the most important log file types in everyday life 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, increases backup costs, and makes log data analysis and searches slow and cumbersome. Here is where, we believe, compression should enter the stage.

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 possible to compress log data about 15–80 times (about 40–45 times on average, even if the back-end compressor is mediocre gzip), preserving 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.

We focus on efficient preprocessing in order to improve the compression in the next stage, that is, using some general-purpose backup compressor. Not all redundancy can be squeezed out using this approach, but its benefits are simplicity and easiness of experimentation, because a significant part of the compression machinery is readily available.

The current paper extends our previous algorithm [2] with techniques manipulating on field values, and more advanced exploitation of field repetitions in proximity, which helped us decrease the average archive size on our test set by an extra 3–17 percent, depending on the back-end compressor used. A preliminary version of the current work was presented in [3].

2. SOURCES OF REDUNDANCY 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. Below we list our observations:

- there is one-to-one correspondence between events (page requests) and single lines in the log file;
- web log files are similar to tables in a relational database: lines (rows) are composed of fields (attributes) in a fixed order, typically separated with blank spaces;
- several pattern types are very frequent: IP addresses, timestamps (in a fixed format), URL’s;
- 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 frequent visitors;
- successive lines tend to store requests from the same user, consequently with repeating IP addresses and client OS / browser data;
- field values repeat in proximity (although not necessarily in successive lines);
- timestamps of the successive entries are often very similar, which suggests differential encoding as an effective means to squeeze out the redundancy;
- 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 apart, is very likely to be followed by the “old” web browser ID string);
- the plain ASCII character set is almost exclusively used in web logs, which means that the byte values over 127 (plus most byte values below 32) are unused and could be spent for cheap substitution of frequent sequences.

Table 1 presents how the mentioned properties of most web log files can be utilized by (general purpose or specialized) compression algorithms. As can be seen, some effects are relevant only (or mostly) for LZ compressors, but others, like phrase substitution and differential encoding of timestamps, serve other compressors (e.g., from the PPM or BWT family) in a similar degree.

We note that really huge log collections (on the order of millions requests) may instigate to more refined and more quantitative observations, like, e.g., stating that the distribution of page access count is Zipf-like [4].

3. RELATED WORK

Most existing utilities for archiving and analyzing log data use zip / gzip (Deflate) compression, while some make use of a newer and stronger compressor bzip2 (e.g., Web Log Mixer). 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.

Rácz and Lukács developed the *differentiated semantic log compression* (DSLC) algorithm [5], but some details of this scheme were not given. It 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. As the reader will see later, our techniques are inspired by DSLC. The results cited in the original work are quite impressive, but the authors of [6] claim that 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”.

Table 1. Properties of web log data and the resulting compression opportunities

data property	**how compression is affected**
1–1 line–request corresp.	Processing simplicity. Spatial closeness of identical sequences (good for LZ compression).
Frequent patterns	Efficient phrase substitution (typically as a preliminary compression step) possible. Phrases, added to a dictionary, can be whole field values or only snippets.
Long repeating text sequences	Can be encoded efficiently as LZ77-matches.
Request from the same user on successive lines	Can be encoded efficiently as LZ77-matches, thanks to a cheap LZ77-match offset.
Field values repeat in proximity	Can be encoded efficiently as LZ77-matches, but also an algorithm for the list update problem (LUP) [7] can be applied
Similar timestamps on successive lines	Differential encoding (as a preliminary compression step) can be very efficient.
DB table like field ordering	Fields can be processed separately. Domain knowledge for the field (which may be inferred in the analysis stage) is precious for compression. For example, knowing that a field contains IP's means that the sequences of digits always form integers in 0...255 range, separated with single dots.
Strong correlations across fields	Fields can be joined, to improve LZ compression due to prolonged matches. Also, longer phrases can be added to the dictionary, for a preliminary phrase substitution step.
Only ASCII symbols 32...127, plus EOLs, used	The remaining symbols could be spent for cheap substitution of frequent phrases.

A highly specialized log compression scheme was developed by Kulpa et al. [8]. They encode the web user activity logs in a client-side monitoring system, written in JavaScript. The obtained compression is mediocre, but this was to be expected because the system has to be fast in the given environment, and works on small log chunks; the involved compression techniques comprise string substitution and differential date/time encoding techniques.

In 2007, Skibiński and Swacha [6] proposed a couple of simple preprocessing variants intended to facilitate further compression of diverse log files. Since their goal was broader than ours (the tested logs were from different applications), they used more general means of transforming data. In the simplest variant, each line is encoded with reference to the previous line, storing the length of the longest match on a single byte (using symbols 128...255), 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). Later variants add a dictionary substitution for words found in a prepass (an idea used earlier, e.g., in [9], for plain text compression), and compact encoding of numbers, dates, times and IP addresses. In their experiments, the transform help shorten Deflate (the default zip algorithm) archives by 37% on average. Significant improvements (on the order of 20%) have also been noticed when stronger back-end compression algorithms (LZMA, PPMVC) were used.

Capturing dependency among columns in two-dimensional tables was the subject of the work by Vo and Vo [10]. They considered reordering columns to maximize compression; although solving this problem optimally is NP-hard, they gave efficient solutions in restricted settings, and also tested their ideas on a number of real tables. There was no web log in their test collection but it is likely that their scheme fits this application domain too.

4. APACHE WEB LOG FORMAT

The default field order in Apache logs is fixed (http://www.jafsoft.com/searchengines/log_sample.html). We list them below. The field numbers are only for reference in the latter sections.

#0 – visitor's IP address,

#1, #2 – username etc. Set to – –, unless accessing password-protected content,

#3 – timestamp of the visit (date, time, timezone),

#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 filesize; 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 came from 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 file.

5. OUR ALGORITHM

The current work is an extension of our previous attempts [2,3] to design a transform suitable for efficient compression of web log files. The enhancements and novelties compared to the previous work will be explicitly stated.

At the start, we split the data into individual fields, and each is compressed separately. In [2], we tried two variants; in the more advanced one, two pairs of fields (#0, #8 and #4, #6) were merged at some step of the transform (not at its start), before further processing. This helps on average, but for some files a little compression loss was observed. Also, hard-coding which fields should be merged cannot be called an elegant solution, and we have not been able so far to find a satisfactory technique to analyze the compression-directed correlation between fields. From those reasons, we abandoned field merging in the current paper.

The rule for splitting a line into fields is quite simple. The separators are spaces (one or more), except for those that occur with a pair of brackets [] or quote marks "". The closing quote mark " is the nearest one after the opening ", with our exception (needed to handle properly one of the test files, rmaccess). If the sequence is something like "abc de" fgh", i.e., the third (supposedly, an opening) quote mark is not prepended with a space, then its predecessor, i.e., the second quote mark in our example, is considered an internal character of the field, and the third one is the closer.

Processing the data field by field, or, in other words, column by column, is equivalent to transposing a relational database table, which is a well-known idea attempting to increase compression [11]. The approach has a number of benefits: it is easier to perform dictionary substitution on individual fields; limiting the scope of the compression model to what is relevant results in lower CPU and memory requirements, recency effects (e.g., runs of occurrences of the same field value) can be conveniently exploited, and so on. Another possibility is to compress the fields with a separate model for each, and switch between those models with every field value. This approach has the benefit of being on-line but requires housing several models at the same time, i.e., needs more memory to work. Moreover, this approach seems to exclude compressors from the LZ77 family, unless for the price of significant complications and decompression slow-downs.

Below we present the processing of the successive fields, as listed in Section IV. Sometimes there are extra fields (the file rmaccess in our test collection), they obtain "default" treatment, as it will be described later.

Field #0. We noticed that recently occurring IP addresses are more likely to occur again than novel IP's. To exploit this, the well-known *move-to-front* (MTF) transform [12] could be applied for this field, which encodes a given value v as the number of *unique* values between the previous and the current occurrence of v. Indeed, this is the solution we used in [2,3]. Namely, in the cited works, for each field value v we sent into the first stream 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 had 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 had 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 was larger than 256, we treated v as a never-appeared-before value and encoded both 2 and v. In [3], MTF codes and the stream of ternary flags were order-2 PPMd compressed, if the back-end compressor is gzip, bzip2 or LZMA, or just the chosen PPM back-end is used otherwise. Analogous MTF processing was used in [2], with the only difference of using order-1 arithmetic encoding (via the archiver arhangel, v1.40a2). Wherever in the current scheme we used PPM for encoding flag streams, order-1 arithmetic coding had been used earlier. What was sent to the third stream, the raw IP addresses, were naturally encoded on four bytes each, without separators, and given to the chosen back-end compressor.

Currently, we refine this idea. We use three streams again, let us denote them with s_1, s_2 and s_3. If field value v occurred just in the previous row, we send 0 to s_1. If field value v is new or its MTF rank is high enough (to be specified later), value 1 is sent to s_1, and v itself is sent to s_3. The novelty is in handling all intermediate values of MTF. We distinguish between low and high MTF ranks. Low ranks are those from 1 up to *thresh*, high ranks exceed *thresh*. Field values v with low MTF ranks are encoded in s_1 as 2+MTF(v). Field values v with high MTF ranks emit two codes: 2 to stream s_1, and MTF(v) – *thresh* to stream s_2. The value of *thresh* was experimentally set to 128. In this way, the MTF list stores 256 + *thresh* = 384 symbols; even higher ranks are handled as field values never seen before. This modification improves overall compression by about 1–2%, depending on the chosen back-end compressor. We have tried some other variants too, like MTF-1 [14], but they were unsuccessful.

Fields #1, #2. They usually contain – (dash) values. We remove duplicates in those fields, which can be seen as an extremely simplified MTF variant: if the current field value is just like on the previous line, we sent 0 to a flag stream, otherwise we sent there 1 and the current value to the other stream. The flag streams are PPM-compressed.

Field #3. Timestamps need special handling. Things would be easier if all requests were from the same time zone (which is also specified in this field; here is an example: "+0100"), but of course this is not always the case. Our algorithm produces two files. In one of them the time zones are ignored, and it contains encoded time differences (as if all the requests were from the same time zone) between successive lines. Those differences are expressed in seconds (the grain of recorded time), and are encoded either on a single byte, if the difference is in the range 0...253, or on five bytes otherwise: 255 (254) stands for a flag for a positive (negative) difference, and the following four bytes encode the absolute value of the difference. The other file, representing time zones, contains one byte per line, and it keeps on individual bits the sign (plus or minus), the hour (0...23) and two bits to distinguish if the local time deviation from Greenwich Mean Time can be expressed in full hours, and if not, how many quarter-hours (1, 2 or 3) have to be added); the latter cases are rare but are also used in some regions of the world). Both output files are PPM-compressed.

Field #4. Handling this field is most complex, and differs significantly to the solution in our previous algorithm. First, if the number of distinct field prefixes is not more than 16, and also the number of common suffixes 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-2 PPMd compressed, while the vocabulary files are given to the main back-end compressor. On the "stub" file, MTF processing is performed, like it was described for the field #1. Those steps were used in the previous version of our algorithm (with the only difference of using PPMd -o2 this time). The next step is new: we noticed that the remaining main stream sometimes stores text sequences which share common prefixes or suffixes, hence we remove them to another pair of extra streams. What we write to those streams are pairs of bytes: one tells which of the 16 previous lines starts with the longest matching prefix (suffix), and the other tells the length of this match. More precisely, our rule some prefers not longest matches if they are much closer. Finally, on what remains after removing the prefix and suffix, we perform phrase substitution, which is also a novelty in the algorithm. In this step, (up to) 156 "most valuable" phrases are found (slightly less, up to 145, was used in [3]), where a phrase is any field subsequence between any pair of the delimiters: /, &, ?, ., +, = and a blank space (this choice was partly dictated by the set of frequent PHP separators). Allowed delimiters are also the beginning and the end of the (remaining) field value. Those phrases (of length at least 2 characters) are then encoded on 128...255 and 1..9, 11, 12, 14...30 ASCII symbols. The set of selected phrases contains those phrases that have the greatest weight, understood as the product of phrase occurrence frequency and phrase length.

We note that occasionally (very rarely) some characters from 128...255 range do appear in some log files. We use the ASCII character 31 to flag any such atypical occurrence, and the non-ASCII character (or character 31) goes to another stream.

Fields #5, #6, #7, #8. Those fields are treated alike, using the MTF-based stream-splitting routine as described for the field #0.

We cared for making the transform fully lossless. To this end, we create a small configuration file, storing the End-Of-Line convention in a given log to compress, number of spaces between fields, and so on.

6. EXPERIMENTAL RESULTS

We have run several experiments to evaluate the performance of our algorithm, in both current and previous version [2]. The variant shown in this work was implemented in C++ and compiled with Microsoft Visual Studio 2008. All tests were run on an AMD Athlon64 3000+ machine with 2 GB of RAM, running Windows XP (SP2) operating system. For order-2 statistical encoding, applied in some stages of our transform, we used PPMd, var. J (links to all mentioned general-purpose compressors can be found at http://www.maximumcompression.

com/programs.php).

The test collection comprises six files, varying in size from 3 MB to over 110 MB. Four of them, access, latexeditor, netaccess and fp, were used in [2], while two new files, 2005access and rmaccess, were taken from [6], provided to us by Przemysław Skibiński.

Rmaccess was quite a problematic file. It had 7 extra fields at the lines' end, and we processed them all, starting from field #9, like its preceding fields #5...#8 (described in the previous section). Also, extracting the fields from the file required a special rule (see Section V).

Our previous implementations [2, 3] were written in Python, which by no means can compete with compiled languages (like C++) in execution speed. Now, we can provide compression times of our transform, which demonstrate that our transform should be practical.

For comparison, we were able to get only one specialized log compressor, logpack [6]. It works on arbitrary logs (not only web logs) and 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 algorithm.

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, v1.3.12, implementing the Deflate method from the LZ77 family, in its default mode -6,
- 7z, v4.57, using its default algorithm, LZMA, a modern representative of the LZ77 family, with default settings,
- bzip2, v1.0.2, a compressor based on the Burrows–Wheeler transform, using 900 kB blocks,
- PPMd, var. J, a efficient implementation of the PPM algorithm, tested in order-6 and order-16, using up to 256 MB of memory,
- PPMonstr, var. J, a state-of-the-art PPM algorithm, where excellent compression comes at a price of high computational requirements; tested in order-6 and order-16, using up to 512 MB of memory.

Table 2 contains the results, with compressed files in columns and compression methods in rows. The first group of rows, under the raw file sizes, stores the compression ratios from the bare general-purpose compressors used in the comparison. In the next group, Logpack (variant 5) results in combination with those compressors are presented. In the third group, Logpack preprocessing is replaced with our previous [2] transform for web log data, still the results are somewhat different than in [2]. This is due to two reasons: one is that, unfortunately, the compression ratios given for the latexeditor file in [2], for our both variants presented there, were wrong (due to a bug which did not affect any other file). Another reason for the discrepancy is that now we replaced order-1 arithmetic coding with PPM coding, as it is described in Section V. In this way, those two groups of results – old and new – are directly comparable. The last group of rows stores the results of our current transform combined with external compressors. The rightmost column contains the average compression ratios across the collection, expressed in bits per character (bpc).

Table 3 gives compression times for the back-end compressors run alone, and the execution times of our transform in the last row. Although we believe there is still room for speed optimizations in our C++ implementation, we see that using the transform with bzip2, 7z (LZMA) or PPMonstr speeds up overall compression significantly. On the other hand, the faster compressors, gzip and PPMd, suffer a big speed drop.

It is worth to note that efficient preprocessing for web logs flattens the compression results from different back-end compressors, hence using faster algorithms in the latter stage is more justified.

7. CONCLUSIONS AND FUTURE WORK

We presented a relatively simple off-line preprocessing scheme 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 algorithm processes each field separately. The biggest improvement, as expected, was achieved in combination with gzip, the weakest (but also fastest and most widely used) among the tested compressors. On average, the archives shrunk 2.7 times with gzip, but those factors varied significantly from file to file.

As expected, with stronger back-end compressors the gains diminished but are still very significant: the respective speedup factors are 1.86 with bzip2, 2.33 with 7z (LZMA) and 1.74 with order-6 PPMd. Our advantage over logpack, a specialized log compressor, is also impressive, the average compression gain factor is 1.74 with gzip back-end. Still, on some files it disappears when the strongest of the tested compressors, PPMonstr, comes into play (especially in a high-order regime). For most files, it seems hard to improve the results of order-16 PPMonstr significantly, by more than 10-15%.

Table 2. Compression results in bits per character (bpc). Second top row holds the original file sizes in bytes.

log file →	2005access	access	fp	latexeditor	netaccess	rmaccess	average
raw file (in bytes)	16 706 861	17 517 060	20 617 071	30 381 282	3 105 150	116 914 549	–
gzip	0.445	0.419	0.562	0.388	0.306	0.960	0.513
bzip2	0.246	0.256	0.281	0.212	0.168	0.736	0.316
LZMA	0.336	0.357	0.360	0.274	0.294	0.732	0.392
PPMd -o6 –m256	0.224	0.201	0.254	0.227	0.162	0.651	0.286
PPMd -o16 –m256	0.165	0.173	0.226	0.175	0.131	0.576	0.241
PPMonstr -o6 -m512	0.131	0.126	0.191	0.134	0.111	0.501	0.199
PPMonstr -o16 -m512	0.104	0.111	0.174	0.109	0.097	0.458	0.176
LP5 + gzip	0.173	0.270	0.333	0.234	0.152	0.825	0.331
LP5 + bzip2	0.116	0.185	0.244	0.157	0.124	0.601	0.238
LP5 + LZMA	0.133	0.222	0.252	0.169	0.127	0.598	0.250
LP5 + PPMd -o6	0.132	0.140	0.210	0.139	0.102	0.530	0.212
LP5 + PPMd -o16	0.110	0.131	0.204	0.128	0.109	0.494	0.196
LP5 + PPMonstr –o6	0.105	0.100	0.192	0.124	0.102	0.483	0.184
LP5 + PPMonstr -o16	0.091	0.093	0.185	0.116	0.097	0.447	0.172
[2], v1 + gzip	0.193	0.229	0.166	0.103	0.107	0.572	0.228
[2], v1 + bzip2	0.125	0.168	0.149	0.096	0.098	0.476	0.185
[2], v1 + LZMA	0.117	0.203	0.156	0.100	0.104	0.452	0.189
[2], v1 + PPMd -o6	0.131	0.135	0.146	0.095	0.097	0.463	0.178
[2], v1 + PPMd -o16	0.107	0.126	0.145	0.094	0.097	0.433	0.167
[2], v1 + PPMonstr -o6	0.093	0.100	0.145	0.094	0.093	0.403	0.155
[2], v1 + PPMonstr –o16	0.085	0.098	0.143	0.093	0.093	0.384	0.149
current + gzip	0.096	0.134	0.157	0.099	0.107	0.549	0.190
current + bzip2	0.091	0.122	0.147	0.095	0.099	0.469	0.170
current + LZMA	0.093	0.122	0.150	0.097	0.104	0.442	0.168
current + PPMd -o6	0.082	0.103	0.146	0.095	0.099	0.461	0.164
current + PPMd -o16	0.081	0.101	0.146	0.095	0.099	0.438	0.160
current + PPMonstr -o6	0.075	0.094	0.139	0.090	0.090	0.398	0.148
curr. + PPMonstr -o16	0.073	0.092	0.138	0.090	0.090	0.384	0.144

Table 3. Compression times in seconds. Rightmost column, lower rows, in parentheses: the ratio (transform_time + backend_compressor_time) / bare_backend_compressor_time.

log file →	2005access	access	fp	latexeditor	netaccess	rmaccess	total time
Gzip	0.67	0.70	0.91	1.13	0.12	6.94	10.47
bzip2	11.10	12.09	14.41	18.99	2.24	59.04	117.87
LZMA	5.50	5.09	6.93	8.42	0.88	78.32	105.14
PPMd -o6 –m256	0.67	0.75	0.90	1.25	0.12	12.76	16.45
PPMd -o16 –m256	0.90	1.03	1.26	1.52	0.14	19.91	24.76
PPMonstr -o6 -m512	15.96	16.54	20.22	28.16	3.00	153.60	237.48
PPMonstr -o16 -m512	18.56	19.30	24.18	32.84	3.47	207.16	305.51
our transform	1.77	3.09	2.29	2.16	0.41	24.79	34.51
transf. + gzip	5.07	6.48	6.56	6.78	4.36	36.30	65.55 (626.1%)
transf. + bzip2	5.09	6.41	6.59	6.79	4.28	42.16	71.32 (60.5%)
transf. + LZMA	5.19	6.71	6.68	6.86	4.30	54.17	83.91 (79.8%)
transf. + PPMd -o6	5.10	6.47	6.56	6.84	4.26	40.38	69.61 (423.2%)
transf. + PPMd -o16	5.17	6.54	6.65	6.86	4.32	42.58	72.12 (291.3%)
transf. + PPMonstr –o6	8.88	10.48	13.14	13.01	5.77	111.51	162.79 (68.5%)
transf. + PPMonstr -o16	9.12	10.84	13.54	13.64	5.80	117.47	170.41 (55.8%)

It is well known that data compression in databases can speed-up overall processing [11,13], because the increase in CPU work due to compression/decompression operations is more than offset by reduced I/O. We are ignoring this aspect, since we assume that web log files are compressed mainly for speeding-up backuping and saving storage. Still, queriable compression is also an interesting research goal, preferably performed on-line to allow fast incremental archive updates. Logpack is an example of an on-line compressor, but this also seriously hampers the compression ratios it attains. We believe that a compromise between those solutions is possible, with column-oriented processing in relatively small blocks.

Even in the off-line setting, several techniques require polishing. It was noticed in our experiments that some ideas help, e.g., to gzip but harm with PPMd, etc. In the future, we are going to tailor the transform for two or three back-end compressor targets (e.g.: LZ compressors, practical PPM compressors, top performance PPM compressors).

The MTF transform is a simple and well-known representative of the family of transforms coping with the list update problem [7]. Still, in the context of BWT compression, there are known more successful solutions (see, e.g., [14]) and it will be interesting to try them out in our application.

8. REFERENCES

[1] P. Skibiński, Sz. Grabowski Sz., J. Swacha, *Effective asymmetric XML compression*, Software–Practice and Experience, accepted.

[2] Sz. Grabowski, S. Deorowicz. Web log compression, *AGH Automatyka* 11 (3) (2007). pp. 417–424.

[3] S. Deorowicz, Sz. Grabowski. Sub-atomic field processing for improved web log compression. *Proceedings of the 9th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET'2008)*, Lviv-Slavsko, Ukraine, 19–23 Feb. 2008, pp. 551–556.

[4] A. A. Benczúr, K. Csalogány, K. Hum, A. Lukács, B. Rácz, Cs. I. Sidló, M. Uher, L. Végh. Architecture for mining massive web logs with experiments. *Proceedings of the HUBUSKA Open Workshop on Generic Issues of Knowledge Technologies*, 2005. http://www.sztaki.hu/~alukacs/Papers/origomining.pdf

[5] B. Rácz, A. Lukács. High density compression of log files. *Proceeding of the IEEE Data Compression Conference (DCC)*, Snowbird, UT, USA, 2004, p. 557.

[6] P. Skibiński, J. Swacha. Fast and efficient log file compression. *CEUR Workshop Proceedings of the 11th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, Varna, Bulgaria, 23 Sept. – 3 Oct. 2007, pp. 330–342.

[7] S. Albers. Online algorithms. Book chapter in *Interactive Computation: The New Paradigm* edited by D.Q. Goldin, S.A. Smolka and P. Wegner, Springer, 2006, pp. 143–164.

[8] A. Kulpa, J. Swacha, R. Budzowski. Script-based system for monitoring client-side activity. Book chapter in *Technologies for Business Information Systems* edited by W. Abramowicz, H. Mayer, Springer, Dordrecht, Netherlands, 2007, pp. 393–402.

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

[10] B. D. Vo, K.-P. Vo. Compressing table data with column dependency, *Theoretical Computer Science* 387 (3) (2007). pp. 273–283.

[11] G. Graefe G., L. Shapiro. Data Compression and Database Performance, *Proceedings of ACM/IEEE-CS Symposium on Applied Computing*, Kansas City, MO, USA, 1991, pp. 22–27.

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

[13] B. R. Iyer, D. Wilhite. Data Compression Support in Databases. *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, Santiago de Chile, Chile, 12–15 Sept. 1994, pp. 695–704.

[14] S. Deorowicz. Universal lossless data compression algorithms. Ph.D. dissertation, Silesian University of Technology, 2003.

__Sebastian Deorowicz__ (b. 1974) received his MSc degree in Silesian University of Technology in 1998 and PhD degree in the same university in 2003, both in computer science. His research interest are in data compression, string matching, sequence alignment, and combinatorial optimization. He has published over 10 journal and conference papers. He is currently an assistant professor at Silesian University of Technology in Gliwice.

__Szymon Grabowski__ (b. 1973) received his MSc degree in Łódź University in 1996 and PhD degree in AGH University of Science and Technology (formerly known as University of Mining and Metallurgy) in Cracow in 2003, both in computer science. His former research, including PhD dissertation, involved nearest-neighbor classification methods in pattern recognition, also with applications in image processing. Currently, his main interests are focused in string matching and text indexing algorithms, and data compression. Some of his particular research topics include various approximate string matching problems, compressed text indexes, and XML compression. He has published about 60 papers in journals and conferences. He is currently an assistant professor at Computer Engineering Department of Technical University of Łódź.