

An Online and Efficient Algorithm for Compressing Web Server Access Logs

Peter Bodik, Wei Xu, Archana Ganapathi

EECS, UC Berkeley
{bodikp,xuw,archanag}@cs.berkeley.edu

Abstract. In this project we look at compression of standard access logs from HTTP servers. It is important for our research since the logs need to be transmitted over network from a production system to our repository or get analyzed in real time. An online algorithm that achieves good compression ratio can help reduce the network traffic and required disk space. We analyze the structure and statistical characteristics of real logs from an e-commerce website, and show that by using a modified version of Dynamic Huffman Tree algorithm [8], we can achieve better compression ratio than standard *gzip*. Also, this scheme is online, i.e., it is capable of compressing or decompressing data even when data is presented line by line.

1 Introduction

An increasing number of large complex systems are being used today to support a range of activities from processing web transaction to computationally intensive simulation. Those systems generate a huge amount of logs that can add up to a terabyte per day. Those logs contain valuable information for business, legal or technical use, and thus need to be archived.

On a distributed computer system, logs are usually sent over network to a centralized place where they are processed and archived on to disk or other media. Compressing the original logs before sending them over the network will significantly reduce the I/O bandwidth and disk space required. Though nowadays network bandwidth and disk space are getting larger and cheaper, there is evidence that CPUs are getting faster much more significantly []. Thus, compressing data before it is sent to I/O devices improves the overall system performance.

Sometimes, logs need not only be archived, but also analyzed in real-time. Therefore we also need an online algorithm that can compress/decompress the log in a unit that is as small as possible. Specifically, in the case of HTTP log from either Apache HTTP Server or Microsoft IIS, we want to compress and decompress the data line-by-line. However, *gzip* cannot achieve its best compression ratio when the data chunk is as small as one line of log [6].

Comparing to general English texts, system logs have better-defined schemas. That is, there can only be a fixed number of fields, and most of the fields have specific data type (e.g. numeric type, or date/time). Even better, most fields contain only a limited number of unique values. Based on those facts, we believe that system logs have smaller entropy than normal English text.

General compression utilities, such as *gzip*, do not handle these types of logs well because they go through the log line by line and fail to exploit the log schema, which is organized column by column. As we will see in the later sections, the block size used by *gzip* is not large enough to capture the reoccurrence of values in some fields [6]. Another problem of *gzip* is that it does not recognize the data type for each field. For example, encoding a date time as a epochs is much better than encoding them as ASCII strings.

In this project, we first analyzed the structure and statistical characteristics of an HTTP log from a real e-commerce website.

We implemented a modified version of Dynamic Huffman Tree [8], which operates on each column. We show that even without many optimizations and heuristics, this algorithm still achieves better compression ratio than *gzip*.

Our algorithm is based on the statistics of our sample data set, but is not too specific for this data set, since the only characteristics we exploited are the column structure and simple column statistics, such as the distribution of distinct values. Those statistics are kept by almost all databases for query optimization [12]. We did not use any knowledge about the internal structure of each column, which is very application specific; though using it will certainly result in even better compression ratio.

2 Related Work

We studied several kinds of data compression and coding algorithms. In this paper, we focus on lossless compression techniques.

General data compression algorithms

There are many general data compression algorithms. They can be classified in two categories: static or adaptive [9].

A static method decides on a mapping from words to codewords before the transmission starts. The classic Huffman Tree algorithm belongs to this category. It encodes all symbols in a static alphabet according to their probability of appearing in the text. These algorithms are simple, run fast, but calculating the statistical characteristics requires looking at the whole text. In addition, the encoding mapping must be sent. This scheme is not suitable for an online algorithm. Also, ways of defining a "symbol" in these algorithms are also static. The symbol we choose may not represent the best set of symbols. For example, using a letter in English as a symbol for encoding may not be as good as using three-letter triples [1].

Adaptive coding algorithms adapt to changes of the input characteristics by changing the encoding scheme over time. They are usually *one pass* and thus online algorithms. There are two different kinds of adaptive algorithms. First, Dynamic Huffman Tree based algorithm. These algorithm are *defined-word* schemes (i.e., similar to classic Huffman Tree, where all words have been chosen statically), but the code adapts current estimates of message probabilities of each word and thus remains optimal for the current estimates. The variations of Dynamic Huffman Tree algorithm includes FGK algorithm designed by Faller, Gallager and Knuth [8], and Vitter algorithm [14]. In our project, we modified Dynamic Huffman Tree algorithm to handle a bounded sized, but unknown alphabet. We discuss this algorithm and our modifications in Section ??.

Lempel-Ziv algorithm [15] is another adaptive coding scheme, but differs from Dynamic Huffman Tree since it is a *free-parse* method – the words of the alphabet are defined dynamically. Lempel-Ziv, if running with a dictionary with unbounded size, can achieve the optimal compression ratio. However, unbounded Lempel-Ziv is not practical due to the memory constraints. Fortunately the words used in a document have certain locality (which is usually the case in human languages). So this scheme works fine in many cases even with a bounded-size dictionary [6].

Specialized Compression Algorithms

General compression algorithms do not work well in all cases. There are some specialized algorithms for certain problems. For example, for lossless image compression, *run-length encoding*, which encodes sequences of repeating patterns as (pattern, length) pairs, shows good results [?].

The algorithm used in IBM's IMS (an early database system developed by IBM [5]) is a hybrid compression algorithm that exploits the local redundancy for each field of a record, and uses different compression schemes for each field, depending on whether it is a letter, a digit or a pair. This algorithm uses structural information of database schemas, but does not use the statistical characteristics for each field.

With the development of bioinformatics, there are many specialized algorithms for compressing DNA sequences. Some of them, such as Biocompress [7] and Cfact [13], use longest exact matching repeats, which are similar to Lempel-Ziv. However, they use better methods to find the longest exact match, which uses less memory and is capable to find non-local repeats.

Other DNA compression algorithms consider the fact that most of the sequence repeats are approximate repeats. GenCompress [3], CTW+LZ [11], and DNACompress [4, 10] uses specialized pattern matching algorithm to find approximate repeats, and those approximate regions are encoded with Hamming distance (the transcript of changing one string to another with only substitutions), or edition distances (using deletion, insertion and substitution).

Implementations of Compression Algorithms

There are standard utilities that are based entirely on a single algorithm. For example, UNIX *compress* is based on Lempel-Ziv and *compact* is based on the FGK algorithm [8].

However, other programs make use of several algorithms in combination. For example, *gzip* uses Lempel-Ziv to find the longest repeating sequence, and uses the Huffman Tree algorithm to encode the offset and length of the sequence. *Bzip2* compresses files using the Burrows-Wheeler block-sorting text compression algorithm [2] and Huffman coding.

Practical implementations, even general purpose ones, usually use a lot of heuristics to reduce CPU/memory usage and complexity of the implementation. For example, in *gzip*, data is broken up into "blocks", whose size is a configurable parameter, and each block uses a different compression mode [6].

3 Observations of System Data

HTTP access logs record information about requests from clients (typically web browsers). In the simplest case, the client sends a request for a certain page and the web server returns the HTML page along with all the images referenced to the client. However, in modern e-commerce websites, most of the pages are dynamically generated with scripts and backend databases, and the client submits a request together with some arguments to those scripts. A session is usually kept for a client for a certain period of time, which helps the website operator to track the preference of each client. These parameters are session information are also contained in the access log.

Usually, certain information in logs is sensitive. For example, to protect privacy of all clients, a company doesn't want others to know how many requests come from which client, and what page a specific client is requesting. Therefore, some fields in the logs are masked by replacing their values with the MD5 hash of their values (despite the fact collisions of MD5 hashes are found, we still assume it is a pretty good one-way hash function.)

Figure 1 shows two lines of a typical HTTP log. The sample logs we used were generated by production servers from a mid-sized e-commerce company over a period of about a week. The file is in comma-separated-values format, which contains 2 millions of lines in the same format as Figure 1. The original file size is 317,163KB.

Each line of the log file contains seven fields, which are sequence number, time stamp, name of the script used, name and value of the parameters to this script, user IDs, session IDs, and the local log file name. The value of some parameters, as well as user IDs and session IDs are masked using MD5.

```
71382,  
2001-07-24 00:00:09.000,  
/malltop.go,  
merchant_id=49743e931ceeeefdbea6e36858642a03,  
31927399077336290688401770303349286400,  
145ed5b359105df0ac53569be69cf3d7,  
2001-07-24-0005_app02.log  
  
129062,  
2001-07-24 00:00:14.000,  
/faqs.go,  
,  
188202603373917999168154629607605610937,  
557f8b2c316f9a7c8e6a99b74486a01a,  
2001-07-24-0005_app01.log
```

Fig. 1. Two lines of a sample log file

Obviously, the entropy of these logs is smaller than standard English text, not only because it has a fixed schema (there are seven fields in each line, each with a certain data type), but also because each field can only have a limited number of distinct values. MD5 encoding, in this case, is actually a inflation encoding since a site cannot possibly have 2^{128} distinct clients.

Of those seven fields, we focus on the fields that contain MD5 hashes. This is because those fields are most critical to the overall compression ratio. The other fields have very simple data types and can be handled by any general compression algorithm well enough.

Some statistics helps us get a better understanding of the log. First, we look at the number of distinct values for each field, shown in Table 1. We see that out of almost two millions of lines of logs, field 2 has only 4200 distinct values. This field is (usually) the name of script, and on a certain website, there are only a small number of distinct scripts. Fields 3 and 5 have less than 70000 distinct values, only 3.5% of the total number of lines. This makes sense semantically, too: these fields represent the parameter and user ID, which have relatively small number of values. Field 4, the session ID, has many distinct values (but still very small comparing to the total number of lines). These repeating values in each field make the data compression possible.

Field number	Meaning	Number of distinct values
0	Sequence Number	*
1	Timestamp	601,008
2	Script Name	4,200
3	Parameter	68,764
4	Session ID	150,172
5	User ID	41,841
6	Log File Name	149

Table 1. Number of distinct values in each field. A * in the distinct values indicates that this field is time stamp related and has almost the same number of distinct values as the number of lines of code.

We also want to find out the correlations between two fields. If two fields are highly correlated, it makes sense to treat them as only one field. We do this by simply calculating the number of distinct values if two fields are concatenated. It turns out that if we concatenate fields two and three (i.e., the name of the script and its arguments), there are only 84,286 distinct values, a little more than field 3 by itself. This correlation is not surprising, since a script can take a limited set of parameters. However, other fields shows no correlations, as the distinct values of any other two fields combined with be more than 200,000.

We also consider the distribution of distinct values for each field. If the each distinct value has an equal probability of appearance, a simple equal-length coding for each unique value is enough. As in many other cases, the distinct values have a Zipf distribution. Figure 2 shows this fact. This requires us to find a way of coding the most commonly used values with shorter code and the least commonly used with longer codes.

As the last part of our statistical analysis, we consider the locality of each unique value. If there is no locality at all, we may need a lot of memory to keep all values we have seen for a long time. Figure 3 shows the number of lines processed versus the number of unique values found. This number is important since we can use this information to predict how much unique values we will see in the future. Only in field 2, most of the values are seen at the beginning of the processing, in all other fields, unique values continuously appear and the lines are almost straight lines.

Figure 4 shows the locality of each unique value in fields 2–5. The locality is calculated by the number of lines of between the first appearance and the last appearance of a unique value. Interestingly, we found that values in field 4 and 5 have much better locality than values in field 2 and 3. This makes sense intuitively. Because a script on the website can be used over and over again and thus can appear anywhere in the log. On the other hand, a session ID is only used for a short period of time and a user is active only during a certain period of time. This analysis of locality helps us understand why *gzip* can not achieve the best compression ratio. With all the dictionary rebuilt, it fails to make use of the repeats of values without much locality.

4 Algorithm and implementation

The standard Huffman coding algorithm computes an optimum prefix code given a sequence of characters from an alphabet. However, the standard Huffman coding algorithm is a two-pass algorithm; in other words,

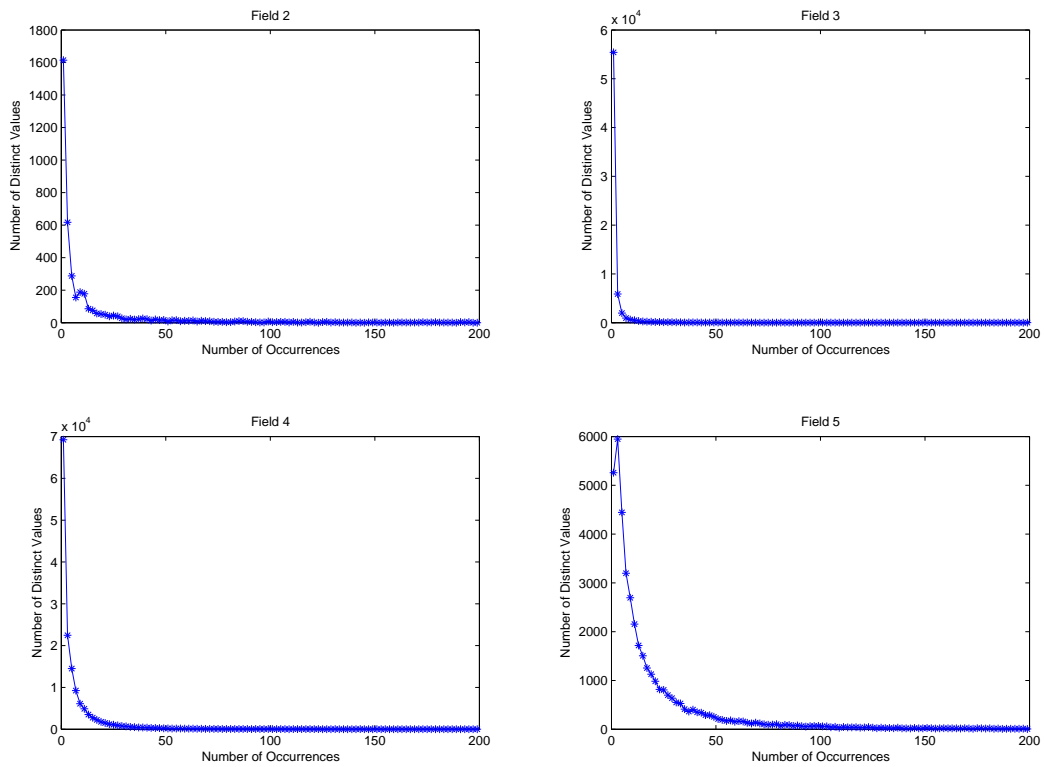


Fig. 2. Distribution of times of occurrence of field 2-5. All of them are in Zipf distribution, which suggests that encoding most frequently appearing symbols with shorter code can significantly improve the compression ratio.

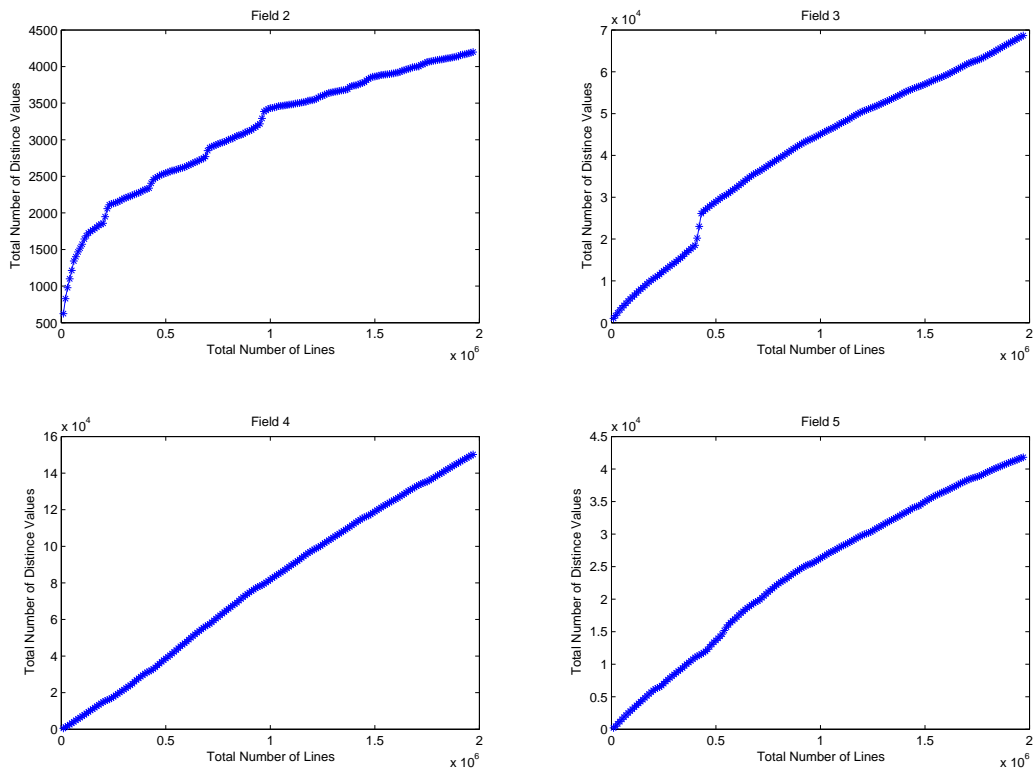


Fig. 3. The number of total lines processed v.s. the number of distinct value in each field we have found. Note that for field 3,4, and 5, it is almost on a line, which indicates that one unique value has good locality in these fields.

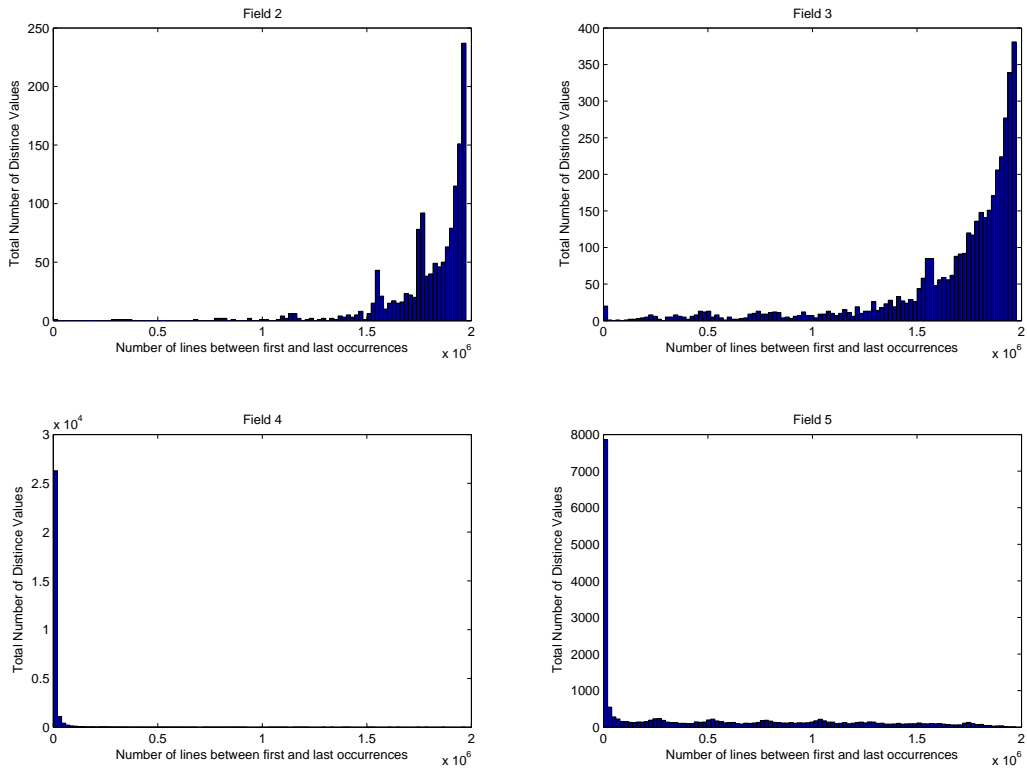


Fig. 4. The locality of distinct values. The locality is calculated by the number of lines between the first occurrence and the last occurrence of each distinct value. Better locality helps us to "forget" about history more easily.

it's *offline*. It requires one pass to compute the weights (the number of occurrences) of the characters and another pass to actually encode the data.

4.1 The FGK algorithm

There exists an online version of the Huffman coding algorithm called FGK [8] that maintains optimal prefix code as the weights of the characters change. Weight of a character at level k of the tree can be increased by one in time $O(k)$, preserving minimality of the weighted path length. Another advantage of this algorithm is that we don't have to send the actual Huffman trees as they can be dynamically constructed by the receiver.

The original FGK algorithm assumes that we know n – the number of characters in the data to be encoded. At the beginning, all characters have zero weight and they are all represented by one *zero-weight node* in the Huffman tree. When a new character appears, FGK encodes it as the "address" of the zero-weight node plus the new character. The zero-weight node is split into two nodes: one representing the new character (with weight 1) and the other is the zero-weight node. To encode a non-zero-weight character, FGK just sends its address in the Huffman tree.

After every encoded character the Huffman tree has to be updated to remain optimal. The FGK algorithm relies on the sibling property of Huffman trees: the nodes of the Huffman tree, when read top-to-bottom and right-to-left, have non-increasing weights (let the nodes be indexed in this order; the node with the highest weight (root) has the highest index). To maintain this property, we have to do the following: assume that the last character was encoded by a leaf node v_k at level k with parent v_{k-1} (with weight w). We find node v'_{k-1} with weight w with the highest index and exchange subtrees rooted at v_{k-1} and v'_{k-1} . We do the same with node v'_{k-1} and its parent and so on all the way to the root of the tree. After these $O(k)$ updates we can increase weight of v_k and of all nodes on the path to the root by one.

To get an algorithm that works on string instead of single characters will simply number the strings starting from 1. The strings are kept in a hashmap that maps strings to their ID numbers.

To encode string s , we locate its ID in the hashmap. If it's not there (i.e., we have a new string), we insert s into the hashmap with its ID equal to the max of the previous IDs plus one. We then encode the ID using the FGK algorithm. However, if we get a new string, we also have to encode the string itself; we first encode the length of the string in bytes and then the string itself. We don't encode the new string in ASCII, but use another – character-by-character – online Huffman tree. This Huffman tree is "trained" on single characters seen in the previous strings so that the new string is encoded as efficiently as possible.

4.2 Implementation and Optimization

When parsing our HTTP logs from a CSV (comma-separated-values) format, we treat each line as a tuple of seven fields (as enumerated in section 3). Each field is treated as a string and compressed/converted into the desired format. There are numerous choices for customizing the FGK algorithm to best suit our HTTP log format. We can optimize compression on a per-field basis as we know the data format and repetition frequency for each field. For example, fields with purely numeric values can be encoded as integers to reduce the number of bits used to represent the data value. Some of our customized compression experiments are enumerated in figure 5 (these results are for an input file of size 100MB).

We first tried using character-by-character Huffman encoding on all seven fields of the HTTP log. We maintained a separate Huffman tree for each of the seven fields. This method compressed a 100MB log file into 43.6MB. Applying the FGK algorithm to all seven fields provided better results, regardless of the encoding scheme applied to each new string.

The first column, representing the sequence number, contains an arbitrary 5-digit number. This value does not repeat as sequence numbers are unique for each event recorded in the log file. The second column, containing a time stamp, also does not contain repeating values. Thus, it is inefficient to encode these two fields using the string-by-string FGK algorithm. We tried encoding each sequence number as an integer and each time stamp as a long value. While this method was more efficient than using the FGK algorithm for these two fields, it was not the optimum technique. Finally, we encoded these two fields using character-by-character Huffman encoding on the values of each string (with separate Huffman trees constructed for each of the two fields). This technique works well and appears to be the optimum compression mechanism for these two fields.

Enc. for seq. number field	Enc. for time stamp field	Enc. for other 5 fields	String enc. for fields using FGK	Output file size
Char-by-char Huffman	Char-by-char Huffman	Char-by-char Huffman	-	43.6 MB
FGK	FGK	FGK	Char-by-char Huffman	41 MB
Char-by-char Huffman	FGK	FGK	Char-by-char Huffman	37.3 MB
FGK	Char-by-char Huffman	FGK	Char-by-char Huffman	26.2 MB
Char-by-char Huffman	Char-by-char Huffman	FGK	Char-by-char Huffman	24.6 MB
FGK	FGK	FGK	ascii	23 MB
Char-by-char Huffman	FGK	FGK	ascii	21 MB
FGK	Char-by-char Huffman	FGK	ascii	18.1 MB
Char-by-char Huffman	Char-by-char Huffman	FGK	ascii	16.9 MB
int	long	FGK	ascii	19.4 MB
int	Char-by-char Huffman	FGK	ascii	18.9 MB
Char-by-char Huffman	long	FGK	ascii	17.5 MB

Fig. 5. Compression Experiment Results

Subsequent columns of the HTTP log (including script name, script parameter name and value, user ID, session ID and local log filename) contain values that repeat quite often. We used the FGK algorithm (as described in the previous section) to encode these fields. Every column had a dedicated Huffman tree representing the string values of that column. In the interest of memory constraints, we limited the maximum number of strings to 500000. We applied two encoding schemes for each string. First we tried character-by-character Huffman encoding (with separate trees for each column of strings). This mechanism was very slow and did not produce optimal compression. Due to the increased number of repetitions in string values for each of these five columns, we outruled character-by-character Huffman encoding on each of these fields. Using ascii to encode each string proved to be an order of magnitude faster (approximately 5 minutes to compress a 100MB file compared to about 45 minutes using character-by-character Huffman on strings). This technique also produced an optimum compression for the data.

5 Evaluation

To evaluate our compression algorithm, we compared the our compression ratios with that of gzip. We used the optimum compression technique (character-by-character Huffman for the first two fields and dynamic huffman with ascii encoding on new strings for fields 3-7) for the HTTP log format on several different input files. The results of the comparison are shown in figure 6.

Input File Size (in B)	FGK Output Size (in B)	FGK Compr. Ratio	gzip Output Size (bytes)	gzip Compr. Ratio
105,863,452	17,791,420	16.8%	13,863,133	13.1%
324,774,647	45,109,348	13.9%	38,796,050	11.9%
638,462,771	85,803,540	13.4%	70,804,378	11.1%
686,809,719	81,386,105	11.8%	149,930,762	!! 21.8% !!
711,570,136	101,201,287	14.2%	75,290,602	10.6%

Fig. 6. Comparison of optimum FGK and gzip compression

Figure 6 clearly shows that gzip compresses the data more than our modified FGK algorithm. There are various reasons for this outcome. First of all, our code is likely to be less optimally written than the gzip code. Gzip has code has evolved over time and been subject to much more scrutiny than our code. In the interest of time (and lack of code reviews), we did not carefully optimize our code. Furthermore, we firmly believe that understanding the format of the input data allows much flexibility with the compression techniques used. It seems unlikely that gzip, without any a priori knowledge of the data format, can perform better than a good implementation of the customized FGK algorithm. There is much scope for future work in optimizing our implementation code.

We also compared the resource utilization of our modified FGK algorithm and gzip. We considered memory usage and CPU utilization over time.

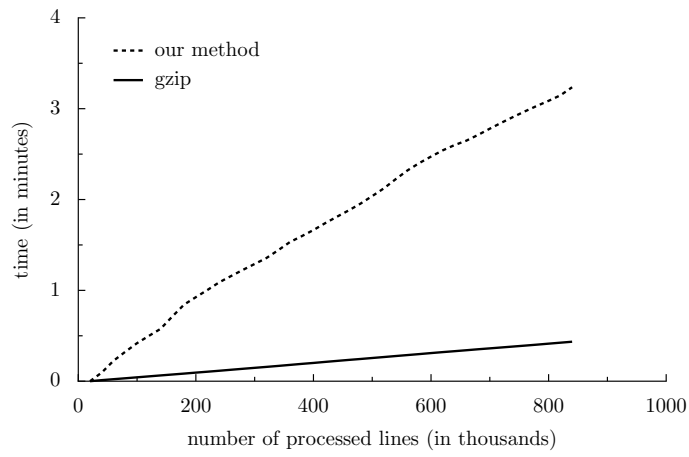


Fig. 7.

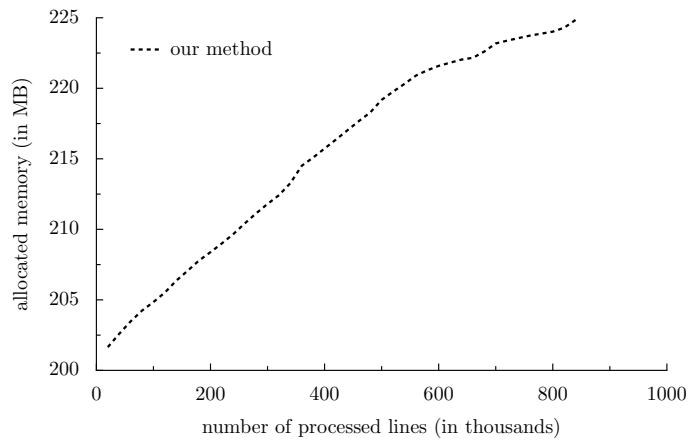


Fig. 8.

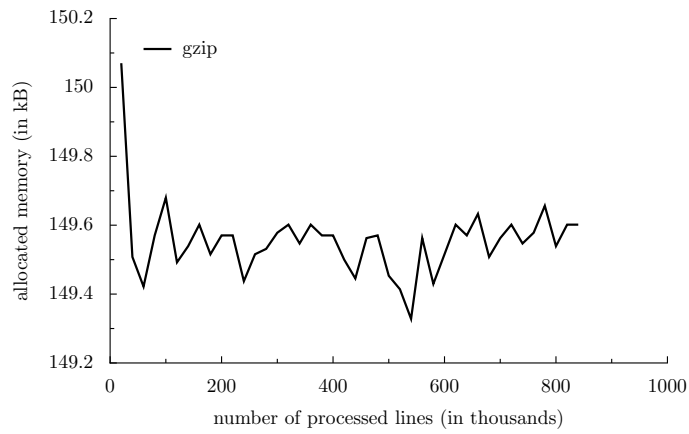


Fig. 9.

Gzip and our modified FGK technique exhibit similar behavior with respect to CPU consumption over time. Both graphs are linear in nature and span a similar range of consumption values. This similarity reflects the workload rather than the specific algorithms used.

Memory consumption of gzip is about 1000x less than that of our modified FGK algorithm. FGK consumes a significant amount of memory to maintain Huffman trees for each column of data. Perhaps given a larger input file and more memory at the disposal of the compression code, the modified FGK algorithm's memory consumption would behave asymptotically as a result of stabilized Huffman trees. However, this suggestion is merely an extrapolation based on the Huffman algorithm.

A future optimization to consider would be to periodically "flush" and recreate Huffman trees. In addition to reducing memory usage, this technique would render locally optimal Huffman trees for each part of the input file. As we plan to use modified FGK as an "online" algorithm to compress streaming files, locally optimal Huffman trees are likely to be more useful as the HTTP log data adapts to the changes in the system being monitored.

6 Conclusion

We presented an algorithm for compression of HTTP logs which exploits the known structure of the files. Our algorithm achieved similar compression ratios as gzip (used at the higher compression setting) and in one case even outperformed gzip by 10%. Also, our algorithm is *line-by-line* online because it outputs compressed data after each line of log whereas gzip has to wait until the end of the current block.

References

1. Peter F. Brown, Vincent J. Della Pietra, Robert L. Mercer, Stephen A. Della Pietra, and Jennifer C. Lai. An estimate of an upper bound for the entropy of english. *Comput. Linguist.*, 18(1):31–40, 1992.
2. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.
3. Xin Chen, Sam Kwong, and Ming Li. A compression algorithm for dna sequences and its applications in genome comparison. In *RECOMB '00: Proceedings of the fourth annual international conference on Computational molecular biology*, page 107, New York, NY, USA, 2000. ACM Press.
4. Xin Chen, Ming Li, Bin Ma, and John Tromp. DNACompress: fast and effective DNA sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002.
5. Gordon V. Cormack. Data compression on a database system. *Commun. ACM*, 28(12):1336–1342, 1985.
6. Antaeus Feldspar. An explanation of the deflate algorithm. Web, 1997.
7. S. Grumbach and F. Tahi. Compression of dna sequences. In *Data Compression Conference, 1993. DCC '93*. IEEE, March 1993.
8. Donald E. Knuth. Dynamic huffman coding. *J. Algorithms*, 6(2):163–180, 1985.
9. Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Comput. Surv.*, 19(3):261–296, 1987.
10. Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.
11. T Matsumoto, K Sadakane, and H Imai. Biological sequence compression algorithms. *Genome Informatics*, 2000.
12. Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2003.
13. E. Rivals, J.-P. Delahaye, M. Dauchet, and O. Delgrange. A guaranteed compression scheme for repetitive dna sequences. In *DCC '96: Proceedings of the Conference on Data Compression*, page 453, Washington, DC, USA, 1996. IEEE Computer Society.
14. Jeffrey Scott Vitter. Design and analysis of dynamic huffman codes. *J. ACM*, 34(4):825–845, 1987.
15. Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.