usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems

Junyu Wei and Guangyan Zhang, *Tsinghua University;* Yang Wang, *The Ohio State University;* Zhiwei Liu, *China University of Geosciences;* Zhanyang Zhu and Junchao Chen, *Tsinghua University;* Tingtao Sun and Qi Zhou, *Alibaba Cloud*

https://www.usenix.org/conference/fast21/presentation/wei

This paper is included in the Proceedings of the
19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems

Junyu Wei[†], Guangyan Zhang[†,*] Yang Wang[‡], Zhiwei Liu[¶], Zhanyang Zhu[†],
Junchao Chen[†], Tingtao Sun[§], Qi Zhou[§]
[†]*Tsinghua University,* [‡]*The Ohio State University,* [¶]*China University of Geosciences,* [§]*Alibaba Cloud*

Abstract

Given the tremendous scale of today's system logs, compression is widely used to save space. While parser-based log compressor reported promising results, we observe less intriguing performance when applying it to our production logs.

Our detailed analysis shows that, first, some problems are caused by a combination of sub-optimal implementation and assumptions that do not hold on our large-scale logs. We address these issues with a more efficient implementation. Furthermore, our analysis reveals new opportunities for further improvement. In particular, numerical values account for a significant percentage of space and classic compression algorithms, which try to identify duplicate bytes, do not work well on numerical values. We propose three techniques, namely delta timestamps, correlation identification, and elastic encoding, to further compress numerical values.

Based on these techniques, we have built LogReducer. Our evaluation on 18 types of production logs and 16 types of public logs shows that LogReducer achieves the highest compression ratio in almost all cases and on large logs, its speed is comparable to the general-purpose compression algorithm that targets a high compression ratio.

1 Introduction

Most systems log internal events for various reasons, such as diagnosing system errors [9, 36, 48], profiling user behaviors [10, 11, 28], modeling system performance [2, 15], and detecting potential security problems [12, 37].

In today's datacenters, the size of such logs can grow large. In 2016, Feng. et al reported their system can generate 100GB of logs per day [13], and in 2019, this number increased to 2TB per day [30]. AliCloud, a major cloud provider and our collaborator, can generate several PBs of logs per day.

These logs usually need to be stored for a long time for multiple reasons: sometimes an anomaly is detected much later than it was logged, so the developer needs to analyze the past logs [1, 9, 21, 24]; certain analysis may require statistics over a long period of time to generate a conclusion [10, 14, 45]; for the purpose of the audition, local laws require a cloud provider to store these logs for a certain amount of time. As a result, AliCloud has decided to store its logs for 180 days. Considering it's generating several PBs of logs per day, storing these logs is a considerable overhead even for a big company.

To reduce log size, a classic solution is to compress these logs. General-purpose lossless compression methods, such as LZMA [40], gzip [6], PPMd [4], and bzip [41], can compress a file by identifying and replacing duplicate bytes. A number of recent works observe that most system logs are generated by adding variables to a string template (e.g. printf("value=%d", v)), and thus by separating them, they only need to store the variables [23, 30, 31, 33, 46]. We call these approaches *parser-based log compression* in this paper.

While these works report promising results, we observe less intriguing performance when applying this method to production logs from AliCloud: when applying Logzip [30], the latest one in this line of work, to our logs, we find it's seven times slower than LZMA, a general-purpose compression method targeting a high compression ratio, and Logzip's compression ratio is worse than LZMA on 13 out of the 18 types of the logs.

To understand whether such problems are fundamental or due to engineering issues, we perform a detailed analysis of Logzip. Our analysis shows that, first, some problems are indeed caused by a combination of sub-optimal implementation and undesirable limits: Logzip is implemented in Python and uses several notoriously slow libraries and data structures like Pandas DataFrame [5]; Logzip limits a log entry to have no more than 5 variables, which is too small for our logs; increasing the limit will further slow down Logzip, which is already seven times slower than LZMA. We address these issues by re-implementing the whole algorithm in C/C++, which significantly improves the compression speed. It further allows us to remove the limit on the number of variables to improve the compression ratio as well.

Second, our analysis reveals new opportunities for further

[*] Corresponding author: gyzh@tsinghua.edu.cn

improvement. At a high level, Logzip uses general-purpose compression methods to further compress variables: while this works well for string variables, it does not work well for numerical variables, since general-purpose compression methods target finding duplicate bytes, and there are not much duplication in numerical data in our experiments. We incorporate three techniques to further compress numerical data:

- We observe timestamps account for over 20% of the space in 8 out of 18 types (even 70% in one type) in the compressed files, mainly because AliCloud needs micro-second level timing information to accurately identify the order of events, for the purposes like performance debugging and resolving conflicts. To compress timestamps, we use the classic differential method to compute and store the delta value of two consecutive timestamps.

- We observe that numerical data are often correlated. A typical example is in an I/O log: when the user performs sequential I/Os, the offset of the next I/O is equal to the sum of the offset and length of the previous I/O. Such correlation provides an obvious opportunity to further compress numerical data. Following this idea, we have developed a novel algorithm to identify simple numerical correlation in log samples and apply the found rules during compression.

- We observe most numerical values are small and using fixed-length coding (e.g. 32 bits for an integer) will generate many 0 bits at the beginning. We propose *elastic coding*, which represents a number with an elastic number of bytes, to trim leading zeroes. Compared to general-purpose compression algorithms, elastic coding is more efficient at trimming leading zeroes; compared to fixed-length coding, elastic coding can reduce the length when the value is small but may increase the length when the value is large, which is a beneficial trade-off given our observation.

By combining all the efforts mentioned above, we have built LogReducer. We have applied LogReducer to 18 types of AliCloud logs (1.76TB in total) and 16 types of public logs (77GB in total). Compared with LZMA, LogReducer can achieve 1.19× to 5.30× compression ratio on all cases and 0.56× to 3.16× compression speed on logs over 100MB (LogReducer is comparably slower on smaller logs because of its initialization overhead). Compared with Logzip, LogReducer can achieve 1.03× to 4.01× compression ratio and 2.05× to 182.31× compression speed. Such results have confirmed that, with proper implementation and optimization, parser-based log compression is promising to compress large-scale production logs.

The contribution of this paper is three-fold. First, we study why state-of-the-art parser-based compression methods do not perform well on our production logs. Second, based on the study, we build LogReducer by improving the implementation of existing methods, applying proper techniques based on the characteristics of the logs, and introducing new techniques. Finally, we demonstrate the efficacy of LogReducer on a variety of logs. LogReducer is open source [39].

2 Background

2.1 Structure of Cloud Logs

We collect a large set of logs generated in AliCloud. They are from different applications developed by different teams, which serve for various purposes, e.g., warning and error reporting, infrastructure monitoring, user behavior tracing, and periodical summary. Table 1 shows examples of three types of logs. Samples of all 18 types can be found in [38].

The basic structure of these logs contains three parts: header, template and variable. A header includes the timestamp and the corresponding log level. The header is added by AliCloud's logging system automatically and its format is relatively static, which allows us to use a regular expression to separate the header from the remaining part. The rest of this section mainly discusses how to parse the remaining part into templates and variables.

Templates are the formalized output statements of logs. In Log F, "Write chunk %s Offset %d Length %d" and "Read chunk %s Offset %d" are two templates. Variables refer to the part which varies in each instance of the same template. In Log F, they include "3242513_B", "339911", "11" ,etc.

User behavior tracing (Log F)
[2019-08-27 15:21:24.456234] [INFO] Write chunk: 3242513_B Offset: 339911 Length: 11
[2019-08-27 15:21:24.463321] [INFO] Read chunk: 3242514_C Offset: 272633
[2019-08-27 15:21:24.464322] [INFO] Write chunk: 3242512_F Offset: 318374 Length: 7
[2019-08-27 15:21:24.474433] [INFO] Write chunk: 3242513_B Offset: 339922 Length: 55
Infrastructure monitoring (Log D)
[2018-01-12 08:53:12.188370] [10593] project:393 logstore: XDoFiqnlmZd shard:78 inflow:3376 dataInflow:18869
[2018-01-12 08:53:12.188390] [10593] project:656 logstore: lOdMafL31Pg shard:37 inflow:7506 dataInflow:42712
Warning and error reporting (Log Q)
Aug 28 03:09:02 h10c10322.et15 su[57118]: (to nobody) root on none
Aug 28 03:09:02 h10c10322.et15 su[57118]: session opened for user nobody by (uid=0)

Table 1: Examples of logs in AliCloud.

2.2 Parser-based Log Compressor

Parser-based log compression first uses a *log parser* to identify the template of each log entry and extract the corresponding variables; it then replaces the template string with a template ID to save space; it finally applies general-purpose compression methods to variables to further reduce space.

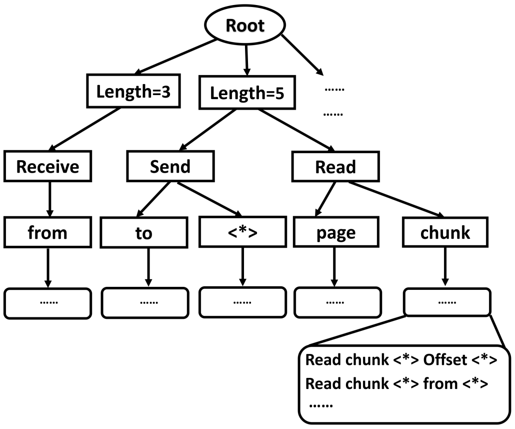


Figure 1: Parser tree architecture.

Log parser can be implemented using longest common string [8], clustering [35], and parser tree [23], among them parser tree shows better effectiveness [49]. Here we first present the concept of parser tree and then show how to build the parser tree and use it to separate templates and variables.

Parser tree. Given a list of templates and log entries, a naive approach to match the entry to a template is to compare the entry to each template and find the template which is most similar to the entry. However, when there are many templates, such one-by-one comparison is inefficient.

To improve the efficiency of template matching, several works [23, 30] use a parser tree to facilitate the matching: as shown in Figure 1, each leaf of the parser tree is a group of templates sharing the same length (i.e., the number of tokens in a log message); the first layer of internal nodes use the length of the log entry to categorize the entry; the following layers of internal nodes form multiple paths, each of them leads to a leaf node in the parser tree. Both the internal node and the template use "<*>" to represent a variable.

Assuming we have built a parser tree as shown in Figure 1, and we have a log entry "Read chunk 3242514_C Offset 272633": since its length is 5, we will first go to the internal node "Length=5"; since its first token is "Read", we will then go to the internal node "Read"; and then "chunk"; finally we will compare the log entry with each template in the leaf node and find "Read chunk <*> Offset <*>" is closest to the log entry, so we will choose this template and identify "3242514_C" and "272633" as variables.

Building the parser tree. Parser-based log compressor first builds the parser tree by parsing a sample of the log entries. For each log entry, the log parser performs four steps, and we use log entry "Read chunk 3242514_C Offset 272633" as an example to explain these steps. In the beginning, the parser tree just has one root node.

In the first step, the log parser uses predefined split characters, such as empty space or comma, to split a log entry into a list of strings called tokens. In our example, the raw log message will be divided into "Read", "chunk", "3242514_C", "Offset", "272633" accordingly.

In the second step, the log parser will check whether the internal node of the length exists (Length=5 in our case). If not, the log parser will create a new internal node. Finally, it moves to the corresponding internal node.

In the third step, the log parser traverses the tree according to the tokens in the log entry and moves to corresponding internal nodes ("Read" and "chunk" in our example). After reaching the limitation of tree depth, it reaches a leaf node, which contains a group of templates. If the corresponding internal node does not exist, the log parser will build the internal node and add the node to the prefix tree.

Definition 1. ***Similarity between log L and template T*** *(l_i is the ith token in L; t_i is the ith token in T; $\phi(a,b) = 1$ if $a = b$, otherwise $\phi(a,b) = 0$, $|\cdot|$ is the number of tokens in a log)*

$$Similarity(L,T) = \frac{\sum \phi(l_i,t_i)}{|L|} \qquad (1)$$

In the fourth step, the log parser searches for the most similar template in this template group using a similarity function defined in equation 1. If the largest similarity is smaller than a threshold ε, the log parser will create a new template, which is the same as the log entry. Note that at this moment the log parser cannot tell which tokens of the log entry are variables. If the largest similarity is larger than ε, the log parser will regard this log entry as an instance of the matching template and update the template accordingly to mark different parts as variables.

For example, suppose the log parser first parses "Read chunk 3242514_C Offset 272633": since there is no template yet, the log parser will create a new template "Read chunk 3242514_C Offset 272633". Then suppose the log parser processes "Read chunk 3242514_B Offset 268832": its similarity to "Read chunk 3242514_C Offset 272633" is 0.6, so if ε is smaller than 0.6, the log parser will consider "Read chunk 3242514_B Offset 268832" as an instance of "Read chunk 3242514_C Offset 272633" and update the template into "Read chunk <*> Offset <*>"; if ε is larger than 0.6, the log parser will treat "Read chunk 3242514_B Offset 268832" as a new template.

Compressing logs. Then the compressor uses the parser tree to compress logs [30]. The procedure is similar to building a parser tree, except that in this phase, the compressor will not update the parser tree. It will first utilize the parser tree to try to match each log entry to a template. If a match is found, the log entry will be converted to the template ID and the variables; if no template is matched, the log entry will be regarded as a mismatch and will not be converted.

Afterward the compressor will group log entries according to their template IDs and store their variables in a column manner, i.e., it first stores the first variable of each log entry in the group, then stores the second variable, and so on. The column-based storage is based on the observation that variables at the same position of the same template are prone to have more redundancy, so that sliding window based algorithms such as LZ77 [40] will have more chances to trim redundancy. Finally the compressor concatenates everything and compresses it with a general-purpose compressor.

3 Restore the Promise of Parser-based Log Compression

We tested Logzip, the most recent parser-based log compression implementation, on 18 types of AliCloud's production logs. First, we find the value of the similarity threshold ε has a critical impact on the performance of Logzip. When using Logzip's default value 0.5, we find Logzip takes nearly 20 days to build the parser tree and can generate tens of thousands of templates, which impairs both the speed and the compression ratio. We tuned this value on our logs and found a value of 0.1 works well for almost all of our logs. This is due to the following reason: Logzip was mainly tested on PC logs, which usually are short and only contain a small number of variables; AliCloud's logs usually have more variables (see Table 2), and thus logs within the same templates are quite different from each other, i.e., they share only a small number of common tokens. Therefore, to extract the correct templates, we need a smaller ε. Manual tuning is always undesirable in a production environment: while we find the value 0.1 works well, for environments with more versatile logs, an automatic tuning procedure might be beneficial.

We continued testing Logzip with $\varepsilon = 0.1$ and found the result is still not ideal in terms of both compression ratio and compression speed: compared with LZMA, the general-purpose compression algorithm that can achieve the highest compression ratio on our logs, Logzip is seven times slower, and on 13 out of the 18 types of logs, Logzip's compression ratio is lower (§6.1). Our detailed analysis revealed a correlated problem between compression ratio and speed:

Logzip implementation assumes that a log entry usually has no more than five variables: for log entries with more than five variables, Logzip will regard content after the first variable as a large variable, and feed it to the general-purpose compressor. However, as shown in Table 2, this assumption does not hold on most of our logs (many have over 10 variables and one has 176 variables). As a result, Logzip loses its effectiveness on our logs, which can explain its poor compression ratio.

We tried to increase this limit and found it further exacerbates the speed problem of Logzip: while Logzip is already seven times slower than LZMA with the limit of five variables, increasing the limit to 256 will make Logzip unbearably slow, which might be the reason Logzip sets a small limit. We profiled Logzip to understand its bottleneck and found it has used several notoriously slow libraries or data structures including Pandas DataFrame, Python array append, etc. To address this problem, we re-implement the whole algorithm in C/C++ and dramatically improve the speed. The increased speed allows us to remove the limit of five variables as well. We further improve the speed with the following techniques:

Cutting the Parser Tree. We have observed that the total number of templates in our production logs is usually small: as shown in Table 2, 15 types of logs have less than 50 templates. If we group them based on length, the number of templates in one group is even smaller.

The reason behind this observation is that these cloud logs are generated by developers in the operation engineering group of AliCloud, and thus their patterns are relatively static compared to logs generated by cloud users.

Based on this observation, we cut the parser tree into only one layer in the compression phase: we only take length into consideration and we store templates with the same length together and search them one by one. This optimization has improved compression speed and avoided the tuning of the depth of the parser tree.

Batch processing. If we need to compress a large number of small log files, and we start one compressor process to compress each log, we observe the overhead to start and stop processes could slow down the whole compression significantly. Therefore, we allow our compressor to take a batch of log files as inputs and compress them together.

With all the efforts mentioned above, we have restored the promise of parser-based compression: as shown in §6.2, compared with LZMA, our implementation (i.e., LogReducer-B) can achieve 1.16× - 3.73× compression ratio and 0.51× - 2.01× compression speed.

4 Further Compressing Numerical Variables

We have done a detailed analysis on the compressed files generated by the previous step and found that in 10 of the 18 types of logs, numerical variables account for over 50% space after compression; for other types of logs the rate is at least 20%; in three cases, the rate is over 80% (Table 3). This is because 1) our logs have a large number of numerical variables and 2) general-purpose compression methods, which try to identify redundant bytes, do not work well with numerical variables.

4.1 Compressing Timestamps

Our analysis shows that timestamps are the first dominant numerical data in our logs. As shown in Table 3, in eight types of logs, timestamps account for more than 20% of space. In one case, this rate can reach close to 70%.

This is because AliCloud needs precise timing information to order events, for purposes like debugging and auditing. In its environment, system logs can be generated at a high

Log type	Log A	Log B	Log C	Log D	Log E	Log F	Log G	Log H	Log I
# of templates	42	29	3	6	74	7	202	39	48
Avg. # of variables	14	5	10	13	22	12	7	57	176
Log type	**Log J**	**Log K**	**Log L**	**Log M**	**Log N**	**Log O**	**Log P**	**Log Q**	**Log R**
# of templates	29	10	4	16	49	1	20	43	130
Avg. # of variables	3	46	7	9	4	13	22	2	5

Table 2: Template information on 18 types of logs in AliCloud.

Log type	Log A	Log B	Log C	Log D	Log E	Log F	Log G	Log H	Log I
Number Rate(%)	46.63	68.51	52.19	82.86	51.69	88.42	33.92	45.51	31.65
Time Rate(%)	36.75	38.97	15.28	15.49	10.42	10.07	22.88	31.23	4.22
Log type	**Log J**	**Log K**	**Log L**	**Log M**	**Log N**	**Log O**	**Log P**	**Log Q**	**Log R**
Number Rate(%)	39.27	69.85	24.89	53.54	53.40	78.47	27.30	29.36	84.96
Time Rate(%)	26.96	7.18	9.32	21.53	25.63	14.79	15.77	14.90	68.27

Table 3: Space consumption of numerical variables and timestamps in compressed file.

speed, up to one million entries per second, which motivates AliCloud to record timestamps at micro-second level. As a result, first, it takes more bits to store the microsecond level timestamps than millisecond or second level timestamps, and second, there is not much redundancy in the timestamps.

To compress timestamps, we use the classical differential method, which records the delta value between two consecutive timestamps. This method can significantly reduce the size of timestamps when the target system generates logs frequently, namely the delta value will be small. By using this method, we can reduce the space overhead and pass a much smaller number to the general-purpose compressor, which can improve both compression ratio and compression speed.

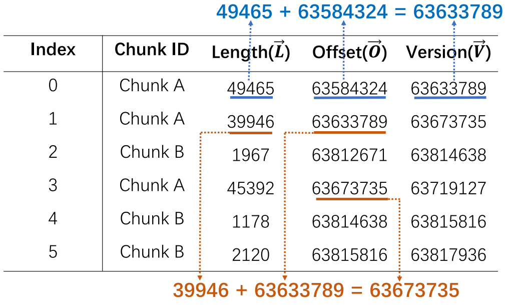
4.2 Correlation Identification and Utilization

We observe numerical variables sometimes are correlated. For example, in an I/O trace, if the user performs sequential I/Os, the offset of the next I/O will be equal to the sum of the offset and length of the previous I/O.

Such correlation provides an obvious opportunity to compress numerical data. If most values of certain variables follow certain kind of correlation, we only need to store how values deviate from the correlation in a *residue vector*; since most values of the residue vector will be zeroes, they will be effectively compressed by a general-purpose compressor.

For example, for logs of type Log F with templates "Write Chunk <*> Length <*> Offset <*> Version <*>", we can extract four variables from the template and three of them are numerical variables, namely $\vec{L}$ (Length), $\vec{O}$ (Offset), $\vec{V}$ (Version). In our logs, we find three types of correlations in these variables. Note that the values of each variable form a vector since there are multiple log entries.

- Inter-variable correlation: Version is often equal to the sum of Offset and Length, namely $\vec{V} = \vec{L} + \vec{O}$, and its residue vector is $\vec{V} - \vec{L} - \vec{O}$.



Index	Chunk ID	Length($\vec{L}$)	Offset($\vec{O}$)	Version($\vec{V}$)
0	Chunk A	49465	63584324	63633789
1	Chunk A	39946	63633789	63673735
2	Chunk B	1967	63812671	63814638
3	Chunk A	45392	63673735	63719127
4	Chunk B	1178	63814638	63815816
5	Chunk B	2120	63815816	63817936

Figure 2: Numerical correlations observed on Log F.

- Intra-variable correlation: Lengths of the same Chunk ID are often close. We can compute its residue vector as $\vec{L}[i] - \vec{L}[i-1] = \vec{\Delta L}$.
- Mixed correlation: if the user is performing sequential I/Os to a chunk, then its lengths and offsets have the following correlation: $\vec{O}[i] = \vec{O}[i-1] + \vec{L}[i-1]$. Its residue vector is $\vec{O}[i] - \vec{O}[i-1] - \vec{L}[i-1] = \vec{\Delta O} - \vec{L}$.

Correlation identification. We propose a novel method to identify such correlation. The goal of correlation identification process is to find the relationship across and within different variables so that we can represent some variables with residue vectors, which can be compressed more effectively. To achieve this goal, we first enumerate different combinations of variables, the IDs to group different entries (e.g. ChunkID in Figure 2), and the aforementioned correlation rules, and compute the corresponding residue vectors. Then we select vectors from the original vectors and the residue vectors with the goals of 1) maximizing compression ratio and 2) being able to recover all original vectors.

The whole identification process is illustrated in algorithm 1, which maintains three sets: the target set ψ; the recover set $\mathbb{R}$ including all original vectors that can be recovered from the current ψ; the total candidate set $\mathbb{T}$ including all candidate vectors. One of its key data structure is a map from the candidate vectors to original vectors: $map(\vec{C})$ will return all original vectors that $\vec{C}$ is built from (e.g., $map(\vec{A}-\vec{B}) = \vec{A}$ and $\vec{B}$).

Algorithm 1 **Correlation identification algorithm**

```
1:  Recoverable set $\mathbb{R} = \emptyset$
2:  Final vector set $\Psi = \emptyset$
3:  Initialize candidate set $\mathbb{T}$
4:  repeat
5:      $\mathbb{C} = \{\vec{C} \in \mathbb{T} : |map(\vec{C}) - \mathbb{R}| = 1\}$
6:      $\vec{C}_{min}$ = vector with the smallest entropy in $\mathbb{C}$.
7:      $\Psi \leftarrow \Psi \cup \vec{C}_{min}$
8:      $\mathbb{R} \leftarrow \mathbb{R} \cup map(\vec{C}_{min})$
9:  until $\mathbb{R}$ contains all original vectors
10: Output $\Psi$
```

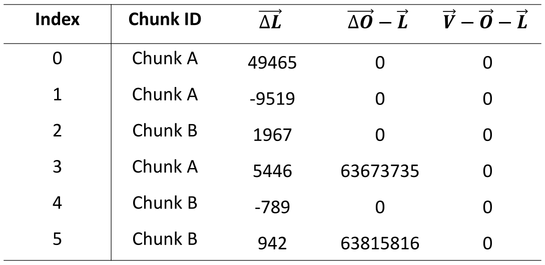
The algorithm works in iterations: in each iteration, it first tries to find all candidate vectors $\vec{C}$ that can recover one more variable compared to the current recoverable set $\mathbb{R}$ (line 5); then among them, it chooses the one with the highest compression ratio (line 6). Here we predict the compression ratio of a candidate using its Shannon Entropy [26], defined in Definition 2; finally it updates Ψ and $\mathbb{R}$ accordingly (lines 7 and 8); it repeats this until Ψ can recover all original vectors (line 9). The cost of enumeration is acceptable, since it is performed on samples of logs.

Definition 2. ***Entropy for a variable vector.*** *S_A denotes the set of all values appearing in $\vec{A}$ and #(s) denotes the number of times the value s appears in $\vec{A}$.*

$$E(\vec{A}) = -\sum_{s \in S_A} \frac{\#(s)}{|A|} \log \frac{\#(s)}{|A|} \quad (2)$$

In Figure 2, Ψ will finally contain three residue vectors, namely:$\{\Delta\vec{L}, \Delta\vec{O}-\vec{L}, \vec{V}-\vec{L}-\vec{O}\}$. These three residue vectors are enough to recover all original variable vectors $\vec{L}, \vec{O}, \vec{V}$, and will have a higher compression ratio than the original variable vectors.

Correlation utilization. The output of the training phase for numerical correlations is the target vector set Ψ. In the compression phase, we calculate each residue vector in set Ψ and discard original vectors that do not appear in Ψ. If we apply three correlations in Ψ to our example in Figure 2, the result is shown in Figure 3. As one can see, for variables that perfectly match certain rules, their residue vectors contain many zeroes; even for those that do not perfectly match the rules, their values are smaller, which facilitates the elastic encoder discussed in the following section.



Index	Chunk ID	$\Delta\vec{L}$	$\Delta\vec{O}-\vec{L}$	$\vec{V}-\vec{O}-\vec{L}$
0	Chunk A	49465	0	0
1	Chunk A	-9519	0	0
2	Chunk B	1967	0	0
3	Chunk A	5446	63673735	0
4	Chunk B	-789	0	0
5	Chunk B	942	63815816	0

Figure 3: Processing result of logs in Figure 2.

4.3 Elastic Encoder

The simplest way to represent numerical variables is to use fixed number of bytes (e.g. 4 bytes to represent an integer, 8 bytes to represent a long value, etc). However, if most numbers are small, these bytes will contain many leading zeroes (for positive numbers) or ones (for negative numbers). General-purpose compression may be able to find such consecutive zeroes or ones, but since it needs to search for such zeroes/ones and store additional metadata to record the length of zeroes/ones, we design a dedicated encoding algorithm to trim leading zeroes.

To efficiently exploit such opportunity, we apply an elastic encoding method to store numbers according to their size. We cut the 32-bit integer into 7-bit segments and add one bit to each segment, indicating whether the segment is the last segment (1 means it is the last). Then we discard the prefix of segments containing only zeroes. Here we choose the number 7 because, after adding one bit, each segment takes a byte, which is easy to handle.

For a negative number represented by two-complement encoding, it is not trivial to just change all ones to zeroes, since the leading ones include the first bit which indicates this number is a negative number. To overcome this problem, we adopt a shifting operator [43] to move the first bit to the last position and reverse all other bits if the original number is negative. By adopting this method, we will process negative numbers in the same way as processing positive ones.

By using elastic encoding, we will trim the leading zeros/ones at the cost of adding one-bit metadata for every remaining 7 bits. Therefore, the smaller the number is, the more redundancy we can trim. More precisely, for an integer between $[-2^{7n}, -2^{7(n-1)}) \cup (2^{7(n-1)}-1, 2^{7n}-1](0 < n < 6)$, elastic encoding can save $(32-8n)$ bits compared with using fixed 32 bits. In our logs, we find this method can save 24 bits (i.e., $n = 1$) for more than 60% of the numbers.

Note that both delta timestamps and applying correlation contribute to the effectiveness of elastic encoding since these techniques tend to make numbers smaller.

5 Architecture and Implementation

Based on the observations and ideas mentioned previously, we have built LogReducer, a parser-based log compressor,

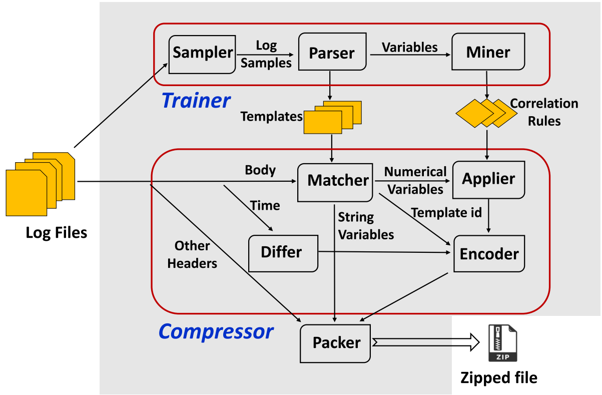


Figure 4: LogReducer architecture.

with about 3,000 lines of C/C++ code. Figure 4 shows its architecture. LogReducer contains two phases, training phase and compression phase.

Training. Training phase is done over sampled data. It uses a parser to extract templates (§2) and a correlation miner to find possible correlations (§4.2). At the end of this phase, LogReducer may find a list of templates and correlation rules.

Just like any other methods relying on sampling, we expect such samples to capture the properties of real logs as much as possible. Traditional parser-based compression methods like Logzip only need to extract templates during the training phase, and since the template of a log entry only depends on the entry itself, these methods can use random sampling. LogReducer, however, tries to identify data correlation across adjacent log entries, and random sampling will lose such relationship. To address this challenge, we first randomly pick several starting points and then choose a contiguous sequence of logs from each starting point. This method shows good performance on both extracting templates and identifying correlations across adjacent log entries.

Compression. In the compression phase, for each log entry, LogReducer will first extract its header. LogReducer will further extract the timestamps from the headers and compute the delta values of consecutive timestamps (§4.1). Then LogReducer will try to match the log entry to templates using the parser (§3) and apply founded correlations to numerical variables (§4.2). Then LogReducer will encode all numerical data, including timestamps, numerical variables, and template IDs, using elastic encoder (§4.3). Finally, LogReducer will pack all data using LZMA since we find it can almost always achieve the highest compression ratio on our logs.

In order to illustrate the whole process of compression, we exhibit a complete compression case. Suppose we have four input log entries of Log F shown in Table 1 (§2) and the templates and correlations founded in the training phase.

LogReducer first extracts their headers and matches their bodies to templates. The results are shown in Table 4, in which each log entry is divided into three parts, namely log header, template ID, and corresponding variables. Here the second log entry belongs to template:"Read chunk <*> Offset:<*>", whose template ID is 2 and the other three log entries belong to template: "Write chunk <*> Offset:<*> Length:<*>", whose template ID is 1. As a result, template 2 has two variables and template 1 has three variables.

Headers	Template ID	V1	V2	V3
[2019-08-27 15:21:24.456234] [INFO]	1	3242513_B	339911	11
[2019-08-27 15:21:24.463321] [INFO]	2	3242514_C	272633	-
[2019-08-27 15:21:24.464322] [INFO]	1	3242512_F	318374	7
[2019-08-27 15:21:24.474433] [INFO]	1	3242513_B	339922	55

Table 4: Extracting headers and matching templates.

Then LogReducer will compute the difference of adjacent timestamps and utilize correlations over numerical variables. Table 5 shows the result of these steps: all timestamps become much smaller except the first one; since LogReducer identifies the sequential access pattern for 3242513_B, it does not need to store the offset of the second access and we calculate the delta result for write length of the same chunk (i.e., log entry 4). Finally LogReducer encodes all numerical results using elastic encoder, organizes all variables in a column manner, and packs them with LZMA.

Time	Other Header	Template ID	V1	V2	V3
2019 08 27 15 21 24 456234	[INFO]	1	3242513_B	339911	11
0 0 0 0 0 0 7087	[INFO]	2	3242514_C	272633	-
0 0 0 0 0 0 1001	[INFO]	1	3242512_F	318374	7
0 0 0 0 0 0 10111	[INFO]	1	3242513_B	0	44

Table 5: Computing the delta values of timestamps and applying correlation.

Others. In cloud environments, logs usually contain a large amount of information. To make such information easy to be understood by humans, logs often need to be truncated into several lines. Such multi-line log entries do not exist in the PC logs where Logzip was tested upon. Logzip simply treats them as a mismatch, which obviously reduces its effectiveness.

We calculate the rate of multi-line logs in our production logs and find in Log H and Log R, the rate of multi-line logs have reached up to 5% of the whole size. To support multi-line log entries, we do not split log entries based on a new line symbol; instead, we split log entries based on log headers, as we discussed in §2.1. Doing so achieves both higher compression ratio and higher compression speed.

Besides, to improve the generality of LogReducer beyond AliCloud logs, we implement a head-format adaptor: based on the assumption that the number of tokens in the head is static for the same type of logs, this adaptor tries to treat the first n tokens as the head (it tries $n = 1$ to 10 in our experiments) to see which n value can achieve the best compression ratio.

6 Evaluation

Our evaluation tries to answer three questions:

- What is the overall performance of LogReducer in terms of compression ratio and compression speed on AliCloud logs? (§6.1)
- What is the effect of each individual technique of LogReducer? (§6.2)
- How does LogReducer perform on logs beyond AliCloud logs? (§6.3)

To answer these questions, we measure the performance of LogReducer on 18 types of production logs from AliCloud with a total size of about 1.76TB (Table 6). We measure both the compression ratio (i.e., $\frac{Original\ size}{Compressed\ size}$) and the compression speed (MB/s).

For comparison, we also measure the performance of two general-purpose compression algorithms (gzip and LZMA) and two log-specific compression algorithms (LogArchive [3] and Logzip [30]). gzip is a classical compression tool. It targets high compression speed instead of a high compression ratio. We use "tar" [7] command to compress log dataset with gzip. LZMA is a well-studied general-purpose compression method based on LZ77 [40] algorithm. It has a high compression ratio but a relatively low compression speed. We use 7z [50] to compress the log data with LZMA. LogArchive is a bucket-based log compression method. We use its open-source code to compress our logs data [17]. Logzip is the latest implementation of parser-based compressor. We use its open-source code [19]. Note that as discussed in §3, we change the ε value of Logzip from 0.5 to 0.1.

Testbed. We perform all experiments using 4 Linux servers, each with 2× Intel Xeon E5-2682 2.50GHz CPUs (with 16 cores), 188GB RAM, and Red Hat 4.8.5 with Linux kernel 3.10.0. For each method, we use 4 threads to compress the log data in parallel and sum their total time.

6.1 Overall Performance

Compression ratio. As shown in Figure 5(a), LogReducer has the highest compression ratio on all logs. It can achieve 1.54× to 6.78× compression ratio compared to gzip, 1.19× to 4.80× compared to LZMA, 1.11× to 3.60× compared to LogArchive, and 1.45× to 4.01× compared to Logzip.

In our experiments, Logzip failed on Log I; LogArchive failed on Log I and Log J. Both of these two logs have much longer log entries than others, which causes buffer overflow in Logzip and LogArchive. We use LZMA to compress failed logs, since it is the default setting of Logzip and LogArchive.

LogReducer can compress all 1.76TB log dataset into 34.25GB, which takes only 1.90% space after compression. gzip, LZMA, LogArchive, and Logzip can compress all 1.76TB into 152.03GB, 107.22GB, 91.54GB, and 89.86GB respectively. As a result, their space consumption is 4.44×, 3.13×, 2.67× and 2.62× as much as LogReducer respectively.

We further compute the improvement of LogReducer over the best of the other four algorithms on all 18 logs. LogReducer has the highest improvement on Log F and lowest improvement on Log L due to the following reasons: Log F has several typical correlations we discussed in §4.2 and LogReducer can identify them and trim redundancy effectively, while other works cannot utilize such correlation. Log L has a low percentage of numerical values (only 24.89%) and timestamps (only 9.32%), which means the new techniques introduced by LogReducer are not very effective.

Compression speed. As shown in Figure 5(b), LogReducer is 4.01×-182.31× as fast as Logzip and 4.49×-11.65× as fast as LogArchive. LogReducer is comparable to LZMA in compression speed (0.56×-3.16×): it is slower than LZMA on 8 out of 18 logs; in some special cases (Log K, Log F, Log O) LogReducer is 2×-3× as fast as LZMA. LogReducer is slower than gzip, as gzip is optimized for speed. We do not show the speed of gzip in Figure 5(b) since its high value will make other bars hard to distinguish.

To compress all 1.76TB logs, LogReducer takes 58.19 hours; Logzip takes nearly 27 days; LogArchive takes nearly 23 days; LZMA takes 91.54 hours; gzip takes 25.35 hours. In other words, LogReducer is 11.22×, 9.43×, and 1.57× as fast as Logzip, LogArchive, and LZMA respectively; it is about 60% slower than gzip.

6.2 Effects of Individual Techniques

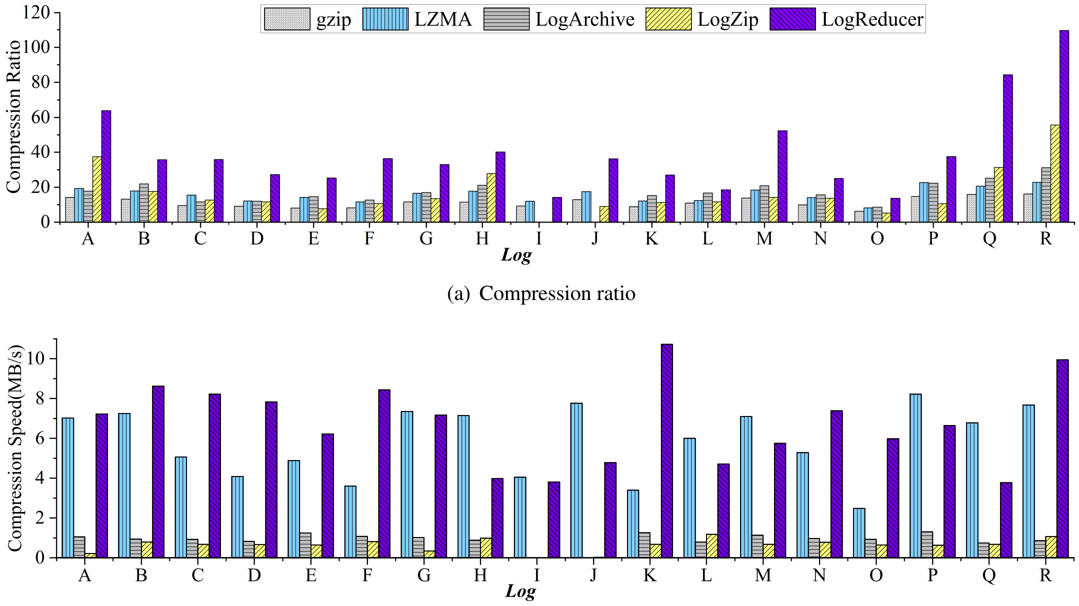
This section measures the effects of individual techniques presented in §3 and §4.

We use an efficient re-implementation of parser-based compressor as our baseline (LogReducer-NB), which includes the C/C++ implementation, removing the limit on the number of variables, and cutting the parser tree (§3). We add batch processing on LogReducer-NB to get LogReducer-B (§3), add delta timestamps on LogReducer-B to get LogReducer-D (§4.1), add elastic encoding approach on LogReducer-D to get LogReducer-ED (§4.3) and finally add numerical correlation utilization (§4.2) on LogReducer-ED to get the full version of LogReducer. The result is shown in Table 7.

As one can see, the efficient re-implementation (NB version) significantly improves the compression ratio and compression speed over Logzip in almost every type of logs. This has confirmed one of our key observations: an efficient implementation is critical to realize the full potential of parser-based log compression.

Log type	Log A	Log B	Log C	Log D	Log E	Log F	Log G	Log H	Log I
Total Size(GB)	18.67	16.05	45.82	65.74	34.98	443.30	148.88	0.19	14.20
Total Line(10^6)	74.74	72.60	231.43	406.98	77.56	1425.37	579.94	1.08	8.65
Time Span(H)	476	75	87	20	6	8	1563	32	8977
Log type	**Log J**	**Log K**	**Log L**	**Log M**	**Log N**	**Log O**	**Log P**	**Log Q**	**Log R**
Total Size(GB)	18.67	16.05	45.82	65.74	34.98	443.30	148.88	0.19	14.20
Total Line(10^6)	74.74	72.60	231.43	406.98	77.56	1425.37	579.94	1.08	8.65
Time Span(H)	85	3335	238	30	174	1512	62	165	722

Table 6: Log dataset description.



(a) Compression ratio

(b) Compression speed

Figure 5: Performance on AliCloud logs

The B version (batch processing) is over 1.5× as fast as the NB version on 10 logs. In particular, it is 1.99× and 1.82× as fast as the NB version on Log D and Log C. These two logs have many files and thus LogReducer can save much time by batch processing. Batch processing has no impact on compression ratio as it does not change the logic of the compression algorithm.

The compression ratio of the D version (delta timestamps) is over 1.1× as high as the B version on 3 logs and over 1.05× as high on 7 logs. In particular, its compression ratio is 1.24× as high as the B version on Log R. Delta timestamps can bring significant improvement to logs which have a large percentage of timestamp values: Log R, Log A and Log H have the highest, third highest and fourth highest timestamp percentage among all 18 logs (see Table 3) and thus can benefit from delta timestamps. Log B, which has the second highest timestamp percentage, is relatively sparser and does not benefit much from delta timestamps. Delta timestamps improves compression speed as well by feeding a smaller intermediate result to the general-purpose compressor: it is over 1.05× as fast as the B version on 6 logs.

The compression ratio of the ED version (elastic encoding) is over 1.05× as high as the D version on 12 logs. It mainly improves compression ratio on logs with a large percentage of small numbers, such as Log D, Log R, and Log M. The ED version is over 1.5× as fast as the D version on 5 logs and 1.2× as fast on 11 logs, since elastic encoding provides a dedicated and thus more efficient way to trim leading zeroes or ones compared with general-purpose methods.

The compression ratio of the LR version (correlation identification and utilization) is over 1.05× as high as the ED version on 4 logs. In particular, it is 2.07× as high as the ED version on Log F and 1.13× as high on Log O, because correlations are common in these two logs. It will incur overhead, its speed is 0.7× to 1.05× compared with ED version.

6.3 Performance on Public Logs

To examine the generality of LogReducer beyond AliCloud logs, we evaluate LogReducer on 16 types of public logs [18]

	Compression Ratio						Compression Speed (MB/s)						
	LZMA	**Logzip**	**B & NB**	**D**	**ED**	**LR**	**LZMA**	**Logzip**	**NB**	**B**	**D**	**ED**	**LR**
Log A	19.30	37.34	53.96	61.94	63.79	63.86	7.03	0.22	3.99	6.42	6.31	7.58	7.23
Log B	17.91	17.64	32.66	33.55	35.63	35.67	7.25	0.79	3.80	6.75	7.21	9.52	8.63
Log C	15.48	12.61	30.36	32.30	34.80	35.81	5.06	0.68	3.47	6.31	6.17	9.50	8.22
Log D	12.16	11.57	23.08	24.50	26.56	27.26	4.08	0.66	2.84	5.64	5.29	9.80	7.83
Log E	14.19	7.73	22.99	23.35	24.73	25.22	4.89	0.64	4.33	5.34	5.42	6.93	6.22
Log F	11.58	10.69	16.32	16.47	17.62	36.42	3.60	0.81	3.32	4.33	4.33	8.00	8.44
Log G	16.58	13.42	30.23	31.76	33.00	32.99	7.35	0.34	4.07	5.89	6.52	7.27	7.17
Log H	17.73	27.73	34.85	38.58	40.05	40.08	7.15	0.99	3.71	3.64	3.83	3.96	3.98
Log I	11.95	/	13.88	13.88	14.03	14.26	4.05	/	5.26	3.81	3.57	3.85	3.81
Log J	17.46	9.04	31.16	33.25	34.94	36.22	7.76	0.03	2.72	4.37	4.60	4.82	4.78
Log K	12.14	11.20	23.88	24.51	25.74	26.97	3.39	0.67	4.53	6.82	6.24	10.76	10.72
Log L	12.38	11.62	17.75	17.96	18.43	18.48	6.01	1.17	2.55	4.74	4.80	5.47	4.71
Log M	18.42	14.20	37.56	39.14	43.56	43.99	7.10	0.67	4.90	5.22	6.55	7.21	5.75
Log N	14.11	13.64	22.43	22.63	23.71	25.01	5.28	0.77	3.56	5.68	5.66	7.61	7.38
Log O	8.25	5.23	11.35	11.28	12.05	13.67	2.48	0.64	2.52	3.42	3.44	7.15	5.98
Log P	22.73	10.61	34.90	35.98	36.92	37.58	8.22	0.63	5.75	5.52	7.14	9.32	6.64
Log Q	20.55	31.27	76.72	79.05	83.09	84.25	6.78	0.68	2.41	3.67	3.72	3.76	3.77
Log R	22.82	55.63	80.73	100.44	109.21	109.51	7.67	1.07	4.94	8.23	7.85	10.87	9.95

Table 7: Effects of individual techniques on compression ratio and compression speed. X is short for LogReducer-X (X∈{B, NB, D, ED}). LR stands for the full version of LogReducer. "/": Logzip failed on Log I.

from diverse sources [25, 49]. As shown in Figure 6, the compression ratio of LogReducer is 1.03×–3.15× compared with Logzip, 1.19×–5.14× compared with LogArchive, 1.23×–5.30× compared with LZMA, and 1.79×–20.27× compared with gzip. We further investigate the logs on which LogReducer has less improvement: some of them have too many templates (e.g. Android, Thunderbird), which causes all parse-based methods, including LogReducer, to have many mismatches; some of them have only a few variables and even fewer numerical variables (e.g., Thunderbird, Proxifer), which causes LogReducer's optimizations to be less effective; in addition, LogZip has a specific optimization for HDFS log, which improves the compression ratio of LogZip.

In terms of compression speed, LogReducer is 2.05×–101.12× as fast as Logzip and 1.79×–9.95× as fast as LogArchive. LogReducer is slower than LZMA by up to 5.88× and than gzip by up to 36.16× due to two reasons. First, since over half of the logs are smaller than 100MB, the initialization overhead of LogReducer (e.g. space allocation) becomes significant, taking over 40% of the time. Second, some cases have too many templates (e.g. Android, Thunderbird), which causes a low matching rate and a waste of time.

Such results have confirmed the assumptions of LogReducer: LogReducer is mainly designed for large-scale logs with a small number of templates and many variables. When such assumptions hold, LogReducer can perform significantly better than existing methods; when such assumptions do not hold, LogReducer is less effective but can still achieve the highest compression ratio.

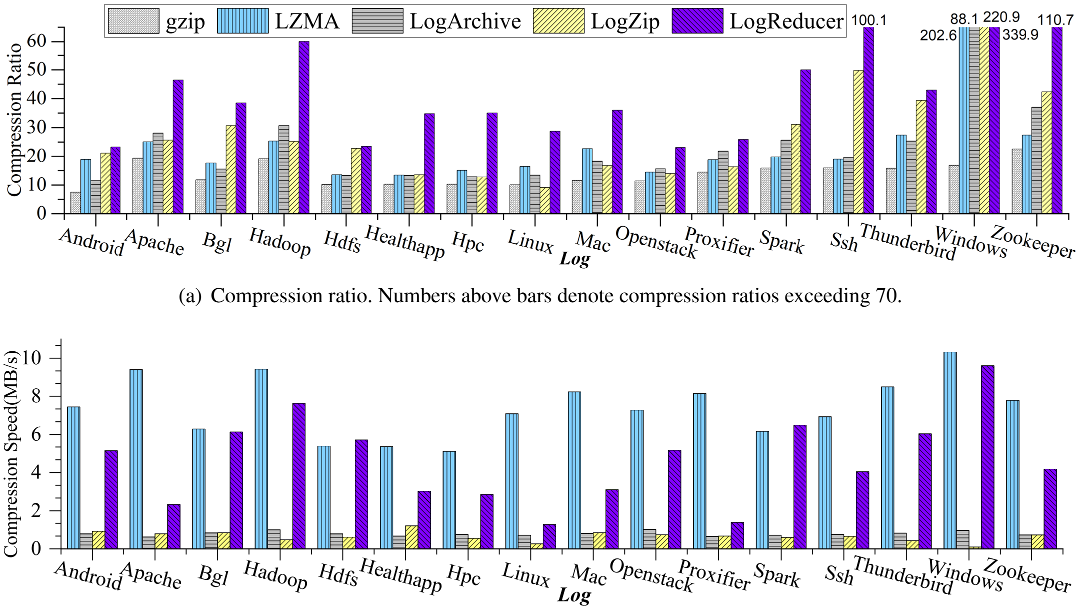
7 Related Work

Log parser. Log parser focuses on the extraction process of log templates, which can be divided into three types: cluster-based methods (LKE [14], LogSig [44], SHISO [34], LenMa [42], LogMine [20]), frequent-pattern-based methods (SLCT [46], LFA [35]), and heuristic-structure-based methods (IPLoM [32], AEL [27], Drain [23]).

Cluster-based methods divide the logs into clusters and extract templates for each cluster. Pattern-based methods try to extract frequent patterns from log entries and regard them as constant templates. Heuristic methods will extract log structure based on observations of log entries. Zhu et al. [49] compare these methods and find that Drain performs better than others. As a result, both Logzip and our implementation are based on Drain.

Number encoding methods. LevelDB [16] has used variant encoding to represent numbers based on their size. Thrift [43] has used Zigzag encoding to get more leading zero to enable efficient data serialization when communicating between processes. Compared with them, LogReducer further uses elastic encoding to reduce the space overhead of storing numerical variables.

General-purpose compression approaches. These methods can be categorized into three kinds: statistic-based, predict-based, and dictionary-based. Statistic-based compression methods (e.g., Huffman coding [40], Arithmetic coding [47]) first collect statistic information about input logs and then design variant length coding for each tokens. Predict-



(a) Compression ratio. Numbers above bars denote compression ratios exceeding 70.

(b) Compression speed

Figure 6: Performance on public dataset.

based compression methods (e.g., PPMd [4]) predict the next token based on current context during reading the input stream, and assigns a shorter encoding if prediction is successful. Dictionary-based compression methods (e.g., LZMA, gzip) search for similar tokens in a sliding window and store them in a dictionary when processing the input stream.

Statistic-based methods need to read the input log file twice. As a result, when the input log file is large, they are not efficient. With prediction-based methods, the appearance of variables will decrease the prediction accuracy. Dictionary-based methods may lose the chance to trim redundancy within a long distance, and do not take the delta of timestamps and correlation of variables into consideration, since they are not related to redundancy literally. Our methods utilize general-purpose compression approaches and improve their effectiveness on log data.

Log-specific compression approaches. These methods can be divided into two categories: parser-based and non parser-based. CLC [22], LogArchive [3], Cowic [29] and MLC [13] process variables and templates together. CLC tries to find the frequent patterns shown in log files and processes these patterns directly. LogArchive uses similarity function and sliding windows to divide log entries into different buckets and compresses buckets together to improve compression ratio. Cowic does not focus on the compression ratio. Instead, it tries to decrease the decompression latency by only decompressing needed logs rather than the whole files. MLC uses block-level duplication methods to find redundancy concealing between log entries and divide them into groups according to their similarities and compress them using delta encoding.

Logzip [30] extracts templates and processes templates and variables separately. It uses a parser to get several templates on a small sample and extracts all templates in original log files by iterative matching. Finally, it compresses template IDs and variables using general-purpose compression methods separately. However, Logzip does not perform well on our logs due to sub-optimal implementation.

8 Conclusion

This work examines the latest parser-based log compression approach on production logs. It observes that, first, an efficient implementation is critical to realize the full potential of this approach; and second, there are more opportunities to further compress logs. Based on these ideas, we have built LogReducer, which shows promising compression ratio and compression speed.

Acknowledgment

We thank all reviewers for their insightful comments, and especially our shepherd, Dalit Naor, for her guidance during our camera-ready preparation. This work was supported by the National key R&D Program of China under Grant 2018YFB0203902, and the National Natural Science Foundation of China under Grants 61672315 and 62025203.

References

[1] Boyuan Chen and Zhen Ming Jiang. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering*, pages 71–81. IEEE, 2017.

[2] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale Internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 217–231. USENIX Association, 2014.

[3] Robert Christensen and Feifei Li. Adaptive log compression for massive log data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1283–1284. ACM, 2013.

[4] John Cleary and Ian Witten. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications*, 32(4):396–402, 1984.

[5] Python date engineer group. Python data analysis library Pandas. https://pandas.pydata.org/, 2015.

[6] Peter Deutsch. DEFLATE compressed data format specification version 1.3. https://tools.ietf.org/html/rfc1951, 1996.

[7] GNU developer group. Homepage and documentation of Tar. https://www.gnu.org/software/tar/, 2019.

[8] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *Proceedings of the 16th International Conference on Data Mining*, pages 859–864. IEEE, 2016.

[9] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.

[10] Susan Dumais, Robin Jeffries, Daniel M Russell, Diane Tang, and Jaime Teevan. Understanding user behavior through log data and analysis. In *Ways of Knowing in HCI*, pages 349–372. Springer, 2014.

[11] Yaochung Fan, Yuchi Chen, Kuanchieh Tung, Kuochen Wu, and Arbee L P Chen. A framework for enabling user preference profiling through Wi-Fi logs. *IEEE Transactions on Knowledge and Data Engineering*, 28(3):592–603, 2016.

[12] Bettina Fazzinga, Sergio Flesca, Filippo Furfaro, and Luigi Pontieri. Online and offline classification of traces of event logs on the basis of security risks. *Journal of Intelligent Information Systems*, 50(1):195–230, 2018.

[13] Bo Feng, Chentao Wu, and Jie Li. MLC: an efficient multi-level log compression method for cloud backup systems. In *Proceedings of 2016 IEEE Trustcom/BigDataSE/ISPA*, pages 1358–1365. IEEE, 2016.

[14] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 9th IEEE international conference on data mining*, pages 149–158. IEEE, 2009.

[15] Mona Ghassemian, Philipp Hofmann, Christian Prehofer, Vasilis Friderikos, and Hamid Aghvami. Performance analysis of Internet gateway discovery protocols in ad hoc networks. In *Proceedings of 2004 IEEE Wireless Communications and Networking Conference*, volume 1, pages 120–125. IEEE, 2004.

[16] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb, 2011.

[17] LogArchive group. Open source code of LogArchive. https://github.com/robertchristensen/log_archive_v0, 2019.

[18] Loghub group. Download link of public log dataset. https://zenodo.org/record/1596245#.XMMZ1dv7S-Y, 2019.

[19] Logzip group. Open source code of Logzip. https://github.com/logpai/logzip, 2019.

[20] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. LogMine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1573–1582. ACM, 2016.

[21] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, 23(6):3248–3280, 2018.

[22] Kimmo Hätönen, Jean François Boulicaut, Mika Klemettinen, Markus Miettinen, and Cyrille Masson. Comprehensive log compression with frequent patterns. In *Proceedings of 2003 International Conference on Data Warehousing and Knowledge Discovery*, pages 360–370. Springer, 2003.

[23] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *Proceedings of 2017 IEEE International Conference on Web Services*, pages 33–40. IEEE, 2017.

[24] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *Proceedings of the 27th International Symposium on Software Reliability Engineering*, pages 207–218. IEEE, 2016.

[25] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. Loghub: A large collection of system log datasets towards automated log analytics. *arXiv preprint arXiv:2008.06448*, 2020.

[26] Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.

[27] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *Proceedings of the 8th International Conference on Quality Software*, pages 181–186. IEEE, 2008.

[28] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. The unified logging infrastructure for data analytics at Twitter. *Proceedings of the VLDB Endowment*, 5(12):1771–1780, 2012.

[29] Hao Lin, Jingyu Zhou, Bin Yao, Minyi Guo, and Jie Li. Cowic: A column-wise independent compression for log stream analysis. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 21–30. IEEE, 2015.

[30] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R Lyu. Logzip: extracting hidden structures via iterative clustering for log compression. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 863–873. IEEE, 2019.

[31] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11):1921–1936, 2011.

[32] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1255–1264, 2009.

[33] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on Program Comprehension*, pages 167–177. ACM, 2018.

[34] Masayoshi Mizutani. Incremental mining of system log format. In *Proceedings of 2013 IEEE International Conference on Services Computing*, pages 595–602. IEEE, 2013.

[35] Meiyappan Nagappan and Mladen A Vouk. Abstracting log lines to log event types for mining software system logs. In *Proceedings of the 7th Working Conference on Mining Software Repositories*, pages 114–117. IEEE, 2010.

[36] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 26–26. USENIX Association, 2012.

[37] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 45–56. IEEE, 2015.

[38] LogReducer research group. Open sample of large-scale cloud logs. `https://github.com/THUBear-wjy/openSample`, 2020.

[39] LogReducer research group. Open source code of LogReducer. `https://github.com/THUBear-wjy/LogReducer`, 2020.

[40] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[41] Julian Seward. The bzip2 home page. `http://www.bzip.org`, 1997.

[42] Keiichi Shima. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213*, 2016.

[43] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[44] Liang Tang, Tao Li, and Chang-Shing Perng. LogSig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 785–794. ACM, 2011.

[45] Sarah K Tyler and Jaime Teevan. Large scale query log analysis of re-finding. In *Proceedings of the 3rd ACM international conference on Web search and data mining*, pages 191–200, 2010.

[46] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management*, pages 119–126. IEEE, 2003.

[47] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[48] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural support for programming languages and operating systems*, pages 143–154. ACM, 2010.

[49] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 121–130. IEEE, 2019.

[50] 7 zip developer group. 7-zip file achiever home page. https://www.7-zip.org/, 2019.