LogShrink: Effective Log Compression by Leveraging Commonality and Variability of Log Data

Xiaoyun Li
Sun Yat-sen University
Guangzhou, China
lixy223@mail2.sysu.edu.cn

Hongyu Zhang*
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Van-Hoang Le
The University of Newcastle
NSW, Australia
vanhoang.le@uon.edu.au

Pengfei Chen
Sun Yat-sen University
Guangzhou, China
chenpf7@mail.sysu.edu.cn

ABSTRACT

Log data is a crucial resource for recording system events and states during system execution. However, as systems grow in scale, log data generation has become increasingly explosive, leading to an expensive overhead on log storage, such as several petabytes per day in production. To address this issue, log compression has become a crucial task in reducing disk storage while allowing for further log analysis. Unfortunately, existing general-purpose and log-specific compression methods have been limited in their ability to utilize log data characteristics. To overcome these limitations, we conduct an empirical study and obtain three major observations on the characteristics of log data that can facilitate the log compression task. Based on these observations, we propose LogShrink, a novel and effective log compression method by leveraging commonality and variability of log data. An analyzer based on longest common subsequence and entropy techniques is proposed to identify the latent commonality and variability in log messages. The key idea behind this is that the commonality and variability can be exploited to shrink log data with a shorter representation. Besides, a clustering-based sequence sampler is introduced to accelerate the commonality and variability analyzer. The extensive experimental results demonstrate that LogShrink can exceed baselines in compression ratio by 16% to 356% on average while preserving a reasonable compression speed.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Log Compression, Data Compression, Log Analysis, Clustering

*Hongyu Zhang is the corresponding author.

ICSE 2024, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3608129

ACM Reference Format:
Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. 2024. LogShrink: Effective Log Compression by Leveraging Commonality and Variability of Log Data. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3608129

1 INTRODUCTION

Recording system events and status during runtime execution is important for maintaining and troubleshooting software systems. The usage of log data spans a wide range of scopes, including software testing prior to system deployment [1, 2], real-time system performance monitoring [3, 4], and root-cause analysis [5, 6, 7, 8, 9]. As the scale of a system expands, the volume of log data can increase exponentially. Recent reports show that the volume of log data generated from software systems has grown significantly in recent years. A modern software system can produce log data at rates of 100 TB to several PB per day [10, 11]. In many scenarios like forensic analysis, log data is stored for a long period for backtracking and understanding security issues. For example, system logs must be stored for up to 180 days in AliCloud [11]. The cost of storing such a vast volume of log data can be prohibitively expensive. To illustrate, consider a cloud provider that needs to store 1 PB of log data per day, and the cost of logging storage is $0.50/GiB [12] per month. The monthly bill can reach up to $465.7k, posing a significant financial burden on storage costs for cloud providers.

To address the challenge of log storage, there are two possible solutions: reducing log generation and compressing log files. Some studies [13, 14, 15] propose to generate log messages on demand to reduce the volume of console logs. However, the amount of logs after the reduction process is still significant and often reaches petabyte-level outputs per day (e.g., reducing from 19.7 PB to 12.0 PB per day [14]). Therefore, log compression is essential to archive large volumes of log data and saves disk storage space while preserving the opportunities for further analysis. Achieving a high compression ratio is critical in practice, as it can result in significant savings in disk space costs.

The most straightforward way to perform log compression is to apply general-purpose data compression algorithms such as lzma [16], gzip [17] and bzip2 [18]. They can obtain a relatively small log file compared to the original log file. These algorithms

analyze log files character-by-character, identify the repeated characters in log data, and replace them with a shorter representation. However, they cannot fully disclose the redundancy of log files, which are well formatted and might enable more effective compression [19, 20]. To overcome this limitation, some log-specific compression methods [10, 21, 22] are proposed to utilize the latent structure of log data to improve compression results. For example, LogZip [20] extracts log events and corresponding parameters via iterative clustering and compresses log data using dictionary encoding. LogReducer [11] proposes to improve compression ratio through delta encoding of timestamps and elastic encoding of numerical values. Although effective, these methods are still limited in utilizing the characteristics of log data and their compression ratios can be further improved.

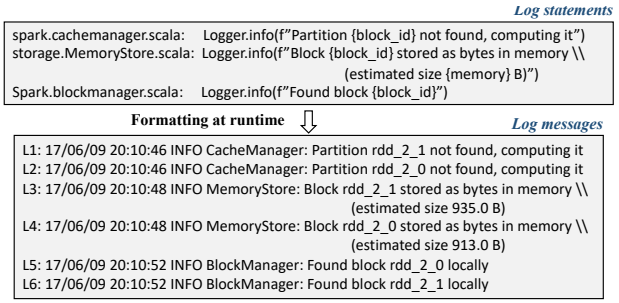
In order to compress log data by leveraging its inherent characteristics, we conduct an empirical study on three real-world log datasets from diverse software systems. We obtain three key observations that could potentially enhance both the compression ratio and speed. First, we observe that commonality and variability exist among log messages and they can be exploited to generate a shorter representation for log messages. Second, we find that the storage style has a significant impact on the compression ratio. Specifically, column-oriented storage style can reduce the compressed file size by 36% to 103% compared to traditional row-oriented storage style due to the columnar nature of log data. Lastly, we observe that the distribution of log sequence types is highly imbalanced. Over 50% of all log sequences belong to only 3%∼5% of log sequence types.

Based on the aforementioned three observations, we propose **LogShrink**, a novel and effective log-specific compression method by leveraging commonality and variability of log data. LogShrink first reads and segments the input log file into multiple log chunks to enable parallel batch processing. Each log chunk is initially parsed into structured logs including log headers, log events, and log variables. Next, we propose a novel clustering-based sequence sampling method to extract representative log sequences in each log chunk. Then, we devise an analyzer to identify commonality and variability in log data. The identified characteristics are sent to a compressor, which matches all log data corresponding to their types, encodes values according to their value types, and stores all the content in a column-oriented format. Finally, all the encoded files are compressed using a general-purpose compressor.

We conduct extensive experiments to evaluate LogShrink on 16 public datasets collected from a variety of software systems [23]. The experimental results show that LogShrink outperforms two log-specific compression methods and three general-purpose compression methods by 16% to 356% on average in terms of compression ratios with a reasonable compression speed. The ablation study on LogShrink confirms that the proposed commonality and variability analyzer and clustering-based sequence sampling contribute to the improvement of both compression ratio and speed.

In summary, the major contributions of this paper are as follows:

- We conduct an empirical study on three real-world log datasets and obtain three observations, which can facilitate the log compression task.
- Based on the obtained observations, we propose a novel and effective log compression method by leveraging commonality and variability of log data. Our proposed commonality and variability analyzer can largely improve the compression ratio. In the meantime, a clustering-based sequence sampler can accelerate the analyzing process thus improving the compression speed.
- We perform extensive experiments on 16 public log datasets, which confirm the efficacy of our proposed method. The source code of our tool and our experimental data are available at https://github.com/IntelligentDDS/LogShrink



LID	Datetime	Level	Component	Log Event	Variables
L1	17/06/09 20:10:46	INFO	CacheManager	Partition * not found, computing it	rdd_2_1
L2	17/06/09 20:10:46	INFO	CacheManager	Partition * not found, computing it	rdd_2_0
L3	17/06/09 20:10:48	INFO	MemoryStore	Block * stored as bytes in memory (estimated size * B)	rdd_2_1, 935.0
L4	17/06/09 20:10:48	INFO	MemoryStore	Block * stored as bytes in memory (estimated size * B)	rdd_2_0, 913.0
L5	17/06/09 20:10:52	INFO	BlockManager	Found block * locally	rdd_2_0
L6	17/06/09 20:10:52	INFO	BlockManager	Found block * locally	rdd_2_1

Figure 1: The generation and parsing of log data

2 BACKGROUND AND RELATED WORK

2.1 Background

Log data is essential for the maintenance and operation of software systems, which allows engineers to understand the system's behaviors and diagnose problems [24, 25, 26, 27, 28, 29, 30, 31]. Figure 1 illustrates the process of generating and parsing log messages. During the development phase, developers instrument log statements like "`logger.info("Partition {block_id} not found, computing it")`" in source code [32, 33]. When the log statements are executed at runtime, a logger is specified to format the log statements to log content like "`Partition rdd_2_1 not found, computing it`". Then the logger pads the log content to the log messages (e.g., "`17/06/09 20:10:46 INFO CacheManager: Partition rdd_2_1 not found, computing it`") based on a pre-defined pattern including datetime, log level, and component information [34, 4]. Log parsing [35] is a fundamental step in log analysis that performs a reversible analysis to convert formatted log messages into structured ones. We introduce the structures of log parsing results as three components: *headers* (e.g., datetime, log level, and component), *log events* (e.g., "`Partition * not found, computing it`") and *variables* (e.g., "`rdd_2_1`"). A log sequence is a series of log messages, representing a system execution flow.

In practice, large-scale systems generate vast amounts of log data. These data must be stored for extended periods for a variety of reasons, including (i) Security analysis to identify potential long-term

network attacks. For example, a recent attack remained undetected for eight months, and log access data is a critical data source to trace the attackers' behaviors [36, 37]; (ii) Audit compliance. Cloud providers are required by local laws to store logs for a specified period; (iii) Long-term statistical analysis. Log data is analyzed for prolonged periods to gain insights into system performance and user behavior. However, archiving such a massive amount of log data introduces expensive overhead on log storage costs.

Current logging frameworks such as ELK (ElasticSearch-LogStash-Kibana) stack [38] utilize general-purpose compressors (e.g., lzma [16], gzip [17], bzip2 [18]) to compress a large volume of log data. These compressors analyze log files character-by-character, identify repeated characters in log data, and substitute them with a shorter representation. For instance, LZ77 [39] algorithm used in gzip replaces the datetime "17/06/09 20:10:46" in L2 with a shorter representation of (76, 16, ' '), which comprises of an offset length, matching length, and the next character. However, general-purpose compressors do not consider the unique characteristics of log data. For example, the datetime here keeps increasing. Advanced log-specific compression methods leverage the latent structure of log data and apply various techniques to enhance the compression ratio of log data. For example, LogZip [20] extracts log events iteratively and replaces the log events with a shorter log event ID. LogReducer [11] also parses log events and analyzes relations only in numerical parameter values. LogBlock [22] targets applying some heuristics-based preprocessing steps on small log block compression. Although they achieve promising results on the log compression task, they either require some domain knowledge, such as manual datetime identification or apply many heuristics rules in preprocessing. Therefore, they still have limitations in practice.

2.2 Related Work

General-purpose data compression methods. Compression algorithms exploit statistical redundancy to eliminate redundancy literally, which can be categorized into three types: entropy-based, dictionary-based, and prediction-based. Entropy-based methods (e.g., Huffman encoding [40], Arithmetic encoding [41]) build a probability model and find the optimal minimized coding for data. Dictionary-based methods (e.g., gzip [17], lzma [16]) search the repeated tokens and replace them with dictionary references when processing the input stream. Prediction-based methods (e.g., PPMd [42], DeepZip [43]) use a set of previous tokens to predict the next token, and encode the prediction results adaptively. Yao et al. [44] conducted an empirical study on the performance of general-purpose compressors on log data. From the results, we can indicate that all general-purpose data compression methods can only analyze log messages character-by-character or bit-by-bit without reorganizing log data based on the characteristics of log data.

Log-specific data compression methods. Considering log data as semi-structured data, it is highly redundant by nature. Log-specific compression methods can be categorized into two types: non-parser-based and parser-based. Non-parser-based log compression methods (e.g., LogArchive [21], Cowic [10] and MLC [45]) process log data without extracting log event patterns. Parser-based log compression methods [20, 19, 22, 11] use a log parser to extract log events and process headers, events, and variables separately. According to the data type, they apply various encoding methods like delta encoding, common sub-pattern extraction, and dictionary encoding to shrink log data heuristically. In terms of data characteristics, most of them [20, 11, 22] deal with the high redundancy of log content. Others [11, 22, 19] leverage the timestamp feature. CLP [46] and LogGrep [47] deliver efficient query tools on compressed data. CLP parses log lines into schemas and stores the variables as dictionary and non-dictionary variables. LogGrep further extracts static patterns and runtime patterns of dictionary variables and packs them into a set of capsules in a fine-grained manner. However, they could not achieve a high compression ratio while preserving searchable features.

3 AN EMPIRICAL STUDY ON CHARACTERISTICS OF LOG DATA

In order to leverage the latent characteristics of log data to improve log compression, we conduct an empirical study on three real-world log datasets from a widely used log repository [23]. The basic statistics (i.e., file size, the number of log event types, and the number of log header fields) of the datasets are presented in Table 1. Through the empirical study, we obtain three observations:

Table 1: The statistics of datasets used in empirical study

Dataset	File size	# Event	# Header
Hadoop	48.61 MB	298	5
HPC	32.00 MB	104	6
OpenSSH	70.02 MB	62	5

Observation 1: Commonality and variability exist among pairs of log components. Considering a log sequence as a formatted execution flow, commonality and variability can be manifested in the following pairs.

- Header-Header (H-H): Each header is rendered by a predefined format, including timestamp, level and component. The timestamp exhibits a strong variability due to its increasing nature. The other meta information, such as log level and log component, shows a strong commonality because they usually come from a limited number of possible values.
- Event-Header (E-H): Among the padded information in log headers, there is some static information bound with events such as log level, logger name, and file location. For example, in Figure 1, the log component of the event "`Partition * not found, computing it`" is always `CacheManager`. The commonality can be identified as the binding among them.
- Event-Variable (E-V): The number and the types of variables are always the same among log messages with the same log event, which can be considered as commonality. For example, in Figure 1, the log messages L3 and L4 both have two variables. Their first variables are of the type of block ID (i.e., "`rdd_x_x`") and their second variables are of the type of floating point number.
- Variable-Variable (V-V): Variables within the same log event share a similar pattern with a slight difference. For example, in Figure 1, variables in L1 and L2 are "`rdd_2_1`" and

"rdd_2_0", which follow the pattern "rdd_x_x". Between them, the variability can be observed as the change of x, namely the change from 1 to 0.

Table 2: The statistics of occurrence and proportion of different pairs. # denotes their occurrence and % denotes their proportion.

Pairs	Hadoop		HPC		OpenSSH	
	#	%	#	%	#	%
H-H	5	100%	6	100%	5	100%
E-H	146	48%	39	40%	20	29%

Two out of four pairs (i.e., H-H, E-H) can be explicitly defined in the empirical study. For H-H pairs, if the header only includes a finite set possible set or an increasing timestamp, then it can be considered as a satisfied pair. We identify E-H pairs as satisfying the condition if the log headers with the same log event only have one value. We first calculate the total number of H-H and E-H pairs in three datasets accordingly. Then we carefully identify the fields that meet the requirements shown above in H-H and E-H pairs. For example, we count the 304 E-H possible pairs in total in Hadoop and identify 146 satisfied E-H pairs. The proportion of E-H pairs is calculated as 48% (146/304). The statistics of the above pairs occurrence and corresponding proportion results among three datasets are shown in Table 2. We can see from the results that the satisfied H-H pairs occupy 100% among all headers. Satisfied E-H pairs also widely exist in log messages, which take around 29% - 48% of all E-H pairs. The identified commonality and variability in these pairs can be leveraged to condense log data through dictionary replacement or differential values.

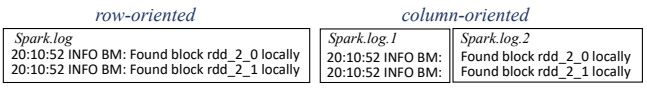


Figure 2: An example of different storage styles

Observation 2: Storage style significantly affects the compression ratio. When reviewing the existing log-specific compression methods, we notice that they rarely consider the file storage style and explain the impact of file storage styles. On the one hand, it is natural to store logs in a row-oriented storage style as log messages are printed line-by-line with logging frameworks [48]. On the other hand, log files are well formatted as each log line is semi-structured text [19]. Furthermore, there is evidence showing that general-purpose data compression methods perform better with structured or semi-structured texts [49, 50] as much redundant information can appear in the structure. Therefore, we argue that different storage styles can affect the performance of log compression. We depict these two storage styles, i.e., column-oriented and row-oriented storage styles, in Figure 2. The columns here are considered as fields of log messages, namely, log headers (e.g., "20:10:52 INFO BM:") and log content (e.g., "Found block rdd_2_0 locally") in Figure 2. Row-oriented storage style is to store columns of each log message in one row. Column-oriented storage style follows another way to store them column-by-column. For example, log headers are first stored in file 1 and then log content in file 2.

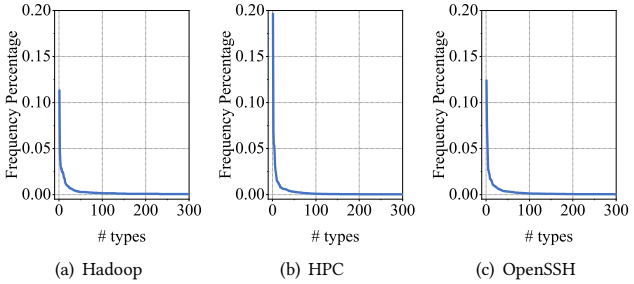
We study the impact of different storage styles on log compression by analyzing three real-world log datasets. The original log files are stored in the row-oriented storage style. Next, we transform logs to the column-oriented style by splitting each log line into its header and its content without further log parsing. We then archive files in different storage styles using two widely-used general-purpose compression methods (i.e., lzma [16] and gzip [17]). We measure the size of the archived files and compute the improvement as $\Delta = \frac{row\ file\ size - column\ file\ size}{column\ file\ size}$. Table 3 shows the results.

Table 3: Archived file size using different storage styles

Dataset	lzma		gzip		Avg Δ
	Row	Column (Δ)	Row	Column (Δ)	
Hadoop	1.23 MB	0.86 MB (0.43↓)	2.20 MB	1.70 MB (0.29↓)	0.36↓
HPC	2.02 MB	1.17 MB (0.72↓)	3.09 MB	2.43 MB (0.27↓)	0.50↓
OpenSSH	3.92 MB	1.73 MB (1.26↓)	4.19 MB	2.34 MB (0.79↓)	1.03↓

The results in Table 3 demonstrate that utilizing a column-oriented storage style for log compression significantly reduces the compressed file sizes, compared to the row-oriented storage style. On average, the column-oriented storage style achieves a storage reduction of 36% to 103% when using different general-purpose compression methods. The main reason is that log data exhibits more common patterns with the column-oriented storage style than with the row-oriented style. This finding suggests that storing logs in a column-oriented manner could enable more effective compression.

Observation 3: The distribution of log sequences is highly imbalanced. A log sequence reflects an execution flow of a program [51]. For example, in Figure 1, three log events (i.e., "Partition * not found, computing it", "Block * stored as bytes in memory (estimated size * B)", and "Found block * locally") form a type of log sequence. Analyzing the commonality and variability among all log sequences is time-consuming, thereby we study the distribution of log sequences to see if there are more efficient ways. Prior work [5] grouped log sequences by task ID and showed the long tail distribution of log sequence types. However, this method is not always feasible when log messages have no identifiers. Instead of using identifiers, we follow recent studies [52, 53]



(a) Hadoop (b) HPC (c) OpenSSH

Figure 3: Imbalanced distribution of log sequence types

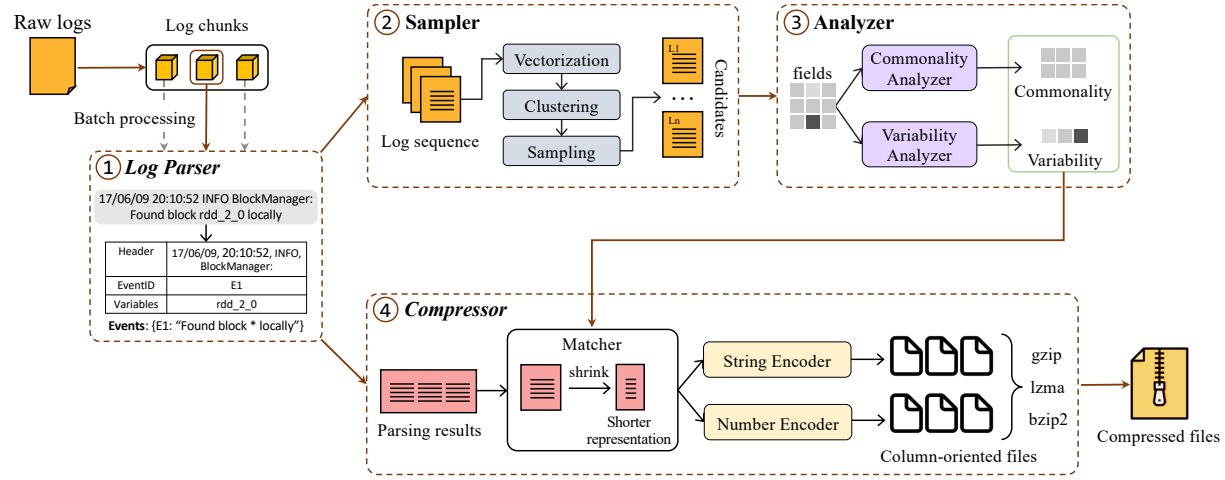


Figure 4: The overview of LogShrink framework

to group log sequences using a fixed-length window of size $h = 50$, and consider two log sequences belonging to the same type if their similarity score is greater than a certain threshold (0.6 by default). The similarity score is calculated as the number of common tokens divided by the number of total tokens in two sequences. We then analyze the frequency percentage of each log sequence type in the three datasets and present the results in Figure 3, where the x-axis denotes the distinct log sequence types, and the y-axis represents the frequency proportion of each log sequence type in the dataset. We can observe that the types of log sequences in the three datasets exhibit a highly imbalanced distribution, which is consistent across all three datasets. Out of hundreds of log sequence types, more than 50% of the log sequences come from the first 3%~5% log sequence types. The results suggest that we can identify commonality and variability more efficiently by analyzing a small sample of log sequences.

4 LOGSHRINK: THE PROPOSED APPROACH

Drawing upon the above observations, we propose LogShrink, a novel log compression method that can exploit the latent characteristics of log data to enable effective compression. The overview of LogShrink framework is illustrated in Figure 4. Since the raw log files that require compression are usually too large for processing, the raw log files are initially segmented into multiple log chunks with equal size. Log chunks are processed in batches, and each log chunk goes through four main components: ① *Log Parser*, ② *Sampler*, ③ *Analyzer*, and ④ *Compressor*.

The main essence behind LogShrink is that we try to represent log sequences in shorter forms based on their commonality and variability. Therefore, in LogShrink, firstly, *Log Parser* partitions log messages in each log chunk into three log components, namely log headers, log events, and log variables. To exploit the commonality and variability observed in Observation 1, we propose a novel and effective *analyzer* to identify the commonality and variability among log components. As it usually requires much time to analyze all log messages, a clustering-based sequence *sampler* is introduced to accelerate the *analyzer*. Based on Observation 3, the *sampler* is designed to sample a small yet representative set of log sequences and then the *analyzer* is applied to identify commonality and variability among sampled log sequences. The *compressor* takes all the parsed results from *log parser* and mined relations from *analyzer* as input. It shrinks log data by replacing the log data with a shorter representation based on the analyzed commonality and variability. Then *encoders* are applied according to value types (e.g., string values and numerical values). Inspired by Observation 2, all the encoded data are stored in a column-oriented storage format and eventually compressed to log files by a general-purpose compressor such as lzma [16] and gzip [17].

4.1 Log Parser

Log parsing is an essential step to convert unstructured log messages into structured ones. The procedure entails separating the log header and log content, followed by the identification of common and variable parts in the log content as log events and log variables, respectively. In recent years, many log parsers such as Spell [54], LogMine [55], Drain [56, 57], and LogPPT [35] are proposed to achieve satisfactory effectiveness. However, these log parsers have a time complexity of approximately $O(n)$, rendering them impractical for log compression tasks, particularly for large log files. In this paper, we adopt a sub-optimal yet practical log parser proposed by LogReducer [11]. It contains two steps: training and matching. Following this method, in the training step, the parser samples log segments and automatically generates header formats. Subsequently, it tokenizes the log segments and iteratively clusters them based on log level, component name, and frequently occurring words. After that, it builds a prefix tree to facilitate event matching where the first layer is the length of events, and the rest layer is tokens of events. In the matching step, the unsampled log messages utilize the built parser tree to search for the most similar event. If unmatched, it collects the raw unmatched log messages in an individual file. The log parser yields three log components, specifically header, events, and variables. Following Observation 2 in Section 3, we process all log components into column-oriented fields. Log headers are separated into multiple fields using space delimiters.

We group log variables by the log event IDs, thereby the variable list with the same log event constructs a matrix. Then we could process and store all the log components in a column-oriented way.

4.2 Clustering-based Log Sequence Sampler

Based on Observation 3, the distribution of log sequence types is highly imbalanced, which brings a big challenge to sampling. To overcome this issue, we devise a clustering-based log sequence sampling to extract the representative log sequences. In this method, log messages are parsed into windows with a fixed length, denoted as h, to form a log sequence. The process of clustering-based log sequence sampling is depicted in Figure 5.

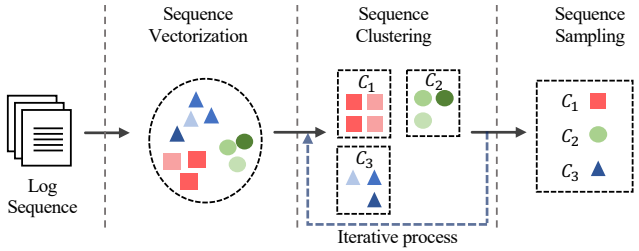


Figure 5: Clustering-based log sequence sampling

4.2.1 Sequence Vectorization. We calculate vector representations for each log sequence. To determine the importance of each log event in log sequences, we adopt a widely used technique Inverse Document Frequency (IDF) in text mining. In this technique, log events that occur frequently across log sequences are assigned lower weights, while those that occur less frequently are assigned higher weights. Specifically, the IDF weight is defined as $w_{idf}(e) = \log\left(\frac{N}{n_e}\right)$, where N is the total number of log sequences and n_e is the number of log sequences that contain log event e. We then construct sequence vectors using both log event frequency vectors V_{fre} and log event weight vectors as Equation 1.

$$V_i = V_{fre} * [w_{idf}(e_1), ..., w_{idf}(e_n)] \quad (1)$$

4.2.2 Sequence Clustering. We use an iterative clustering method to efficiently cluster log sequences. The process involves three steps: sampling, clustering, and matching. The input is a set of log sequences' vectors, and the output is their corresponding log sequence types. Specifically, given a set of N log sequence vectors and a sample rate ξ (default as 0.01), we randomly select a subset of $M = min(\xi * N, k_{min})$ sequence vectors as the input for the i-th iteration. Here, k_{min} is set to the minimum size that ensures the sampled data contains at least two samples of a single log sequence type. We calculate the value of k_{min} as $k_{min} = \underset{k}{\text{argmin}}\,(C_k^2 p^{(k-2)}(1-p)^2 \geq 1 - e^{-6})$. We then calculate the distance between each pair of sampled sequence vectors using Euclidean metrics (defined as $d(u,v) = \sqrt{\|u - v\|}$) and use a *Hierarchical Agglomerative Clustering (HAC)* to cluster the sequences. *HAC* seeks to build a hierarchy of clusters in a bottom-up way. It performs a linkage between two clusters if their distance is smaller than a threshold θ. The resulting clusters yield k sequence centers and the cluster IDs of all log sequences.

4.2.3 Sequence Sampling. Sequence sampling takes the candidate number M as input and output $max(k, M)$ sampled log sequences. Given sampled data size M, we first calculate the sampled data size $m_i = \lceil M/k \rceil$ for each log sequence cluster. Then we randomly sample a data size of m_i from each log sequence cluster and concatenate them. Finally, we obtain $max(k, M)$ log sequences. The impact under different window lengths, candidates M and threshold θ are evaluated in Section 5.

4.3 Commonality and Variability Analyzer

Observation 1 in Section 3 indicates that *commonality* and *variability* widely exist in pairs of log components. They have the potential to generate a more concise representation of log data. Specifically, we define commonality as the common part among the log components and variability as the relatively steady change among the log components. For example, for a sequence $A1$ = {`task#0-1`, `task#1-2`, `task#2-3`}, the commonality is manifested as a common string pattern "`task#x-x`", where x denotes the variable parts among the sequence. The variability in $A1$ is also obvious to observe. The task id is increasing by a step of 1 (i.e., from 0 to 1, and to 2). By exploiting such nature, we can present the corresponding log components by replacing the common part with a shorter index (e.g., {'`task#x-x`' → `0`}) and the variable part with a shorter difference (e.g., 1), respectively.

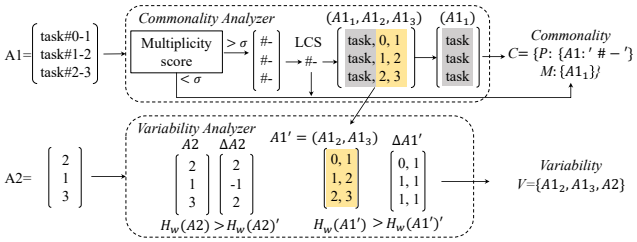


Figure 6: The process of analyzer

Given a series of representative log sequences obtained from sequence sampler in Section 4.2, Analyzer aims to analyze the latent commonality C and variability V in them. Figure 6 demonstrates the process of Analyzer. In order to identify commonality in sequence, a multiplicity score is employed to measure whether a field contains multiple identical values. However, in cases such as $A1$, three values are distinct but share some common values which are separated by delimiters. To address this, we introduce a delimiter-level Longest Common Subsequence (LCS) to extract the common delimiters in the sequence (e.g., '#-' in $A1$). This cannot be done by log parsers because they treat the entity with many delimiters inside (e.g., `rdd_0_1`, `/10.10.8.8`) as a single field. Consequently, $A1$ is divided into three sub-fields $A1_1, A1_2, A1_3$ using the mined delimiters. A fine-grained analysis is performed on these sub-fields. In the meantime, an entropy-based variability analyzer is introduced to see if the steps in the sequence follow a steady way. Given two input cases $A1$ and $A2$, the commonality C of $A1$ is eventually concluded as the combination of the common delimiters and the field index as shown in Figure 6. The variability V involves the indices of those satisfied fields. Since the commonality we observed in log data mostly exists in string values in Section 3, we analyze

the commonality in fields containing string values. Similarly, we analyze the variability in fields containing numerical values.

4.3.1 Commonality Analyzer. Each field in log components usually shares the commonality in string pattern. We first use a multiplicity function to determine whether the field satisfies the multiplicity constraint. The multiplicity function is shown in Eq. 2, where the multiset X refers to the possible values of one field. We calculate the multiplicity by dividing the count of the unique set in X by the count of the multiset X. For example, in Figure 6, given a multiset $A1$, the multiplicity score $M(A1) = \frac{3}{3} = 1$. If the multiplicity score of one field is smaller than a preset threshold σ, we consider that the field is highly redundant so we can replace it with a shorter representation. Otherwise, LogShrink activates the LCS-based pattern miner to further analyze whether there is a fine-grained commonality in this field.

$$M(X) = \frac{|\{x|x \in X\}|}{|X|} \tag{2}$$

LCS is to find the longest common subsequence among a sequence set. Compared with token-level LCS used in other log parsers, we apply a delimiter-level LCS to mine the shared delimiters among the field. Suppose Σ is a universe of delimiters. Given any sequence $\alpha = \{a_1, a_2, ..., a_m\}$, we extract the delimiter sequence $d = \{d_1, d_2, ..., d_m\}$, such that $d_i \in \Sigma$. Then a delimiter subsequence of d is defined as $\{d_i, d_{i+1}, ..., d_j\}$, where $i \in \mathbf{Z}^+$ and $1 \leq i \leq j \leq m$. A common subsequence is a subsequence of both sequences d_i and d_j. In the case of log field $A1$, the delimiter sequences are extracted as $d = \{'\#-','\#-','\#-'\}$. The delimiter-level LCS of d is '#-'.

Then, we separate each field by the mined common delimiters. The input example $A1$ is divided into three fields. We then calculate the multiplicity score of the first field and analyze the variability of the other two fields (i.e., numerical fields). Finally, the commonality of $A1$ is $C = \{P : \{A1 :' \#-'\}, M : \{A1_1\}\}$, including the delimiter-level LCS and the field indices that satisfy the multiplicity constraint.

4.3.2 Variability Analyzer. The main characteristic of variability is that the data reveals a relatively stable differential. Given a sequence $\alpha = \{a_1, a_2, ..., a_m\}$, the difference of α is defined as $\Delta\alpha = \{a_1, a_2 - a_1, ..., a_m - a_{m-1}\}$. Entropy is a widely used measurement in information theory to measure disorder, randomness, and uncertainty. In this paper, we use entropy to measure the degree of variability. A higher entropy represents that the sequence contains more variability to store. The definition of entropy is given as follows:

$$H(X) = \sum_{x \in X} P_X(x) \log_2 \frac{1}{P_X(x)} \tag{3}$$

where, in the context of a field, we use $P(x)$ to represent the probability of observing the value x and use X to represent all of the possible values in this field. However, we also need to consider the output bits of these fields. Intuitively, small values take fewer bits of storage [11, 34]. For example, a value of 3 only needs to take 2 bits to store. We use a metric $w(x) = \log_2(x + 1)$ to measure the minimum required bits of storage. Then we incorporate the minimum required bits of storage with the entropy as a weighted entropy metric:

$$\begin{aligned} H_w(X) &= \sum_{x \in X} w(x) P_X(x) \log_2 \frac{1}{P_X(x)} \\ &= -\sum_{x \in X} \log_2(x+1) P_X(x) \log_2 P_X(x) \end{aligned} \tag{4}$$

We calculate $H_w(X)$ of the original sequence α and the difference of sequence $\Delta\alpha$ as $H_w(\alpha)$ and $H_w(\Delta\alpha)$. If $H_w(\Delta\alpha)$ is smaller than $H_w(\alpha)$, the sequence α is considered to satisfy the variability, and vice versa. For example, in Figure 6, considering a sequence $A2 = \{2, 1, 3\}$ and its difference $\Delta A2 = \{2, -1, 2\}$, the weighted entropy of them are 1.68 and 0.79, respectively. $A2$ is thus considered to satisfy the variability requirement. In summary, after analyzing the variability of given $A2$ and the two sub-fields $A1_2$, $A1_3$ in $A1$, the results of V are that $V = \{A1_2, A1_3, A2\}$.

4.4 Compressor

The compressor takes all the parsing results from the log parser and identifies commonality and variability from the analyzer as input. It matches them to all log data and shrinks with a shorter representation. Then it encodes all values using an encoder according to its value types. All the encoded data are stored in multiple files in a column-oriented storage style. Finally, we use a general-purpose compressor as the zip tool to further improve the compression ratio.

4.4.1 Matcher. After obtaining the identified commonality and variability from representative log sequences, we need to apply them to all log data. LogShrink shrinks log data by replacing the field's values with shorter representations according to their characteristic types. For those fields whose commonality includes delimiter-level common patterns, we first separate them using the corresponding P. Then, for those fields in the commonality set satisfying the multiplicity constraint M, we build a dictionary for repetitive tokens and replace these tokens with a dictionary index. The built dictionary and replaced data are stored as two files. As for variability, we perform the differencing operation between consecutive values for those fields in the variability set. For all the analyzed characteristics, if one of them cannot be satisfied in all log data, we will drop them and store the rest.

4.4.2 Encoder. We categorize objects in log data as two data types: string values and numerical values. For string values, LogShrink outputs their raw values. As the numerical values are fairly small, they only use a few bits specified by their types. For example, a 4-byte integer value of 13 only requires 4 bits to store losslessly. Following the method used in LogReducer [11], we adopt an elastic encoder to encode numerical values.

4.4.3 Zip tool. Based on Observation 2 in Section 3, the processed data are stored in a column manner. For example, each header (e.g., timestamp, log component) and each variable are stored in separate files to improve the compression ratio. The final compression is done by a general-purpose compressor such as lzma [16], gzip [17], and bzip2 [18].

4.5 Decompressor

The decompression process is a reverse process of compression. At first, the decompressor applies the general-purpose decompressor

(e.g., lzma [16], gzip [17], bzip2 [18]) to decompress the whole compressed file into many uncompressed files. Next, LogShrink loads the files that record the identified commonality and variability sets. Then, LogShrink performs the recovery process of differencing operations on fields and the dictionary mapping operation accordingly. After that, we obtain three log components which are generated by the log parser. To recover the message content, we parameterize the asterisk in log events using variables in order. Finally, the header and message content can be joined together with the extracted delimiters to obtain the original log messages.

5 EVALUATION

We conduct extensive experiments on a variety of log datasets. Our evaluation aims to answer three questions:

- RQ1: What is the overall performance of LogShrink?
- RQ2: What is the effect of each individual component in LogShrink?
- RQ3: What is the impact of different settings?

5.1 Experimental Design

5.1.1 Datasets. We use 16 representative log datasets [23] from a wide range of systems to evaluate LogShrink, including distributed systems (e.g., HDFS, Hadoop, Spark, Zookeeper, OpenStack), supercomputers (e.g., BGL, HPC, Thunderbird), operating systems (e.g., Windows, Linux, Mac), mobile systems (e.g., Android, HealthApp), server applications (e.g., Apache, OpenSSH), and standalone softwares (e.g., Proxifier). All these logs amount to over 77 GB in total. The details of experiment datasets are presented in Table 4.

Table 4: The statistics of experimental datasets

System Type	Dataset	File Size	# Lines
Distributed systems	HDFS	1.47 GB	11,175,629
	Hadoop	48.61 MB	394,308
	Spark	2.75 GB	33,236,604
	Zookeeper	9.95 MB	74,380
	OpenStack	58.61 MB	207,820
Supercomputers	BGL	708.76 MB	4,747,963
	HPC	32.00 MB	433,489
	Thunderbird	29.60 GB	211,212,192
Operating systems	Windows	26.09 GB	114,608,388
	Linux	2.25 MB	25,567
	Mac	16.09 MB	117,283
Mobile systems	Android	183.37 MB	1,555,005
	HealthApp	22.44 MB	253,395
Server applications	Apache	4.90 MB	56,481
	OpenSSH	70.02 MB	655,146
Standalone software	Proxifier	2.42 MB	21,329

5.1.2 Evaluation Metrics. To measure the performance of LogShrink in log compression, we use the compression ratio and compression speed, which are widely used in the evaluation of compression methods [20, 11, 22]. The definitions are given as follows:

$$\text{Compression Ratio } (CR) = \frac{Original\ File\ Size}{Compressed\ File\ Size} \tag{5}$$

$$\text{Compression Speed } (CS) = \frac{Original\ File\ Size}{Cost\ Time} \tag{6}$$

5.1.3 Baselines. We compare our proposed method with two state-of-the-art log compression methods (e.g., LogZip [20] and LogReducer [11]) and three representative general-purpose compression methods (e.g., gzip [17, 58], lzma [16], and bzip2 [18]). gzip [17] is a traditional compression method based on DEFLATE algorithm, which achieves a good compression speed instead of a good compression ratio. Compared to gzip, bzip2 [18] is based on the Burrows-Wheeler transform algorithm. It has a better compression ratio yet worse compression speed. Lzma [16], which uses dictionary compression algorithms, usually gets a better compression ratio but has a relatively slow compression speed. In this study, we use a python package *tarfile* [59] to compress data with gzip and bzip2. The compression level opts to the highest 9 to achieve the best compression ratio. For lzma, we use the standalone *7za* tool [16] in the Linux system to compress data. In terms of state-of-the-art log-specific compression methods, LogZip extracts hidden structures for all log messages and performs different levels of compression. We select the level 3 to achieve the highest compression ratio. LogReducer is also a log parser-based compressor, which can further compress numerical values. We use their open-sourced code [60, 61] in our experiments.

5.1.4 Implementation and Environment. We implement LogShrink in Python 3.8. The raw log files are segmented with an equal size of 100k lines. We adopt the log parser implemented by the LogReducer [11] and modify it to adapt to our framework. As for the setting of parameters, we pre-define the universe of delimiters $\Sigma = \{-\#\ ><\ \_\ :;,\ []\backslash/.()\}$ used in the commonality analyzer. We set threshold $\theta = 4$, fixed-window length $h = 20$, and $M = 16$ used in sequence sampler as defaults. The threshold σ used in the commonality analyzer is set to 0.3 by default. Since the sampling in LogReducer [11] and LogShrink might yield random results during execution, we run them 10 times for all experiments and obtain the average results. We conduct our experiments on a Linux server equipped with 8× Intel Xeon 2.2GHz CPUs (with 32 cores in total) and 128GB RAM, and Red Hat 8.1 with Linux kernel 4.18.0.

5.2 RQ1: The Overall Performance of LogShrink

In this RQ, we compare LogShrink with state-of-the-art tools for log compression, including three general-purpose compressors (i.e., gzip [17], lzma [16], and bzip2 [18]) and two log-specific compressors (i.e., LogZip [20] and LogReducer [11]).

5.2.1 Effectiveness: Firstly, we compare the results of LogShrink with baselines in terms of Compression Ratio (CR). For a fair comparison, we use lzma from the 7z packet [16] as the zip tool for both LogZip and LogReducer. Table 5 shows the results.

From the results, we can see that LogShrink outperforms existing methods or achieves comparable results on almost all datasets (14 out of 16 datasets) in terms of CR. Specifically, LogShrink exceeds all general-purpose compressors. It can achieve a CR of 4.57× on

Table 5: Comparison in terms of Compression Ratio

Dataset	gzip	lzma	bzip2	LogZip	LogReducer	LogShrink
Android	7.742	18.857	12.787	**25.165**	20.776	21.857
Apache	21.308	25.186	29.557	30.375	43.028	**55.940**
BGL	12.927	17.637	15.461	32.655	38.600	**42.385**
Hadoop	20.485	36.095	32.598	35.008	52.830	**60.091**
HDFS	10.636	13.559	14.059	16.666	22.634	**27.319**
HealthApp	10.957	13.431	13.843	22.632	31.694	**39.072**
HPC	11.263	15.076	12.756	27.208	32.070	**35.878**
Linux	11.232	16.677	14.695	23.368	25.213	**29.252**
Mac	11.733	22.159	18.074	26.306	35.251	**39.860**
OpenSSH	16.828	18.918	22.865	42.606	86.699	**103.175**
OpenStack	12.158	14.437	15.231	17.258	16.701	**22.157**
Proxifier	15.716	18.982	23.619	21.493	25.501	**27.029**
Spark	17.825	19.908	26.497	20.825	57.135	**59.739**
Thunderbird	16.462	27.309	25.428	—	**49.185**	48.434
Windows	17.798	202.568	67.533	310.596	342.975	**456.301**
Zookeeper	25.979	27.667	36.156	47.373	94.562	**116.981**

Note: "—" denotes timeout. LogZip cannot parse and compress Thunderbird log within 1 week.

average and 25.64× at best over gzip, a traditional compression algorithm. Besides, LogShrink achieves 1.16× to 5.54× CR compared to lzma, and 1.14× to 6.76× compared to bzip2. In the comparison with two log-specific compression methods, LogShrink significantly outperforms LogZip by achieving better CRs on 15 out of 16 datasets. It achieves a CR of 1.66× on average and 2.87× at best (on Spark) over LogZip. Moreover, LogShrink equips higher CR on 15 out of 16 datasets compared to the most powerful log compressor, LogReducer. In particular, it exceeds LogReducer by 4.56% (Spark) to 33.04% (Windows). On the Thunderbird dataset, LogShrink also performs comparably by achieving a high CR of 98.47% over LogReducer. It is worth noting that on large-scale datasets (i.e., BGL, HDFS, Spark, and Windows) except Thunderbird, LogShrink performs the best compared to other log-specific compression methods. Its CR is 1.05× to 2.87× that of LogZip and LogReducer. Note that, in our experiments, LogZip failed to parse and compress Thunderbird data within 1 week. The reason why LogShrink performs worse on Android dataset than LogZip is that we adopt a sub-optimal yet practical log parser in our work but Logzip adopts an optimal yet slower log parser. The number of templates in Android is up to 76,923, making it ineffective in extracting log events from a limited number of samples.

5.2.2 Efficiency: Our LogShrink explicitly aims at compressing log files with high compression ratio in a reasonable running time. Therefore, we next analyze and compare LogShrink with log-specific compression methods in terms of Compression Speed. Table 6 shows the results.

We can see that LogShrink can compress log files with reasonable efficiency. It can achieve a compression speed of 2.95 MB/s on average, ranging from 1.31 to 5.51 MB/s. Compared to LogZip, which is also a Python-based compressor, LogShrink outperforms it significantly in efficiency. Specifically, LogShrink is 1.83×-273.79×

Table 6: Comparison in terms of Compression Speed (MB/s)

Dataset	LogZip	LogReducer	LogShrink
Android	0.068	8.918	5.347
Apache	0.737	1.686	1.880
BGL	0.874	18.189	2.519
Hadoop	0.901	4.919	3.137
HDFS	0.701	20.570	3.253
HealthApp	0.736	4.108	2.064
HPC	0.644	5.110	2.485
Linux	0.687	0.526	1.307
Mac	0.009	2.887	2.572
OpenSSH	0.715	13.268	3.409
OpenStack	0.537	6.389	2.945
Proxifier	0.716	0.929	1.315
Spark	0.550	18.871	2.821
Thunderbird	—	19.532	4.069
Windows	1.357	31.938	5.507
Zookeeper	0.842	3.071	2.523

(26.6× on average) as fast as LogZip. LogShrink is generally slower than LogReducer, as LogReducer is written in C++ and thus is more optimized in terms of execution speed than LogShrink (written in Python).

5.3 RQ2: Ablation Study

Table 7: Ablation study results

Dataset	LogShrink		w/o Sampler		w/o Analyzer	
	CR	CS	CR	CS	CR	CS
Android	21.857	5.347	21.784	4.354	20.804	7.048
Apache	55.940	1.880	56.075	1.565	42.803	2.286
BGL	42.385	2.519	43.489	1.767	32.965	5.283
Hadoop	60.091	3.137	61.206	2.904	57.885	7.281
HDFS	27.319	3.253	31.270	2.737	22.023	7.631
HealthApp	39.072	2.064	39.726	1.569	28.015	4.276
HPC	35.878	2.485	36.706	1.894	27.317	4.771
Linux	29.252	1.307	29.365	1.151	25.310	1.253
Mac	39.860	2.572	39.333	2.258	35.183	3.878
OpenSSH	103.175	3.409	101.727	2.675	71.874	5.382
OpenStack	22.157	2.945	22.648	2.136	20.573	5.484
Proxifier	27.029	1.315	28.061	1.119	26.262	1.265
Spark	59.739	2.821	59.234	2.227	49.147	5.651
Thunderbird	48.434	4.069	45.367	3.289	40.130	6.047
Windows	456.301	5.507	501.502	4.316	390.574	11.445
Zookeeper	116.981	2.523	120.003	2.002	70.993	3.066

To evaluate the effectiveness of individual components in LogShrink, we perform an ablation study among the full LogShrink, LogShrink without clustering-based sequence sampling (denoted as w/o Sampler), and LogShrink without commonality and variability analyzer (denoted as w/o Analyzer). The experiment results of both compression ratio and speed are presented in Table 7.

The contribution of commonality and variability analyzer can be observed in the comparison between LogShrink and LogShrink w/o Analyzer. We can see from the results that LogShrink can achieve an improvement of 23.12% in CR on average. Specifically, LogShrink achieves an improvement ranging from 2.92% (on Proxifier) to 64.78% (on Zookeeper) in terms of CR compared to LogShrink without Analyzer. The improvement largely depends on the amount of commonality and variability in log data. It is worth noting that LogShrink w/o Analyzer also stores the log parsing results using the column-oriented format. From the results, we can see that without Analyzer, LogShrink can achieve 1.1 × to 3.8 × CR compared to lzma, which is consistent with Observation 2 in Section 3. In terms of CS, LogShrink performs 0.64 × slower than LogShrink w/o Analyzer. It is a trade-off to consider both CR and CS in practical usage. The CS with a higher CR is probably slower than the CS with a lower CR.

The clustering-based sequence sampling contributes to the improvement of CS. In Table 7, we can observe that LogShrink is faster than the LogShrink w/o Sampler by 8.03% to 42.52%, which is consistent with the expectation. In terms of CR, LogShrink achieves comparable results on 13 out of 16 datasets although we perform the commonality and variability analysis on a small number of representative log sequences out of all log sequences. However, not all datasets can mine the representative log datasets effectively. LogShrink shows a slight decrease of 10.0% to 14.7% in dataset Windows and HDFS. This is because the number of log sequence types in Windows and HDFS are much higher than others, their CR will be significantly impacted by the parameters set in clustering-based sequence sampling.

5.4 RQ3: The Impact of Different Settings

In this RQ, we explore the impact of critical parameters (i.e., the window length h to form a log sequence, the number of sampling samples M, the distance threshold θ, and the zip tool) on LogShrink. Due to the space constraint, we show the evaluation results on 3 out of 16 datasets (i.e., BGL, HDFS, and Spark).

Impact of windows length h in sequence sampling. A larger window length h can involve more log messages and generate more log sequence types. We run LogShrink with different window lengths h in the range [5,50]. Figure 7(a) presents the experimental results. We can observe that the CR in all datasets has a sheer increase when h changes from 5 to 10. But for other h in [10,50], all datasets except HDFS show relatively stable results in CR. Hence, we select $h = 20$ as the default, which shows a relatively high CR.

Impact of M sampled candidates in sequence sampling. A higher M means that more log sequences can be fed into the analyzer. We adjust M from 2^0 to 2^7 and observe the changes in CR among the three datasets. The experiment results are illustrated in Figure 7(b). From the figure, we can see that the CR increases with rising M. Especially, the CR of HDFS is significantly affected by the parameter M, showing a 23.1% increase in $M = 128$ compared to $M = 1$. From the results, we select $M = 16$ as the default in LogShrink , considering the trade-off between CR and CS.

Impact of the distance threshold σ in sequence sampling. A larger distance threshold θ represents that the coverage of one cluster is wider and the number of clusters is less. Figure 7(c) shows the experimental results of the distance threshold θ in the range of [1,10]. The CR of BGL shows a steady decline with θ rising in the range [2,10]. However, the CR of the other two datasets remains stable as θ grows. From the results, we can see that the distance threshold θ insignificantly affects the CR, thereby it is set to 4 by default.

Impact of different zip tools used in compressor. LogShrink's CR is influenced by the zip tools employed in the compressor. To demonstrate this, we experimented with different zip tools (i.e., gzip, bzip2, lzma) on LogShrink and two other state-of-the-art methods (i.e., LogReducer, LogZip). Due to the space limit, we present the evaluation results for the top 7 largest datasets in Figure 8. The consistent superiority of LogShrink is observed across different zip tools. In comparison to the best CR performance among all baselines, LogShrink exhibits an improvement of 15.07%, 30.67%, and 15.30% on average with gzip, bzip2, and lzma, respectively.

6 THREATS TO VALIDITY

We have identified the following threats to validity:

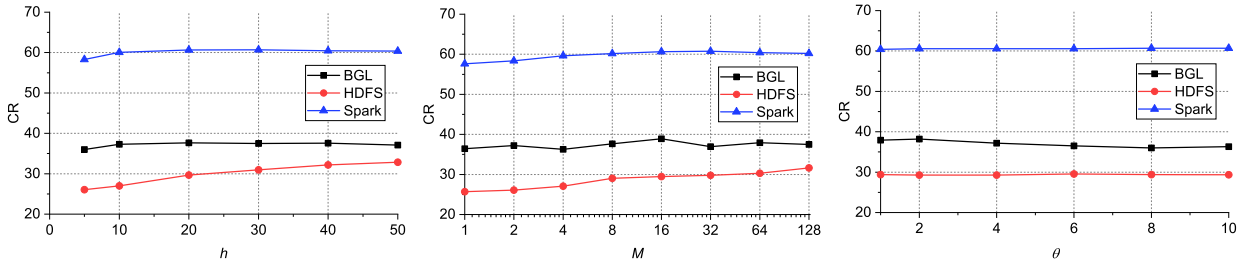
Subject systems: We only conduct our empirical study on three representative datasets. Also, our experiments are performed on a limited number of log datasets from 16 subject systems, which cannot represent all software systems. In the future, we plan to collect more log data and evaluate our methods on more software systems.

Implementation: The runtime performance of a program can differ across different programming languages. We have implemented LogShrink using Python, which prioritizes code readability over execution speed. According to a comparison of the speed of programming languages [62], the performance of C/C++ is 10 × as fast as Python. As a result, the compression speed of LogShrink is slower compared to compressors [17, 16, 18, 11], which are written in faster programming languages like C++. This language bias can affect the comparison of compression speed. To address this, we plan to optimize the LogShrink implementation in other faster programming languages, such as C++, in the future.

Tool comparison: In our evaluation, we compared LogShrink with three general-purpose compressors (i.e., lzma [16], gzip [17], bzip2 [18]) and two state-of-the-art log-specific compressors (i.e., LogZip [20], LogReducer [11]). To reduce the threat from tool comparison, we directly use the code provided by their papers [60, 61]. Also, we use the popular implementations of the general-purpose compressors [16, 17, 18].

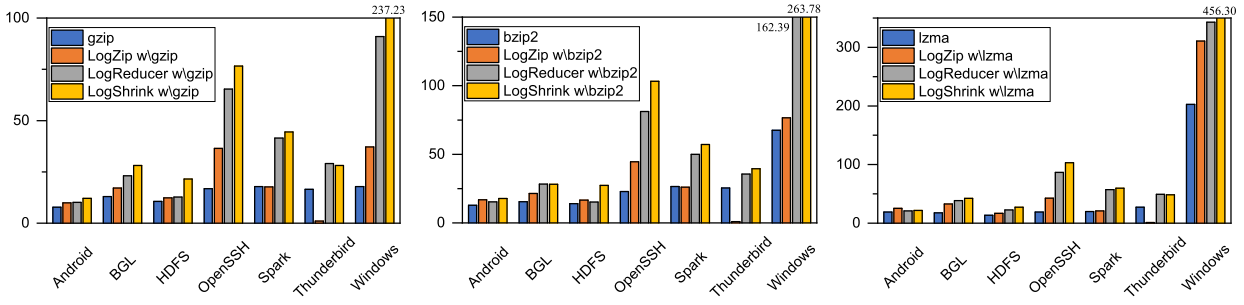
7 CONCLUSION

The sheer volume of log data presents a significant challenge for storage costs. Current compression methods, including general-purpose and log-specific methods, have limited capability in utilizing the characteristics of log data. We have conducted an empirical study on the characteristics of log data and derived three major observations, which led us to propose LogShrink, an effective log compression method, by leveraging commonality and variability of log data. Our experimental results show that LogShrink can outperform existing compressors by 16% to 356% on average with respect to compression ratio while maintaining reasonable compression



(a) Impact of window length h (b) Impact of sampling samples M (c) Impact of distance threshold θ

Figure 7: Impact of different settings in clustering-based sequence sampling



(a) gzip (b) bzip2 (c) lzma

Figure 8: Impact of different zip tools used in compressor

speed. In the future, we will optimize the LogShrink implementation in faster programming languages such as C++.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers for their insightful reviews. This research is supported by the National Key Research and Development Program of China (2019YFB1804002), the National Natural Science Foundation of China (No.62272495), the Guangdong Basic and Applied Basic Research Foundation (No.2023B1515020054), Australian Research Council Discovery Projects (DP200102940, DP220103044). Xiaoyun Li is supported by the China Scholarship Council (202206380116) during this work.

REFERENCES

[1] James H Andrews. "Testing using log file analysis: tools, methods, and issues". In: *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No. 98EX239)*. IEEE. 1998, pp. 157–166.

[2] Boyuan Chen et al. "An automated approach to estimating code coverage measures via execution logs". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 305–316.

[3] Vaibhav Agrawal et al. "Log-based cloud monitoring system for OpenStack". In: *2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE. 2018, pp. 276–281.

[4] Kundi Yao et al. "Log4perf: Suggesting logging locations for web-based systems' performance monitoring". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 127–138.

[5] Shilin He et al. "Identifying impactful service system problems via log analysis". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 60–70.

[6] Min Du et al. "Deeplog: Anomaly detection and diagnosis from system logs through deep learning". In: *SIGSAC'17: Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 1285–1298.

[7] Weibin Meng et al. "LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs." In: *IJCAI'19: Proc. of the 28th International Joint Conference on Artificial Intelligence*. 2019, pp. 4739–4745.

[8] Chenxi Zhang et al. "DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 623–634.

[9] Xiang Zhou et al. "Latent error prediction and fault localization for microservice applications by learning from system trace logs". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 683–694.

[10] Hao Lin et al. "Cowic: A column-wise independent compression for log stream analysis". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2015, pp. 21–30.

[11] Junyu Wei et al. "On the Feasibility of Parser-based Log Compression in {Large-Scale} Cloud Systems". In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 2021, pp. 249–262.

[12] *Logging Storage Pricing*. https://cloud.google.com/stackdriver/pricing. [Online]. 2023.

[13] Rui Ding et al. "Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis". In: *2015 USENIX Annual Technical Conference, USENIX ATC '15*. USENIX Association, 2015, pp. 139–150.

[14] Guangba Yu et al. "LogReducer: Identify and Reduce Log Hotspots in Kernel on the Fly". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 1763–1775.

[15] Xu Zhao et al. "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 565–581.

[16] *7za tool*. https://linux.die.net/man/1/7za. [Online]. 2023.

[17] *The gzip home page*. https://www.gzip.org. [Online]. 2023.

[18] *The bzip2 home page*. https://sourceware.org/bzip2/. [Online]. 2023.

[19] Hailun Ding et al. "ELISE: A Storage Efficient Logging System Powered by Redundancy Reduction and Representation Learning." In: *USENIX Security Symposium*. 2021, pp. 3023–3040.

[20] Jinyang Liu et al. "Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression". In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. IEEE, 2019, pp. 863–873.

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

[22] Kundi Yao et al. "Improving state-of-the-art compression techniques for log management tools". In: *IEEE Transactions on Software Engineering* 48.8 (2021), pp. 2748–2760.

[23] *LogHub Datasets*. https://zenodo.org/record/3227177. [Online]. 2023.

[24] Patrick Loic Foalem, Foutse Khomh, and Heng Li. "Studying Logging Practice in Machine Learning-based Applications". In: *arXiv preprint arXiv:2301.04234* (2023).

[25] Qiang Fu et al. "Where do developers log? an empirical study on logging practices in industry". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 24–33.

[26] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. "Characterizing logging practices in open-source software". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 102–112.

[27] Pinjia He et al. "Characterizing the natural language descriptions in software logging statements". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 178–189.

[28] Xiaoyun Li et al. "SwissLog: Robust and Unified Deep Learning Based Log Anomaly Detection for Diverse Faults". In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2020, pp. 92–103.

[29] Xiaoyun Li et al. "SwissLog: Robust Anomaly Detection and Localization for Interleaved Unstructured Logs". In: *IEEE Transactions on Dependable and Secure Computing* 20.4 (2023), pp. 2762–2780.

[30] Xiaoyun Li et al. "Going through the Life Cycle of Faults in Clouds: Guidelines on Fault Handling". In: *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2022, pp. 121–132.

[31] Van-Hoang Le and Hongyu Zhang. "Log-based anomaly detection with deep learning: How far are we?" In: *Proceedings of the 44th international conference on software engineering*. 2022, pp. 1356–1367.

[32] Zhongxin Liu et al. "Which variables should i log?" In: *IEEE Transactions on Software Engineering* 47.9 (2019), pp. 2012–2031.

[33] Ding Yuan et al. "Improving software diagnosability via log enhancement". In: *ACM Transactions on Computer Systems (TOCS)* 30.1 (2012), pp. 1–28.

[34] Stephen Yang, Seo Jin Park, and John K. Ousterhout. "NanoLog: A Nanosecond Scale Logging System". In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018, pp. 335–350.

[35] Van-Hoang Le and Hongyu Zhang. "Log Parsing with Prompt-based Few-shot Learning". In: *45th IEEE/ACM International Conference on Software Engineering*. IEEE, 2023, pp. 2438–2449.

[36] David E Sanger, Nicole Perlroth, and Eric Schmitt. "Scope of Russian hacking becomes clear: multiple US agencies were hit". In: *The New York Times* (2021).

[37] Christopher Bing. "Suspected Russian hackers spied on US Treasury emails-sources". In: *Reuters, Dec* 13 (2020).

[38] *ELK Stack*. https://www.elastic.co/elastic-stack. [Online]. 2023.

[39] Jacob Ziv and Abraham Lempel. "A Universal Algorithm for Sequential Data Compression". In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.

[40] David A Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[41] Ian H Witten, Radford M Neal, and John G Cleary. "Arithmetic Coding for Data Compression". In: *Communications of the ACM* 30.6 (1987), pp. 520–540.

[42] John Cleary and Ian Witten. "Data compression using adaptive coding and partial string matching". In: *IEEE transactions on Communications* 32.4 (1984), pp. 396–402.

[43] Mohit Goyal et al. "DeepZip: Lossless Data Compression Using Recurrent Neural Networks". In: *Data Compression Conference, DCC 2019*. IEEE, 2019, p. 575.

[44] Kundi Yao et al. "A study of the performance of general compressors on log files". In: *Empirical Software Engineering* 25 (2020), pp. 3043–3085.

[45] Bo Feng, Chentao Wu, and Jie Li. "MLC: An efficient multi-level log compression method for cloud backup systems". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE. 2016, pp. 1358–1365.

[46] Kirk Rodrigues, Yu Luo, and Ding Yuan. "CLP: Efficient and Scalable Search on Compressed Text Logs." In: *OSDI*. 2021, pp. 183–198.

[47] Junyu Wei et al. "LogGrep: Fast and Cheap Cloud Log Storage by Exploiting both Static and Runtime Patterns". In: *Proceedings of the 18th European Conference on Computer Systems*. 2023, pp. 452–468.

[48] Risto Vaarandi. "A data clustering algorithm for mining patterns from event logs". In: *IPOM'03: Proc. of the 3rd IEEE Workshop on IP Operations & Management*. IEEE. 2003, pp. 119–126.

[49] Joaqun Adiego, Gonzalo Navarro, and Pablo de la Fuente. "Lempel-Ziv compression of structured text". In: *Data Compression Conference, 2004. Proceedings. DCC 2004*. IEEE. 2004, pp. 112–121.

[50] Craig G Nevill-Manning, Ian H Witten, and Dan R Olsen. "Compressing semi-structured text using hierarchical phrase identifications". In: *Proceedings of Data Compression Conference-DCC'96*. IEEE. 1996, pp. 63–72.

[51] Shilin He et al. "Experience report: system log analysis for anomaly detection". In: *ISSRE'16: Proc. of the 27th International Symposium on Software Reliability Engineering*. IEEE. 2016, pp. 207–218.

[52] Bo Zhang et al. "Anomaly detection via mining numerical workflow relations from logs". In: *2020 International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2020, pp. 195–204.

[53] Bo Zhang et al. "Semi-supervised and unsupervised anomaly detection by mining numerical workflow relations from system logs". In: *Automated Software Engineering* 30.1 (2023), p. 4.

[54] Min Du and Feifei Li. "Spell: Streaming parsing of system event logs". In: *ICDM'16: Proc. of the 16th International Conference on Data Mining*. IEEE. 2016, pp. 859–864.

[55] Hossein Hamooni et al. "Logmine: Fast pattern recognition for log analytics". In: *CIKM'16: Proc. of the 25th ACM International on Conference on Information and Knowledge Management*. ACM. 2016, pp. 1573–1582.

[56] Pinjia He et al. "Drain: An online log parsing approach with fixed depth tree". In: *ICWS'17: 2017 IEEE International Conference on Web Services*. IEEE. 2017, pp. 33–40.

[57] Pinjia He et al. "A directed acyclic graph approach to online log parsing". In: *arXiv preprint arXiv:1806.04356* (2018).

[58] Peter Deutsch. *GZIP file format specification version 4.3*. Tech. rep. 1996.

[59] *Python Library tarfile*. https://docs.python.org/3/library/tarfile.html. [Online]. 2023.

[60] *Open source code of LogZip*. https://github.com/logpai/logzip. [Online]. 2023.

[61] *Open source code of LogReducer*. https://github.com/THUBear-wjy/LogReducer. [Online]. 2023.

[62] *Speed comparison of programming languages*. https://github.com/niklas-heer/speed-comparison. [Online]. 2023.