LogCluster: Log Compression with Clustering and a New Encoding Method

Yuji Ishikawa
Newcastle, Australia
81yishikawa@gmail.com

***Abstract*—The inherent nature of software-intensive systems results in the generation of vast console logs, vital for runtime status and event analysis. However, the long-term storage of these logs presents formidable challenges, including high storage demands, energy consumption, and network bandwidth utilization, consequently driving up operational costs. Existing log compression methods, both general-purpose and log-specific, have been explored, but often exhibit performance instability and potential data loss. To address these limitations, we present LogCluster, a groundbreaking approach. LogCluster leverages a unique combination of density clustering for structural pattern discovery and delta encoding for efficient numerical compression. Rigorous evaluation across 16 diverse log datasets demonstrates that LogCluster delivers significantly improved and stable lossless compression.**

***Index Terms*—Log compression, log data, clustering, delta encoding**

I. Introduction

To facilitate troubleshooting and track runtime status, software-intensive systems generate massive amounts of console logs. In large-scale deployments, this can be equivalent to 120-200 million lines of tracing logs per hour [1]–[3].

These logs are typically semi-structured text, but their formats differ significantly between software systems, increasing the complexity of the analysis. Beyond simply recording system-specific events, log messages provide essential insight for software maintenance and operations, empowering engineers to better understand system behavior and efficiently diagnose problems. [4]–[6]. Even with their diverse applications, efficiently storing logs continues to be a difficult task. The sheer volume of logs produced by large-scale, continuously operating systems exacerbates this challenge(e.g., 50 gigabytes per hour [1]). Beyond immediate troubleshooting, logs are vital for long-term analysis, including identifying duplicate problems and mining failure patterns [7], [8]. This requires archiving substantial log volumes for extended periods, with regulatory requirements often requiring years of storage for sensitive data [9]. This practice places a tremendous strain on the storage infrastructure. To mitigate these issues, researchers have proposed various techniques aimed at removing redundancy and reducing storage footprints.

The most straightforward way to reduce the volume of logs is to use a traditional general-purpose compressor, such as gzip or LZMA, which can compress a file by identifying and replacing duplicate bytes. For instance, Yao et al. [10], an analysis of 12 general-purpose compressors applied to log data, demonstrated their inadequacy, as they disregarded key characteristics like local repetitiveness. This inability to exploit the inherent structure of log messages resulted in suboptimal compression performance. To address this problem, some studies [2], [11] propose extracting the special structure of logs (i.e., template and variables) by log parsing [12], [13] to reduce the amount of information needed to store. Liu et al. [2] introduced Logzip, which preprocesses logs by grouping similar lines into templates and variable sets. Similarly, Wei et al. [11] LogReducer utilizes log parsing and elastic encoders to correlate and summarize variables. Despite these techniques, both Logzip and LogReducer are hindered by random bias, resulting in low or inconsistent compression rates.

To mitigate the performance deficiencies of existing log compression strategies, we propose LogCluster, a novel approach. LogCluster leverages HDBSCAN, a density-based clustering algorithm, to derive a representative log sample for log parser construction. This parser enables the extraction of log templates and variables. Subsequently, a novel delta encoding scheme is employed for numerical variable compression, while a dictionary-based compressor is utilized for other log attributes, such as templates and headers. To assess LogCluster's effectiveness, we conducted a comprehensive evaluation using 16 benchmark datasets. We compared LogCluster against established baselines, including Logzip, LogReducer, and general-purpose compressors such as gzip and 7z (LZMA). The results consistently showed that LogCluster achieved higher compression rates than the baseline methods in most datasets.

The major contributions of this paper are as follows:

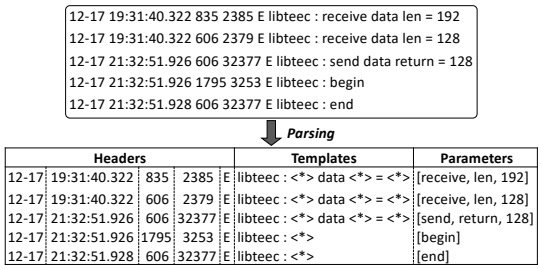
- We propose LogCluster, a novel log compression tool designed to efficiently process log data. LogCluster first employs a density-based clustering algorithm and log parsing to sample and analyze logs, revealing underlying structures. Subsequently, a novel delta encoding compressor is used to compress numerical values effectively.
- We assess LogCluster's performance on 16 public datasets, revealing its capacity to deliver not only higher compression rates but also greater stability and accurate decompression.
- We release the source code and experimental data of LogCluster[1].

[1]https://anonymous.4open.science/r/LogCluster-F65E

II. Background

A. Log Data and Structure

During system operation, logs are produced at runtime from logging statements embedded in the source code. These logs capture the system's events and internal states. The structure of a log message is determined by the specific logging framework employed and usually includes message headers and content.



Headers					Templates	Parameters
12-17	19:31:40.322	835	2385	E	libteec : <*> data <*> = <*>	[receive, len, 192]
12-17	19:31:40.322	606	2379	E	libteec : <*> data <*> = <*>	[receive, len, 128]
12-17	21:32:51.926	606	32377	E	libteec : <*> data <*> = <*>	[send, return, 128]
12-17	21:32:51.926	1795	3253	E	libteec : <*>	[begin]
12-17	21:32:51.928	606	32377	E	libteec : <*>	[end]

Fig. 1. An example of log messages and their structure

The figure 1 provides an example of a log message and its structural components from the Android system. The message headers, the format of which is defined by the logging framework, encapsulate essential information, including the timestamp, verbosity level, and process ID (PID). For example, in Figure 1, "`19:31:40.322`", "`21:32:51.926`", and "`21:32:51.928`" refer to the printed time of log messages. PIDs include "`835`", "`606`", and "`1795`". Developers determine the content of log messages, which is typically structured into two parts: a constant portion consisting of keywords that define the event template and a variable portion containing dynamic runtime information such as IDs or IP addresses.

Log data is characterized by substantial redundancies, such as structural and word repetition. To address this, log parsing serves as an essential preprocessing stage for log compression. By identifying and extracting common event templates and variable parameters, log parsing effectively reduces structural redundancy and decreases the overall data volume requiring compression.

B. Log Parsing

The function of a log parser is to first determine the template for each log entry and then extract the associated variables. There are many log parsing techniques, including frequent pattern mining [14], [15], clustering [13], [16], heuristics [12], [17], etc. The heuristic-based approaches make use of characteristics of logs to build a *parsing tree* and have been found to perform better than other techniques [18].

The parsing tree structure consists of a root node and leaf nodes, where each leaf node encapsulates a cluster of log templates with identical token lengths. To initiate tree construction, the parser performs tokenization of log messages, employing predefined delimiters, including whitespace and commas, to segment messages into token sequences. A log parser is used to identify the template of each log entry and extract the corresponding variables. For example, the log template "`libteec : <*> data <*> = <*>`" can be extracted from the first log message in Figure 1 associated with parameters [`receive, len, 192`]. Here, "`<*>`" denotes the position of a parameter. Over years of research and development, log parsing has become a crucial tool for log analysis tasks, such as anomaly detection [4], [6], [19], duplicate issue identification [8], [20], and failure diagnosis [21]. While directly matching logs to source code is a simple log parsing concept, it's impractical due to its ad-hoc nature and the extensive labeling required for diverse log formats [22].

Alternatively, manual creation of regular expressions or grok patterns can extract log templates and parameters. However, rapid software development necessitates frequent manual updates to these rules, resulting in low scalability and accuracy [18]. In response to these challenges, recent years have seen the development of data-driven log parsers, which leverage data mining techniques, including frequent pattern mining [23], [24], clustering [15], [16], heuristics [12], [17], etc. The core objective of these log parsing methods is to differentiate log messages into common templates and variable parameters.

III. Proposed Method

This section details LogCluster, our proposed log compression method. We begin with an overview of LogCluster's workflow, followed by a detailed description of each component. Finally, we outline the decompression process for recovering the original log dataset.

A. Overall Architecture

LogCluster processes log batches to compress large log files. The workflow, as depicted in the figure2, commences with the separation of log messages into header and content components using regular expressions. Subsequently, a clustering-based algorithm is employed to sample a minimal yet representative subset of logs for log parser construction. Log headers, event templates, and parameters are then extracted from each log message. The extracted values are further classified into numerical and string types. String values and templates are encoded using a dictionary-based approach, while numerical values are compressed using a novel mode-based delta encoding scheme, $arrect$-delta encoding, where values are differenced against the most frequent vertical mode.

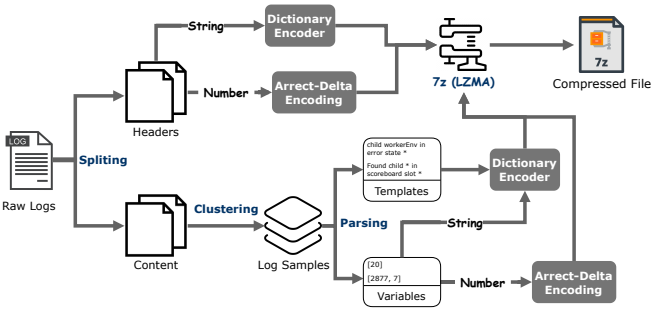


Fig. 2. LogCluster Architecture

B. Clustering

Log parsers are vital for reducing redundancy and enabling efficient log compression. However, their execution time becomes a significant bottleneck when processing large datasets, as evidenced by the 20-day parsing time for AliCloud's production logs [11], thereby slowing down the overall compression process. Therefore, to optimize the speed of log parsing yet maintain its effectiveness, we propose to use a density-based clustering method, namely HDBSCAN [25] (i.e., Hierarchical Density-Based Spatial Clustering), for sampling an optimal set of log samples. Initially, we randomly shuffle the log messages within the content to mitigate any potential bias in the distribution. Subsequently, we employ a pre-trained BERT model [26] to generate a semantic vector for each log message, capturing its contextual meaning. Then, FastICA [27] algorithm reduces the dimension of feature vectors. The generated semantic vectors are utilized as input to the HDBSCAN algorithm [25], enabling the clustering of sampled log messages. Following cluster formation, we employ a distribution-aware sampling strategy to select representative instances from each cluster. Algorithm 1 describes the proposed sampling algorithm.

Algorithm 1 processes a log batch $\mathcal{D}$, utilizing a sampling ratio r (default 1%) and a specified number of sampling rounds N_{round} (default 5). The algorithm is initiated by computing the number of log messages to be sampled per round (N_{sample}) and generating a randomly shuffled log message set ($D_{shuffle}$) in lines 1 and 2. Subsequently, lines 3-5 initialize three core components for further operations; (1) an empty set $\mathcal{S}$, which is the result of the algorithm; (2) s_{index}, which is the index of the first log message in $D_{shuffle}$ for the first sampling round; (3) e_{index}, which is the index of the last log message in $D_{shuffle}$ for the first sampling round. The iterative sampling process, spanning lines 6-19, selects N_{sample} log messages for each round. Within this loop, lines 7-10 perform the following steps: extraction of semantic vectors, dimensionality reduction of these vectors, and clustering of the log messages using the HDBSCAN algorithm [25]. Lines 11-14 then implement a cluster size-based sampling strategy using an inverse function. This function calculates the sampling rate IC for each cluster $\hat{c}$ as follows:

$$IC = \frac{(1 - \frac{len(\hat{c})}{len(D)})}{len(C) - 1} \tag{1}$$

where $len(C)$ is the number of clusters ($len(C) > 1$), $len(\hat{c})$ is the size of cluster $\hat{c}$, and $len(D)$ is the total number of all log messages. This step aims to select a diverse set of logs based on the assumption that larger clusters have more similar logs while smaller clusters contain more discrete logs. In lines 15-17, the indexes of logs in the next round (i.e., s_{index} and e_{index}) are computed. The outer loop repeats N_{round} times, and the algorithm returns the set of sampled logs $\mathcal{S}$ as the final output.

Algorithm 1: Clustering-based Sampling Algorithm

Data: $\mathcal{D}$: Log batch
r: The sampling ratio
N_{round}: The number of sampling rounds
Result: $\mathcal{S}$: the sampled log messages

1 $N_{sample} \leftarrow \frac{\text{len}(\mathcal{D}) \times r}{N_{round}}$ // each round's number of samples
2 $\mathcal{D}_{shuffle} \leftarrow$ shuffle($\mathcal{D}$) // shuffle all log messages in $\mathcal{D}$
3 $\mathcal{S} \leftarrow \emptyset$
4 $s_{index} \leftarrow 0$
5 $e_{index} \leftarrow \frac{\text{len}(\mathcal{D})}{N_{round}}$
6 **while** $N_{round} > 0$ **do**
7 $\quad \mathcal{T} \leftarrow D_{shuffle}[s_{index} \rightarrow e_{index}]$
8 $\quad E \leftarrow BERT(\mathcal{T})$ // extract semantic feature vectors
9 $\quad V \leftarrow FastICA(E)$ // reduce feature vectors' size
10 $\quad C \leftarrow HDBSCAN(V)$ // clustering
$\quad$ /* select samples from each cluster */
11 $\quad$ **for** $\hat{c} \in C$ **do**
12 $\quad\quad IC \leftarrow \frac{1 - len(\hat{c})/len(D)}{len(C) - 1} \times N_{sample}$
13 $\quad\quad \mathcal{S}$.**add**(random IC samples from cluster $\hat{c}$)
14 $\quad$ **end**
15 $\quad s_{index} \leftarrow e_{index}$
16 $\quad e_{index} \leftarrow \textbf{MIN}\Big(\text{e}_{index} + \frac{\text{len}(\mathcal{D})}{N_{round}}, \text{len}(\mathcal{D})\Big)$
17 $\quad N_{round} \leftarrow N_{round} - 1$
18 **end**
19 **return** $\mathcal{S}$

C. Encoding

1) Dictionary-based Encoding: In practice, many log messages share the same template. For example, the HDFS dataset contains around 11.2 million log messages, but they share only 39 templates [2]. The inherent length of log messages leads to significant storage overhead. Moreover, log data typically contains numerous lengthy, inseparable, and high-frequency fields, including node IDs, block IDs, and file paths. To mitigate these storage inefficiencies, we apply dictionary-based encoding, which effectively compresses these lengthy values into compact representations. Figure 3 illustrates the process of dictionary-based encoding of textual (i.e., string) information of log messages in Figure 1.

We utilize a dictionary structure composed of key-value pairs, where auto-incrementing IDs serve as keys and corresponding encoded representations (e.g., block IDs, file paths) constitute the values. To enhance storage efficiency, a vertical-based dictionary encoding method is applied, which processes each header field independently. As illustrated in Figure 3,

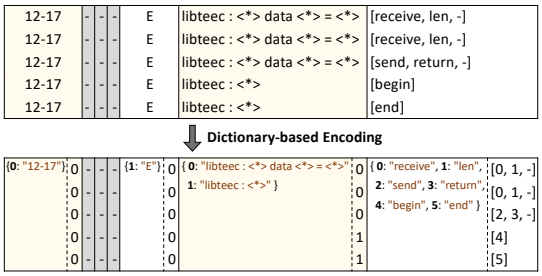


Fig. 3. An illustration of dictionary-based encoding

unique values are extracted from each header field, and dedicated dictionaries are constructed to store these values. A separate dictionary is created to map all string parameters. Consequently, for the log messages depicted in Figure 3, four distinct dictionaries are generated to encode the textual information associated with the date, verbosity level, log templates, and string parameters.

2) Arrect-Delta Encoding: During system runtime, log data accumulates a range of numerical values, including status codes, data sizes, and timestamps, which are crucial for recording dynamic runtime information. Timestamp data, when stored in `hour:minute:second` format, poses a storage challenge due to the 86,400 distinct values generated daily. To optimize compression of these numerical values, we present *arrect*-delta encoding, a novel method based on delta encoding [28]. Figure 4 illustrates the *arrect*-delta encoding process, demonstrating its application to compress numerical values, such as timestamps and data sizes, derived from log messages.

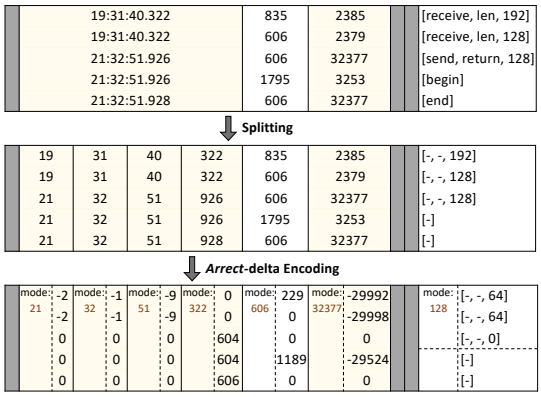


Fig. 4. An illustration of *arrect*-delta encoding

Arrect-delta encoding introduces a vertical splitting strategy for numerical values (e.g., timestamps), segmenting them into units such as hours, minutes, and seconds, resulting in position-specific groups. This grouping is extended to numerical parameters from the same log templates. As depicted in Figure 4, timestamp values are extracted into hour, minute, second, and millisecond groups, exemplified by `[19, 19, 21, 21, 21]` for hours. For each group, the *mode* (most frequent value) is computed, and all values within the group are differenced against their respective *mode*. This *mode*-based differencing strategy yields a six-fold reduction in unique patterns compared to conventional delta encoding methods.

IV. Evaluation

A. RQ1: The effectiveness and efficiency of LogCluster

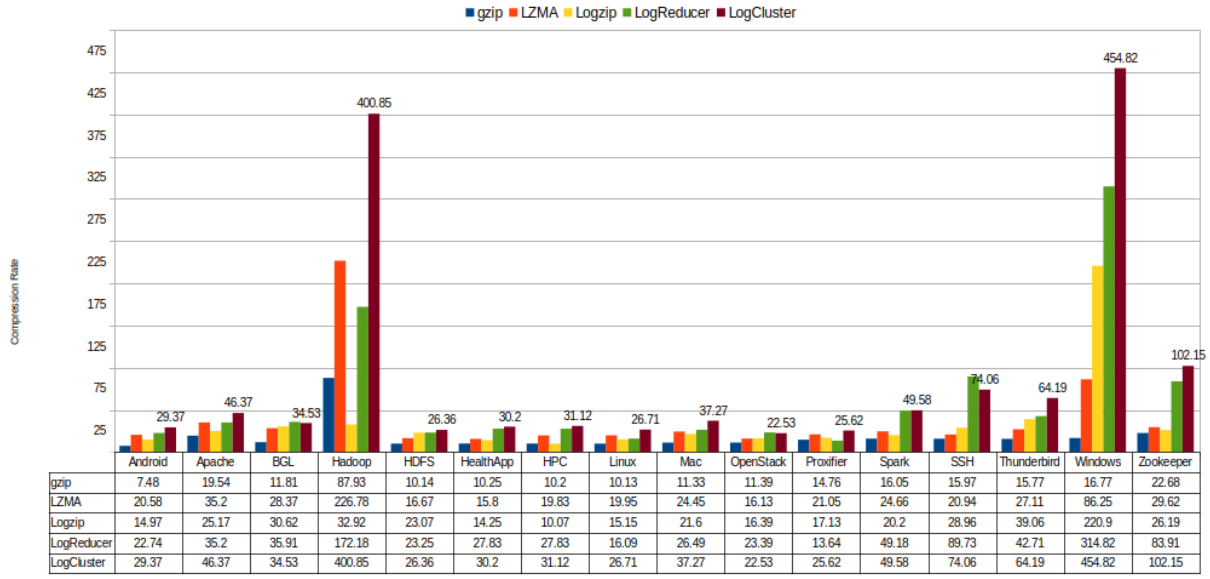
In this RQ, we compare LogCluster with four baselines (including gzip [29], LZMA [30], Logzip [2], and LogReducer [11]) on all 16 log datasets. First, we compare the effectiveness of LogCluster in terms of *Compression Rate* with baselines. The results are shown in Figure 5.

From the results, we can see that our proposed approach outperforms baseline methods on 14 out of 16 datasets. LogCluster can compress all 66 GB log datasets into 1.15 GB in total, which takes only 1.76% space after compression. For example, on the largest dataset (i.e., Thunderbird), LogCluster can compress to 0.50 GB from the original size of 31.8 GB, which saves 98.44% of storage space. In terms of compression rate, LogCluster exceeds the most powerful log compression method (i.e., LogReducer) by achieving 1.01× (Spark) to 2.33× (Hadoop) compression rates compared to LogReducder on 14 out of 16 datasets. It is worth noting that LogCluster achieves comparable results on the rest two datasets (i.e., OpenStack and SSH) compared to LogReducer. LogCluster significantly outperforms Logzip, which is also a log-specific compression method, by achieving 1.13× (BGL) to 12.18× (Hadoop) compression rates. Compared to general-purpose compression methods (i.e., gzip and LZMA), it achieves 1.22× to 27.12× compression rates. While LogReducer, the best baseline method, achieved comparable compression rate values to LogCluster on certain datasets, such as Spark and HealthApp, our observations revealed a high degree of instability in LogReducer's compression rates.

Given the performance variability of log-specific compression methods due to random biases [2], we evaluated LogCluster's robustness by conducting 30 independent experiments on each dataset, each with a different random seed. The compression rate was recorded for each experiment. Figure 6 illustrates the compression rate distribution of LogCluster across these runs for 16 datasets, with box plots depicting the minimum, 25th percentile, median, 75th percentile, and maximum compression rate values. Datasets with significantly higher compression rate values, such as Windows and Hadoop, were excluded from Figure 6 to maintain visual clarity.

The experimental results indicate that LogCluster effectively mitigates the impact of random biases, as evidenced by the consistently low variance across 30 repeated runs. Standard errors ranged from 0.02 to 0.06, with the Apache dataset exhibiting a median compression rate of 45.72 and a standard error of 0.03.

Therefore, LogCluster achieves stability through its clustering-driven approach, which enables the selection of a diverse log sample for parsing. In contrast, LogReducer's random sampling method introduces variability, making its results less consistent.



	Android	Apache	BGL	Hadoop	HDFS	HealthApp	HPC	Linux	Mac	OpenStack	Proxifier	Spark	SSH	Thunderbird	Windows	Zookeeper
gzip	7.48	19.54	11.81	87.93	10.14	10.25	10.2	10.13	11.33	11.39	14.76	16.05	15.97	15.77	16.77	22.68
LZMA	20.58	35.2	28.37	226.78	16.67	15.8	19.83	19.95	24.45	16.13	21.05	24.66	20.94	27.11	86.25	29.62
Logzip	14.97	25.17	30.62	32.92	23.07	14.25	10.07	15.15	21.6	16.39	17.13	20.2	28.96	39.06	220.9	26.19
LogReducer	22.74	35.2	35.91	172.18	23.25	27.83	27.83	16.09	26.49	23.39	13.64	49.18	89.73	42.71	314.82	83.91
LogCluster	29.37	46.37	34.53	400.85	26.36	30.2	31.12	26.71	37.27	22.53	25.62	49.58	74.06	64.19	454.82	102.15

Fig. 5. Comparison with baseline log compression methods

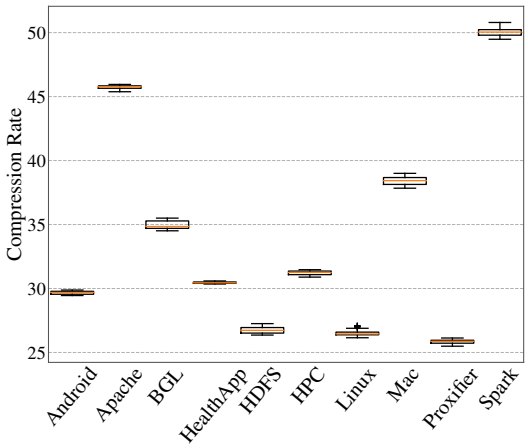


Fig. 6. Compression rate distribution of LogCluster

An evaluation of LogCluster's time efficiency was conducted on the subject datasets. LogCluster demonstrated an average compression speed of 1.40 MB/s, with speeds ranging from 0.49 to 5.00 MB/s. This speed is lower than that of general-purpose text compression tools such as LZMA and gzip, primarily because of the extensive template counts in log datasets, which slow down the matching process. Compared to Logzip, LogCluster achieved a speed improvement of 1.15× to 8.26× (2.08× on average). However, LogCluster's Python implementation results in lower compression speeds compared to LogReducer's C++ implementation.

The comprehensive experimental evaluation confirms that LogCluster provides an effective and efficient solution for log compression. Notably, it successfully mitigates the impact of random biases, resulting in stable performance across diverse experimental iterations.

B. RQ2: Results with different clustering

LogCluster is a combination of different components. In this RQ, we would like to evaluate the performance of LogCluster when different clusters are used for its major components.

First, we verify the choice of the clustering algorithm (i.e., HDBSCAN [25]) in the data sampling phase. For the purpose of this evaluation, we experimented with LogCluster by replacing the HDBSCAN algorithm with various clustering algorithms, including:

- **DBSCAN** [31]: A Density-Based Spatial Clustering of Applications with Noise, which finds core samples of high density and expands clusters from them.
- **Mean Shift** [32]: A centroid-based algorithm, which assigns the data points to the clusters iteratively by shifting points towards the highest density of data points (i.e. cluster centroid).
- **Affinity Propagation** [33]: Affinity Propagation creates clusters by sending messages between data points until convergence.

We chose these clustering methods for our evaluation based on their practical benefits and demonstrated performance. First, they do not necessitate a predetermined number of clusters (log templates), a value often unavailable. Second, they have been shown to be effective and efficient, with widespread application in diverse fields [34]–[36]. Table I shows the results.

Analysis of the results reveals that LogCluster maintains relatively high compression rates when employing different clustering algorithms. Notably, DBSCAN and Mean Shift demonstrate comparable performance to HDBSCAN on certain datasets, indicating their ability to identify optimized log message sets for parsing. Nevertheless, HDBSCAN consis-

TABLE I
LOGCLUSTER: RESULTS WITH 4 DIFFERENT CLUSTERING ALGORITHMS

Dataset	HDBSCAN	DBSCAN	Mean Shift	Affinity Propagation
Android	29.37	27.03	28.67	26.93
Apache	46.37	46.04	46.27	46.86
BGL	34.53	34.48	45.62	33.98
Hadoop	400.85	369.44	375.73	355.24
HDFS	26.36	24.29	26.44	19.11
HealthApp	30.44	30.06	30.25	30.01
HPC	31.45	30.31	30.86	30.09
Linux	26.81	26.45	26.52	25.66
Mac	38.84	30.90	33.88	33.35
OpenStack	23.24	20.52	22.41	19.10
Proxifier	26.42	26.21	25.78	24.08
Spark	49.58	38.57	48.89	28.50
SSH	74.3	73.19	73.98	62.73
Thunderbird	64.19	62.01	63.66	53.32
Windows	454.82	348.39	441.13	295.87
Zookeeper	102.15	99.76	98.97	99.49

tently outperforms other tested clustering methods across the majority of datasets.

C. RQ3: Results with different parsing

Log parsing is an essential preprocessing step in log compression, primarily because it effectively minimizes structural redundancy within log messages. In this RQ, we explore different log parsers, including:

- **Spell** [17]: Spell is an online streaming parser, which utilizes the longest common subsequence-based approach to parse system event logs.
- **AEL** [14]: AEL separates log messages into multiple groups by comparing the occurrences between constant tokens and variable tokens.
- **SHISO** [37]: SHISO builds a tree of log messages based on log format similarity to mine and refines log format continuously for online parsing.

The demonstrated effectiveness and efficiency of these log parsers have resulted in their widespread deployment within both industrial and academic settings [18]. Table II shows the results.

TABLE II
LOGCLUSTER: RESULTS 4 WITH DIFFERENT LOG PARSERS

Dataset	LogCluster	Spell	AEL	SHISO
Android	29.37	-	28.82	-
Apache	46.37	45.29	39.39	40.16
BGL	34.53	34.31	35.33	-
Hadoop	400.85	68.46	80.68	54.69
HDFS	26.36	25.21	24.64	26.40
HealthApp	30.20	18.63	26.43	-
HPC	31.12	46.19	42.51	26.55
Linux	26.71	23.39	26.76	26.32
Mac	37.27	29.65	36.74	29.75
OpenStack	22.53	33.12	48.97	18.56
Proxifier	25.62	23.44	29.13	26.44
Spark	49.58	-	-	-
SSH	74.06	70.43	50.68	49.82
Thunderbird	64.19	60.01	59.82	65.98
Windows	454.82	-	-	-
Zookeeper	102.15	-	-	-

Note: '-' denotes timeout (30 hours).

The selection of a log parser significantly impacts the compression efficiency of LogCluster. Certain log parsers, including Spell and SHISO, exhibit scalability limitations, failing to complete parsing and compression within 30 hours on four and six datasets, respectively. The empirical results confirm that Drain provides the best performance for LogCluster, achieving a compression rate advantage of 1.07× to 7.33× over Spell, AEL, and SHISO.

D. RQ4: Losslessness of LogCluster

Accuracy is paramount in many applications, especially those dealing with critical information. Lossless compression guarantees that the data remains accurate after compression and decompression. LogCluster decompresses datasets it has previously compressed. Due to the high computational cost associated with processing large datasets, this experiment excluded substantial data volumes such as Windows and Thunderbird. Our initial observation shows that LogReducer's compression performance is comparable. To further evaluate these tools, we conduct an experiment specifically designed to compare the losslessness of LogCluster and LogReducer. Table III shows the results.

In terms of average losslessness, LogCluster and LogReducer show similar results. However, a deeper analysis reveals notable differences in their minimum performance and frequency of loss. Specifically, LogCluster's lowest losslessness rate is 95.9%, while LogReducer maintains a minimum of 99.09%. Additionally, LogCluster recorded seven instances of losslessness, compared to only three for LogReducer. This discrepancy can be attributed to LogCluster's sensitivity to dataset structure. It excels with organized data like SSH, but its losslessness suffers when encountering irregularities such as empty lines or tab spaces, as seen in Hadoop.

TABLE III
RESULT OF LOSSLESSNESS

Dataset	LogCluster	LogReducer
Android	98.28 %	99.58 %
Apache	100 %	99.88 %
BGL	97.34 %	100 %
Hadoop	95.9 %	99.26 %
HDFS	99.27 %	99.98 %
HealthApp	98.11 %	99.95 %
HPC	100 %	100 %
Linux	100 %	99.7 %
Mac	98.54 %	99.09 %
OpenStack	99.51 %	100 %
Proxifier	100 %	99.81 %
Spark	100 %	100 %
SSH	100 %	99.77 %
Zookeeper	100 %	99.95 %
Average	99.06 %	99.78 %

Note: 100% equals full losslessness.

V. THREATS TO VALIDITY

We have identified the following threats to validity:

- Character encoding: LogCluster's current implementation is focused on the compression and decompression of log data encoded in UTF-8. Consequently, it does not provide full restoration capabilities for datasets containing UTF-16 characters, exemplified by Kanji. Support for UTF-16 encoding will be incorporated in future development efforts.

- Empty lines: LogCluster maintains high accuracy in log compression and decompression operations. Nevertheless, a current limitation, similar to LogReducer, is the potential omission of a limited number of empty lines within the log data. While this does not alter the semantic interpretation of the log data, it does prevent perfect lossless compression. This issue is slated for resolution in future development iterations.
- Diverse log formats: The evaluation presented herein examined the performance of LogCluster using 16 public log datasets. Given the extensive variety of logging formats encountered in real-world systems, future studies will encompass a broader spectrum of log datasets to provide a more thorough assessment of LogCluster's capabilities.

VI. Related Work

A. General-purpose compression

Data compression is a critical component of modern software systems, as it effectively reduces the storage space required for files and minimizes the time necessary for data transfer or download. General-purpose compression methods have been the subject of extensive development for many years [30]. These methods can be categorized into three groups: statistic-based, predict-based, and dictionary-based. Statistic-based compression methods, such as Arithmetic encoding [38], Huffman coding [39], and Run-Length (RLE) encoding [40], collect statistic information about inputs (e.g., the occurring probabilities) and then design a variant length coding for each token. Predict-based compression methods, such as PPMd [41] and PPM* [42], predict the next token based on the set of previous tokens and assign a shorter encoding if the prediction is successful. Dictionary-based compression algorithms, such as LZMA [30] and gzip [29], search for similar tokens to encode them as a single token and store them in a dictionary. Recently, machine learning-based data compression has been proposed [43]–[45] and achieved better compression rates benefiting from the recent advances in machine learning and deep learning [43], [46]. For example, NNCP [43] utilizes natural language models such as LSTM [47] and transformers [48] techniques for compressing text data. DeepZip [46] employs a deep neural network (DNN) and arithmetic encoding to locate statistical redundancies, thus improving compression performance. However, despite achieving state-of-the-art compression rates, the significant training time required by these deep learning-based methods limits their efficiency in log compression applications.

B. Log-specific compression

In an effort to enhance the compression of log files, researchers have proposed log-specific compression methods in recent years, aiming for higher compression rates. For example, CLC [49], MLC [50], LogArchive [51], and Cowic [52] process templates and parameters together to compress different log buckets in parallel. Recently, some studies [2], [11] proposed to process templates and parameters separately to achieve higher compression rates. LogZip [2] leverages an iterative clustering algorithm to identify log templates and extract corresponding parameters. It then achieves compression by substituting headers, templates, and variables with compact IDs using a series of dictionaries. While demonstrating significant compression rate enhancements compared to general-purpose compression techniques, LogZip's need to process the entire log dataset for log parser construction results in performance bottlenecks when applied to large-scale datasets. LogReducer [11] employs a random sampling strategy to construct a log parser, which extracts log templates and parameters. It implements an elastic encoding scheme to optimize numerical value compression by removing leading zeros and maps templates to unique identifiers. Variables are subsequently compressed using general-purpose compression techniques. Implemented in C++, LogReducer aims to enhance compression speed. However, the random sampling methodology introduces instability, resulting in fluctuating compression rates.

LogCluster distinguishes itself by using a clustering algorithm to select an optimized log message sample for parsing, thereby achieving more stable compression rates and enhanced speed. Furthermore, it efficiently compresses numerical information through $arrect$-delta encoding and string information using dictionary-based encoding.

VII. Conclusion

This paper proposes LogCluster, a novel log compression tool integrating two innovative techniques: clustering and encoding. The clustering method optimizes sample selection, facilitating efficient parse tree construction irrespective of log dataset scale. The arrect-delta encoding successfully mitigates timestamp pattern redundancy and is generalizable to any numerical variable. Experimental evaluations on 16 public datasets demonstrate that LogCluster achieves the highest and most stable compression rates when compared to three baseline methods.

Our source code and experimental results are available at: https://github.com/Yuji-github/LogCluster.

References

[1] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1245–1255, 2013.

[2] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, "Logzip: extracting hidden structures via iterative clustering for log compression," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 863–873.

[3] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ {MapReduce} for log processing," in *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[4] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1285–1298.

[5] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 207–218.

[6] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 807–817.

[7] A. Amar and P. C. Rigby, "Mining historical test logs to predict bugs and localize faults in the test logs," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 140–151.

[8] R. Ding, Q. Fu, J. G. Lou, Q. Lin, D. Zhang, and T. Xie, "Mining historical issue repositories to heal large-scale online service systems," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 311–322.

[9] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–37, 2021.

[10] K. Yao, H. Li, W. Shang, and A. E. Hassan, "A study of the performance of general compressors on log files," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3043–3085, 2020.

[11] J. Wei, G. Zhang, Y. Wang, Z. Liu, Z. Zhu, J. Chen, T. Sun, and Q. Zhou, "On the feasibility of parser-based log compression in {Large-Scale} cloud systems," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 249–262.

[12] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 33–40.

[13] R. Vaarandi and M. Pihelgas, "Logcluster-a data clustering and pattern mining algorithm for event logs," in *2015 11th International conference on network and service management (CNSM)*. IEEE, 2015, pp. 1–7.

[14] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting execution logs to execution events for enterprise applications (short paper)," in *2008 The Eighth International Conference on Quality Software*. IEEE, 2008, pp. 181–186.

[15] L. Tang, T. Li, and C.-S. Perng, "Logsig: Generating system events from raw textual logs," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, 2011, pp. 785–794.

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

[17] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 2016, pp. 859–864.

[18] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 121–130.

[19] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs." in *IJCAI*, vol. 7, 2019, pp. 4739–4745.

[20] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 102–111.

[21] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 353–366.

[22] Y. Liu, X. Zhang, S. He, H. Zhang, L. Li, Y. Kang, Y. Xu, M. Ma, Q. Lin, Y. Dang *et al.*, "Uniparser: A unified log parser for heterogeneous log data," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 1893–1901.

[23] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 114–117.

[24] H. Dai, H. Li, C. S. Chen, W. Shang, and T.-H. Chen, "Logram: Efficient log parsing using n-gram dictionaries," *IEEE Transactions on Software Engineering*, 2020.

[25] L. McInnes, J. Healy, and S. Astels, "hdbscan: Hierarchical density based clustering." *J. Open Source Softw.*, vol. 2, no. 11, p. 205, 2017.

[26] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. [Online]. Available: https://arxiv.org/abs/1908.10084

[27] E. Oja and Z. Yuan, "The fastica algorithm revisited: Convergence analysis," *IEEE transactions on Neural Networks*, vol. 17, no. 6, pp. 1370–1381, 2006.

[28] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for http," in *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, 1997, pp. 181–194.

[29] P. Deutsch, "Gzip file format specification version 4.3," Tech. Rep., 1996.

[30] K. Sayood, *Introduction to data compression*. Morgan Kaufmann, 2017.

[31] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "Density-based spatial clustering of applications with noise," in *Int. Conf. Knowledge Discovery and Data Mining*, vol. 240, no. 6, 1996.

[32] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE transactions on pattern analysis and machine intelligence*, vol. 17, no. 8, pp. 790–799, 1995.

[33] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *science*, vol. 315, no. 5814, pp. 972–976, 2007.

[34] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan, "Practical accuracy estimation for efficient deep neural network testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020.

[35] S. Anand, S. Mittal, O. Tuzel, and P. Meer, "Semi-supervised kernel mean shift clustering," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 6, pp. 1201–1215, 2013.

[36] K. Wang, J. Zhang, D. Li, X. Zhang, and T. Guo, "Adaptive affinity propagation clustering," *arXiv preprint arXiv:0805.1096*, 2008.

[37] M. Mizutani, "Incremental mining of system log format," in *2013 IEEE International Conference on Services Computing*. IEEE, 2013, pp. 595–602.

[38] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[39] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[40] A. H. Robinson and C. Cherry, "Results of a prototype television bandwidth compression scheme," *Proceedings of the IEEE*, vol. 55, no. 3, pp. 356–364, 1967.

[41] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE transactions on Communications*, vol. 32, no. 4, pp. 396–402, 1984.

[42] J. G. Cleary and W. J. Teahan, "Unbounded length contexts for ppm," *The Computer Journal*, vol. 40, no. 2_and_3, pp. 67–75, 1997.

[43] F. Bellard, "Nncp v2: Lossless data compression with transformer," NNCP v2: Lossless Data Compression with Transformer, 02 2021.

[44] M. V. Mahoney, "Fast text compression with neural networks." in *FLAIRS conference*, 2000, pp. 230–234.

[45] J. Schmidhuber and S. Heil, "Sequential neural text compression," *IEEE Transactions on Neural Networks*, vol. 7, no. 1, pp. 142–146, 1996.

[46] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, "Deepzip: Lossless data compression using recurrent neural networks," in *2019 Data Compression Conference, DCC 2019*. Institute of Electrical and Electronics Engineers Inc., 2019, p. 575.

[47] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[48] Y. Chen, H. Shu, W. Xu, Z. Yang, Z. Hong, and M. Dong, "Transformer text recognition with deep learning algorithm," *Computer Communications*, vol. 178, pp. 153–160, 2021.

[49] K. Hätönen, J. F. Boulicaut, M. Klemettinen, M. Miettinen, and C. Masson, "Comprehensive log compression with frequent patterns," in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2003, pp. 360–370.

[50] B. Feng, C. Wu, and J. Li, "Mlc: an efficient multi-level log compression method for cloud backup systems," in *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 2016, pp. 1358–1365.

[51] R. Christensen and F. Li, "Adaptive log compression for massive log data." in *SIGMOD Conference*, 2013, pp. 1283–1284.

[52] H. Lin, J. Zhou, B. Yao, M. Guo, and J. Li, "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.