LogLite: Lightweight Plug-and-Play Streaming Log Compression

Benzhao Tang[§], Shiyu Yang[§*], Zhitao Shen[†], Wenjie Zhang[◇], Xuemin Lin[‡], Zhihong Tian[§°]
[§]Cyberspace Institute of Advanced Technology of Guangzhou University & Huangpu Research School of Guangzhou University; [†]Ant Group; [◇]University of New South Wales; [‡]Shanghai Jiao Tong University; [°]Guangdong Key Laboratory of Industrial Control System Security
benzhaotang@outlook.com; syyang@gzhu.edu.cn; zhitao.szt@antgroup.com
wenjie.zhang@unsw.edu.au; xuemin.lin@sjtu.edu.cn; tianzhihong@gzhu.edu.cn

ABSTRACT

Log data is a vital resource for capturing system events and states. With the increasing complexity and widespread adoption of modern software systems and IoT devices, the daily volume of log generation has surged to tens of petabytes, leading to significant collection and storage costs. To address this challenge, lossless log compression has emerged as an effective solution, enabling substantial resource savings without compromising log information. In this paper, we first conduct a characterization study on extensive public log datasets and identify four key observations. Building on these insights, we propose LogLite, a lightweight, plug-and-play, streaming lossless compression algorithm designed to handle both TEXT and JSON logs throughout their life cycle. LogLite requires no predefined rules or pre-training and is inherently adaptable to evolving log structures. Our evaluation shows that, compared to state-of-the-art baselines, LogLite achieves Pareto optimality in most scenarios, delivering an average improvement of up to 67.8% in compression ratio and up to 2.7× in compression speed.

PVLDB Reference Format:
Benzhao Tang, Shiyu Yang, Zhitao Shen, Wenjie Zhang, Xuemin Lin, Zhihong Tian. LogLite: Lightweight Plug-and-Play Streaming Log Compression. PVLDB, 18(11): 3757 - 3770, 2025.
doi:10.14778/3749646.3749652

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/benzhaotang/LogLite.

1 INTRODUCTION

Logs provide detailed records of various events and states occurring during system operation, including but not limited to user actions, system errors and warnings, security-related activities, and more. They serve as essential data sources for system troubleshooting and diagnostics [11, 18, 29, 37, 43, 55, 58], performance monitoring [2, 53], security auditing [5, 15, 38], and business improvements [13, 14]. As modern software and IoT systems become increasingly large and complex, coupled with the massive growth in user numbers, logs are being generated at an astonishing rate,

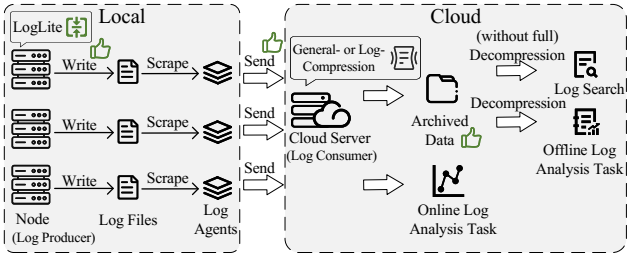


Figure 1: The life cycle of log data.

ranging from hundreds of terabytes (TB) to tens of petabytes (PB) daily. For instance, Alibaba's IoT devices transmit hundreds of TB of logs each day [4]; Uber generates over 10 PB of logs during a busy day across all its services [47]; and large real-time messaging applications like WeChat produce 16–20 PB of logs daily [54]. Due to the importance of logs and regulatory requirements, these substantial volumes of log data must often be collected centrally and preserved losslessly for extended periods. Typically, logs are retained for at least six months, and in certain important industries (such as finance and healthcare), retention periods can extend to several years [35, 41, 49, 50].

To explore the significant cost implications of log collection and storage, we present the complete life cycle of logs from generation to consumption in a distributed cluster, as shown in Figure 1. Logs generated independently by multiple nodes responsible for specific services are initially written to local storage. Once a certain amount of time elapses or the log file reaches a predefined size, the log agent marks the file as non-writable and scrapes the logs. These logs are then sent over the network to dedicated log processing machines or cloud services. For scenarios with high real-time requirements, these logs are immediately utilized for online log analysis tasks, such as real-time anomaly detection and performance monitoring [7]. The line-by-line log compression and transmission to the cloud from local nodes support real-time log processing. For other scenarios, the collected massive log data is compressed using general-purpose or log-purpose compression algorithms and archived to disk. When tasks require precise log retrieval, such as root cause analysis for system failures or forensic analysis for regulatory compliance, the archived logs need to be decompressed before log searching. Some log-purpose methods [41, 47, 49] enable log searches without full decompression. However, for offline log analysis tasks, such as resource optimization, business improvement, or security auditing, full decompression is still required.

Writing logs incurs CPU and I/O overhead, sending logs consumes network bandwidth, and archiving logs requires storage

*Shiyu Yang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749652

resources [7, 35, 54, 57]. Given the massive scale of logs, these operations result in significant cost burdens. Some log reduction methods [24, 36, 42, 54] attempt to minimize log generation at the source by selectively generating logs based on necessity, thereby directly reducing the production of non-critical logs. However, this approach inevitably sacrifices some log information, and the reduced log volumes often remain within the same order of magnitude, such as decreasing from 19.8 PB to 12.0 PB per day [54]. As a result, log compression exhibits significant potential to save resources and reduce economic costs. Some log systems involve using general-purpose compression algorithms such as LZMA [26] and Zstd [10]. These algorithms do not require any prior knowledge or rules, treating the massive log files as byte streams and replacing repetitive byte sequences with shorter representations. However, general-purpose compression methods fail to fully exploit the unique characteristics of logs [30]. To address this limitation, some log-purpose compression methods [35, 41, 47, 49, 50, 56] have been proposed, which either manually define rules or automatically extract patterns (often referred to as templates) from logs. During compression, the pattern of each log is replaced with dictionary entries, requiring only the storage of variables. By further applying general-purpose compression, higher compression ratios can be achieved. However, after investigating these approaches, we identify the following three challenges:

(1) Resource-constrained devices struggle to efficiently compress and transmit logs in real-time. In many scenarios, devices that generate logs have limited resources to store and process them, such as industrial or home IoT devices and nodes focused on their primary tasks [34]. These devices often prioritize real-time performance monitoring and anomaly detection. As a result, logs are expected to be transmitted to cloud services as quickly as possible while consuming minimal resources. However, both general-purpose and log-purpose compression methods require the logs to be collected to a sufficient size to achieve effectiveness. When logs are stored in small files or have limited samples in resource-constrained devices, these methods struggle to identify redundancies effectively. This limitation prevents these methods from delivering effective compression in resource-constrained devices.

(2) Most log compression methods require frequent adaptation due to evolving formats, patterns, and structures, limiting their plug-and-play functionality. Logs generated by different devices in diverse deployment environments often contain unique information (e.g., timestamps, IP addresses, and identifiers) and patterns. Consequently, most log-purpose methods need to know the predefined templates or be trained to learn the internal structure of the logs, which means they struggle to achieve plug-and-play functionality. Moreover, as log information and patterns evolve over time, for example, transitioning from TEXT formats to JSON, methods relying on manually defined rules or patterns extracted through sampling and training become obsolete and require additional resources for redefinition and retraining.

(3) Pattern mining or pre-training in existing log compression methods often compromises efficiency, significantly limiting their applicability in time-sensitive scenarios. Throughout the log processing life cycle, three key metrics-compression ratio, compression speed, and decompression speed-are critical. However, most log-purpose compression techniques sacrifice both compression and decompression efficiency to accommodate pattern mining and fine-grained processing. This trade-off hinders their effectiveness in high-speed environments, such as real-time log ingestion, where rapid processing is essential.

To effectively and efficiently leverage the characteristics of logs, we conduct a comprehensive characterization study on the relationship between log length and log similarity using 16 public unstructured TEXT datasets [59] and 5 semi-structured JSON datasets [47]. Through this study, we present four key observations, highlighting a strong correlation between logs of the same length and their likelihood of belonging to the same pattern. Based on these insights, we propose a novel and efficient log-purpose compression method, named LogLite. LogLite utilizes log length to perform lightweight compression and processes log data line-by-line in a streaming manner. For each incoming log, it searches in reverse order within a sliding window that caches logs of the same length to identify similar logs. It then applies our custom-designed operations: character-preserving XOR and byte-aligned run-length encoding. LogLite offers inherent plug-and-play compression for streaming logs, operating automatically without requiring predefined rules, sampling, training, or pattern mining. This design enables immediate compression of individual log entries directly at the point of generation for instant transmission. Furthermore, it seamlessly adapts to dynamic changes in log structure without consuming additional resources. Consequently, as illustrated in Figure 1, LogLite excels in real-time, streaming, and data-agnostic scenarios, such as the transmission of logs from edge nodes or IoT devices to Cloud Servers for online analysis, by significantly reducing resources for log writing and sending while preserving real-time performance. This capability stems from key advantages: high compression ratio and speed, zero reliance on prior knowledge, lightweight design, and robust adaptability. Furthermore, LogLite supports the use of general-purpose compression algorithms during log archiving, similar to other log-purpose compression methods, to achieve additional storage savings. Our contributions are summarized as follows:

- We conduct an in-depth characterization study on 21 real-world log datasets from a non-log-parsing perspective and identify four key observations, which reveal for the first time that log length is a highly general and valuable feature for log compression.
- We propose LogLite, which, to the best of our knowledge, is the first lightweight, plug-and-play, streaming, lossless log compression algorithm based on log length to compress both TEXT and JSON logs, offering excellent adaptability. We introduce three core components, including L-Windows, XOR-P and RLE-b/B, that enable LogLite to work effectively and efficiently across the entire log life cycle.
- We evaluate LogLite against SOTA log-purpose and general-purpose compression baselines on 16 unstructured TEXT and 5 semi-structured JSON log datasets. The results demonstrate that LogLite achieves Pareto-optimal performance in most cases.

2 CHARACTERIZING LOG DATA

We first introduce the characterization study on 21 widely used log datasets and four observations that motivate the design of LogLite. Previous studies [30, 47, 48] conduct characterization analyses on large-scale log data using log parsers. However, our objective is to

Table 1: Characterization Study of Unstructured(TEXT) and Semi-structured(JSON) Log Datasets.

	Log Datasets			Finite Length			Compliance Proportion					Search Count		Upper Bound	
	Name	Size(MB)	#Lines	AL	NDL	RDL	PSL (%)					Seq	Rev	MSS (%)	AN
							32	16	8	4	2				
TEXT	Apache (Apa)	4.95	56,482	90	69	0.1222%	96.4	95.8	95.1	94.4	93.5	10	2	96.4	9
	Linux (Lin)	2.27	25,567	91	208	0.8135%	91.7	90.5	88.3	86.1	83.6	9	3	92.1	7
	Zookeeper (Zoo)	9.94	74,380	139	88	0.1183%	99.7	99.7	99.6	99.5	99.2	1	1	97.8	7
	HealthApp (Hea)	22.44	253,395	92	132	0.0521%	96.3	95.6	94.8	93.3	91.2	8	2	93.0	7
	HPC	32.00	433,490	76	272	0.0627%	97.5	97.2	96.8	96.3	95.3	7	2	93.7	8
	Android (And)	183.36	1,555,005	123	720	0.0463%	86.5	82.2	76.9	71.1	64.9	10	7	88.9	10
	Hadoop (Had)	46.35	395,286	122	327	0.0827%	98.3	97.2	95.7	93.8	87.2	3	2	97.7	22
	BGL	708.76	4,747,963	156	235	0.0049%	99.0	98.7	98.2	97.7	96.8	3	1	93.9	9
	Mac	16.10	117,283	143	490	0.4178%	92.2	90.7	88.6	84.7	77.0	8	4	92.6	9
	OpenStack (Ope)	38.63	137,074	295	131	0.0956%	76.0	75.6	74.8	74.1	72.7	9	8	90.1	15
	Proxifier (Pro)	2.42	21,329	118	104	0.4876%	89.1	86.9	84.2	80.7	76.7	9	4	93.3	10
	Spark (Spa)	2805.76	33,236,604	87	284	0.0009%	84.9	83.9	83.0	82.3	81.3	7	6	95.3	13
	SSH	70.02	655,147	111	122	0.0186%	98.0	97.8	97.5	96.9	96.0	3	2	95.6	8
	Thunderbird (Thu)	30320.64	211,212,192	153	413	0.0006%	97.7	95.9	92.7	88.8	83.5	5	3	95.3	11
	Windows (Win)	26716.16	114,608,388	243	654	0.0006%	98.3	97.0	96.5	95.1	92.9	3	2	96.0	16
	HDFS	1505.28	11,175,629	140	135	0.0012%	59.0	56.7	52.2	44.5	35.1	21	15	87.3	12
JSON	MongoDB (Mon)	66355.20	186,287,600	395	370	0.0014%	99.9	99.9	99.9	99.9	95.7	1	1	99.4	14
	CockroachDB (Coc)	10024.96	24,772,932	423	2867	0.0116%	94.8	93.5	91.4	88.4	84.5	1	1	99.4	7
	Elasticserch (Ela)	8171.52	14,001,234	611	1194	0.0085%	99.7	99.7	99.6	99.4	99.2	1	1	97.4	13
	Spark	2027.52	1,011,651	2101	1718	0.1698%	97.5	96.0	93.1	89.2	84.4	3	2	96.4	11
	PostgreSQL (Pos)	392.84	1,000,000	411	265	0.0265%	99.5	98.9	97.7	94.6	92.6	2	2	97.5	13

achieve lightweight streaming compression without parsing logs or relying on prior knowledge. Thus, we perform a detailed characterization study of logs from a **non-log-parsing perspective**. To the best of our knowledge, this is the first in-depth characterization study of unstructured and semi-structured log datasets that focuses on log length while employing a non-log-parsing approach.

Table 1 presents statistics for the log datasets, comprising 16 unstructured TEXT logs from Loghub [59] and 5 semi-structured JSON logs from μSlope [47]. To ensure a comprehensive and unbiased study, we incorporate all publicly available datasets from both sources. These datasets are among the most representative and widely adopted in log compression [30, 35, 47, 49, 50, 56] and collectively reflect the diverse characteristics of log data from a broad spectrum of real-world applications.

First, we define several terms used in the analysis.

Log Dataset: A log dataset $S = (\log_1, \log_2, \ldots, \log_n)$ consists of log entries, where each element $\log_i$ is a string. The length of the n-th log entry is denoted by $len(\log_n)$.

Log Entry: Each log entry $\log_n$ is denoted as:

$$log_n = (c_1, c_2, \ldots, c_{len(\log_n)})$$

where each c_i represents a character in the n-th log entry.

Similarity Score: Motivated by the Hamming distance of strings, given two log entries $\log_n$ and $\log_m = (c'_1, c'_2, \ldots, c'_{len(\log_m)})$, their similarity score is defined in Equation (1):

$$Sim(\log_n, \log_m) = \frac{\sum_{i=1}^{len(\log_n)} \mathbf{1}_{\{c_i = c'_i\}}}{len(\log_n)} \quad (1)$$

where $len(\log_n) = len(\log_m)$, and $\mathbf{1}_{\{c_i = c'_i\}}$ is an indicator function that equals 1 when c_i is identical to c'_i, and 0 otherwise.

Adjacent Logs: The adjacent logs of $\log_n$ are the subset $(\log_{n-k}, \ldots, \log_{n-1})$, where $1 \le k < n$, representing the k preceding logs.

	[*] [notice] jk2_init() Found child * in scoreboard slot *
①	[Sun Dec 04 03:49:10 2005] [notice] jk2_init() Found child 27772 in scoreboard slot 6
②	[Sun Dec 04 03:49:10 2005] [notice] jk2_init() Found child 27774 in scoreboard slot 8
③	[Sun Dec 04 03:49:10 2005] [notice] jk2_init() Found child 27776 in scoreboard slot 10
④	[Sun Dec 04 04:51:08 2005] [notice] jk2_init() Found child 6725 in scoreboard slot 10
⑤	[Sun Dec 04 04:51:09 2005] [notice] jk2_init() Found child 6726 in scoreboard slot 8

Figure 2: Samples from Apache log.

Additionally, $\log_n$ is more adjacent to $\log_j$ than to $\log_i$ if and only if $n - k \le i < j < n$.

Next, we illustrate the log generation process, using Apache system log generation as an example, which provides insights for our key observations. During development, developers write log statements, for example, `logger.info("jk2_init() Found child {child_id} in scoreboard slot {slot_id}")`. When this log statement is executed, the variables `child_id` and `slot_id` are filled with real values, a timestamp and other information are added, and it is formatted to produce a log entry such as: `[Sun Dec 04 03:49:10 2005] [notice] jk2_init() Found child 27772 in scoreboard slot 6`. Thus, logs typically consist of pre-defined patterns combined with variables populated at runtime. This common structure inherently creates redundancy, which motivates us to derive the following observations for log compression.

Observation 1: Log lengths and their variations are practically finite. In the above log generation, we observe that each log entry typically consists of an invariant pattern and a small number of variables. As shown in Figure 2, these log entries are generated by a predefined pattern composed of three variables: a timestamp and two integers. The length of the pattern is relatively long and fixed, and the length of the variables fluctuates within a relatively small range. In practice, since the variables usually represent different

values for an identical field, possess consistent data types, and are formatted for readability in log outputs, their length variations tend to remain relatively small. For example, in Figure 2, the timestamp variable has a fixed length, the second variable typically ranges from 4 to 5 characters, and the third variable typically ranges from 1 to 2 characters. The length values for the log entries in this same pattern are 84, 85, and 86. Furthermore, for any given log dataset, the number of distinct patterns is finite. In most cases, the number of patterns is on the order of a few hundred, with more complex datasets potentially containing over a thousand distinct patterns.

As shown in Finite Length of Table 1, we calculate the **average length** (AL), **number of different lengths** (NDL), and **ratios of different lengths** (RDL) using equations (2), (3), and (4).

$$AL(S) = \frac{1}{n}\sum_{i=1}^{n} len(\log_i). \tag{2}$$

$$NDL(S) = \text{count}(\text{unique}(\{len(\log_1), \ldots, len(\log_n)\})). \tag{3}$$

$$RDL(S) = \frac{1}{n} NDL(S). \tag{4}$$

The average length (AL) of unstructured TEXT logs is generally small, with an average value of 136. Semi-structured JSON logs, due to their JSON format and the logging strategies of the systems, are larger, with an average length (AL) of 788. The average numbers of different lengths (NDL) for TEXT and JSON logs are 274 and 1283, respectively, with average RDL of 0.15% and 0.04%, respectively, where each RDL is less than 1%. This implies that, in practical applications, both log length and its variation are highly finite and do not increase drastically with growing log volume.

Observation 2: Logs with identical lengths have a strong probability of high similarity scores. Since most logs are generated in the above manner, logs belonging to the same pattern have a large number of redundant characters, and these characters are aligned by index because the pattern is fixed and the variation of the variables is very small, which means these redundant characters are suitable for elimination using XOR and run-length coding (e.g., log ① and ② in Figure 2). The more variables that have the same length between two logs, the more redundant characters are aligned. Additionally, from Observation 1, we learn that the number of different lengths in a log dataset is finite, and the pattern accounts for the majority of log length. Consequently, we naturally hypothesize that if two logs have the same length, then they have a high probability of belonging to the same pattern and having variables of the same length, meaning many of their redundant characters are aligned and they have high similarity scores. To further verify this, we calculate the **probability of high similarity scores based on length** (PSL), according to formula (5), which also represents the proportion of logs in dataset S that have a high similarity score exceeding the threshold θ. We categorize the n logs from S into $b = NDL(S)$ buckets, where each bucket contains logs of identical length and maintains the relative order of the original file. The number of logs in each bucket is denoted as $(n_1, n_2, \ldots, n_b)$.

$$PSL(S) = \frac{1}{n}\sum_{i=1}^{b}\sum_{j=1}^{n_i} \mathbf{1}\Big\{\exists q \in \{1,2,\ldots,k\} \quad \text{s.t.} \quad j-q>0 \quad \text{and} \quad Sim(\log_j, \log_{j-q}) \geq \theta\Big\}. \tag{5}$$

where the similarity threshold $\boldsymbol{\theta}$ is set to a **default value of 0.85**.

As shown in Compliance Proportion of Table 1, the results show that when the window size k is 32, the PSL exceeds 90% for most datasets, with some even exceeding 99%. This implies that, given identical length, in a log dataset, most log entries can be matched to a counterpart with a high similarity score, suggesting that length can be leveraged to group and align redundant characters. For datasets with low PSL, such as OpenStack and HDFS, our further analysis reveals that their variables are more diverse and change more frequently compared to other datasets. After adjusting the similarity threshold θ to 0.75, the PSL for both exceed 98.83%.

We investigate the impact of different window sizes k on the PSL. Clearly, as k decreases, the probability of finding similar logs also decreases, since a larger window provides more opportunities to find similar logs. Additionally, statistical data show that even with a window size of 2, the average PSL is 82.94% for TEXT logs and 91.27% for JSON logs, both of which are notably high.

Observation 3: More adjacent logs with identical lengths exhibit a stronger probability of high similarity scores. Logs are inherently a type of sequential data, because user and system operations tend to exhibit consistency over short periods. For instance, users may repeatedly access the same path or IP within a brief timeframe, many applications and systems maintain a specific state during execution, and multiple threads in concurrent systems may perform the same operations over a short duration. Hence, adjacent logs often exhibit considerable consistency in timestamps, user identifiers, system paths, IP addresses, and event types.

To further quantify, within each bucket of Observation 2, for each log, we search through its k most adjacent logs with identical lengths in sequential and reverse order to find a log whose similarity score with the current log surpasses the similarity threshold θ, continuing until such a log is identified. Here, we set the window size $k = 32$ and use the default value of θ. **Sequential order** refers to searching in the original sequence of the log file, whereas **reverse order** is the opposite and prioritizes searching from the most adjacent logs first. We record the average number of searches needed to find the satisfied logs in both cases, referred to as Seq and Rev, respectively. If no matching log entry is found within the k logs, the number of search attempts will be recorded as k.

As shown in Search Count of Table 1, on average, the sequential search required 6 attempts to find a similar log, while the reverse search only required 3 attempts, which implies that it is easier to find similar logs when starting the search from more adjacent logs.

Observation 4: Logs with identical lengths can achieve significantly high similarity scores. Observation 2 investigates the number of log entries that can be matched to a potentially suboptimal counterpart with a similarity score exceeding the default threshold θ (i.e., 85%). To further investigate the upper bound of the average similarity scores a log dataset can achieve, we traverse the $k = 32$ most adjacent log entries of the same length to find the optimal counterpart and identify the average **maximum similarity scores** (MSS) and the theoretical **average number** (AN) of reverse searches required to find the MSS.

As shown in Upper Bound of Table 1, the MSS for most datasets exceeds 90%, with an average of 94.7% across all datasets, which indicates that under best match, a substantial number of characters

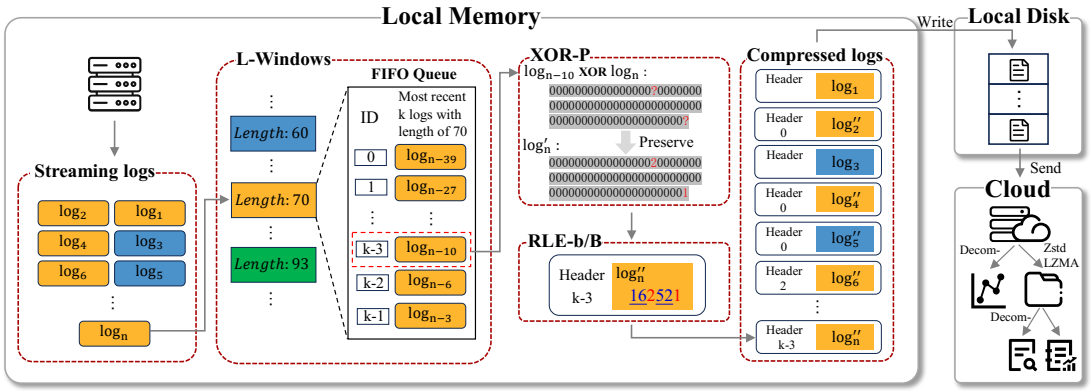


Figure 3: Overview of the LogLite framework (Logs of the same length have the same color).

in a log dataset are redundant, representing the maximum compressible redundancy. Additionally, the average number (AN) of reverse searches required is less than 16 in most cases.

Discussion. Observation 2 shows that logs of identical length typically follow the same pattern, with the vast majority matching at least one adjacent log within their length bucket. This insight allows LogLite to cache and group adjacent logs by length and eliminate redundancy using XOR and RLE (Section 3). Given the practical finiteness of log lengths (Observation 1), the memory required to cache k adjacent logs per length remains modest. Observation 3 reveals that searching in reverse chronological order significantly improves similarity detection efficiency. This informs LogLite's design to prioritize more adjacent logs, enhancing compression speed. While Observation 2 identifies the proportion of suboptimal logs suitable for redundancy elimination, Observation 4 quantifies the maximum of eliminable characters under best match, establishing an upper bound for potential compression gains.

Log length thus serves as an accessible and effective attribute for organizing logs. LogLite leverages this characteristic to group, align, and eliminate redundant content across log datasets, enabling both efficient and effective compression.

3 METHODOLOGY

3.1 Preliminaries

The design of the LogLite compression algorithm is inspired by the bitwise XOR operation on string data. Specifically, performing an XOR operation on two identical characters yields a null character (denoted as '\0'), whereas the XOR of two distinct characters results in a non-'\0' character. This pivotal characteristic enables a reversible encoding mechanism: performing an XOR operation between the resultant value and either of the original characters accurately reconstructs the other original character. Next, we introduce the detailed solution of LogLite.

3.2 Solution Framework

In the aforementioned analysis, we observe that log entries with identical length and belonging to the same pattern not only contain a large number of identical characters but also these identical characters are often aligned with each other in terms of positional index. The core idea of our solution leverages XOR to convert identical characters into '\0' and then employs run-length encoding (RLE) to achieve efficient and compact compression of these consecutive '\0' characters. Based on this concept, we design three components: L-Windows, XOR-P, and RLE-b/B.

As shown in Figure 3, a device generates log entries line-by-line, and as these entries are generated, they are immediately compressed line-by-line in local memory using LogLite, with the compressed logs being output line-by-line. Similar to general compression algorithms, LogLite caches previous logs using L-Windows, while organizing them by length. During compression, it identifies similar log entries in the cache to remove redundancy through XOR-P and RLE-b/B. Without predefined rules or sampling pre-training, LogLite maintains the plug-and-play and adaptability of general compression and leverages the specific characteristics of log data.

In the example of Figure 3, the length of the orange log entries is 70. Specifically, to compress the log entry $\log_n$ whose length is 70, LogLite first locates a corresponding queue whose key is 70 within L-Windows. The corresponding queue maintains the most adjacent k log entries, each matching the length of $\log_n$, from which it selects the entry significantly similar to $\log_n$, assumed to be $\log_{n-10}$. By performing XOR between $\log_{n-10}$ and $\log_n$, it obtains a string having the same length and containing a significant number of consecutive '\0' characters (represented by '0' in the figure). XOR-P then preserves the original characters of $\log_n$ to replace non-'\0' characters (represented by '?' in the figure, where $\log_n$ is ② in Figure 4(a)), resulting in $\log_n'$. The RLE-b/B component then performs compact run-length encoding on $\log_n'$, resulting in $\log_n''$. Additionally, to enable lossless decompression of the log entry, it stores auxiliary information, specifically the header information and the corresponding window ID. After compressing $\log_n$, the log entries in queue of L-Windows are updated: $\log_{n-39}$ is dequeued from the head, and $\log_n$ is inserted at the tail.

Once compressed in memory, the logs are written to local disk files in a streaming manner. After a predetermined time interval or when the file size reaches a specified threshold, the file is sent to the cloud for online analysis and archiving. As the logs are compressed in local memory, both writing resources and sending network bandwidth are significantly reduced. In high-real-time-demand scenarios, individual log entries can be directly sent to the cloud after compression, eliminating time spent writing to disk and waiting for the file to reach the sending condition.

3.3 Length-Windows

Length-Windows (L-Windows) is a hash table with integer keys representing log lengths and values as queues of log entries. Each element in the queue is a single log entry. The queue caches up to k log entries whose length corresponds to the key and follows a first-in-first-out (FIFO) policy.

The main idea of L-Windows is to partition log entries that have been compressed into separate queues based on their length and cache them in memory. When a new log entry needs to be compressed, it only needs to find a similar log entry among the k entries in the queue whose key is equal to the length of the new log entry, which facilitates subsequent XOR operations to enhance the number of consecutive '\0' characters obtained. If there is no key equal to the length of the new log entry that needs to be compressed, a new queue is created, and after the new log entry enters the queue, it is directly output without being compressed by XOR-P or RLE-b/B (e.g., $\log_1$ and $\log_3$ in Figure 3).

Based on Observation 1, the variation in log length from the same device or service is practically finite, particularly relative to the total size of the logs. Therefore, caching log entries into different queues based on their length is highly efficient, and the memory occupied by L-Windows is minimal. The overhead of obtaining the length of log entries and the enqueueing and dequeueing times are negligible. Theoretically, with k = 8, for a TEXT log dataset, L-Windows requires on average less than 0.3 MB of memory to cache log entries. Based on Observations 2 and 3, L-Windows' FIFO strategy ensures that adjacent log entries are cached, and by adjusting the window size k, the diversity of cached log entries can be controlled to adapt to the characteristics of different log sources.

3.4 XOR-Preserve

XOR-Preserve (XOR-P) consists of the following two phases: XOR-based Similarity Search and Preserving Original Characters.

XOR-based Similarity Search. To identify a log entry similar to the current log entry being compressed among the k log entries of the same length in the queue of L-Windows, XOR-P performs XOR operations with the log entry at the tail of the queue. Then, it calculates the proportion of '\0' characters in the XOR result, which is the similarity score of the two log entries. If this proportion exceeds a preset similarity threshold θ, the log entry is considered similar and selected; otherwise, it performs XOR operations with the next log in the queue from the tail to the head (i.e., in reverse order), until the threshold is reached. When the threshold is not even met, the log entry with the highest proportion of '\0' characters is deemed the most similar entry and is selected.

The design of XOR-based Similarity Search is based on the following two rationales: (1) Compared to other string similarity calculation methods, such as edit distance, XOR-based similarity score does not require expensive string matching. XOR operates at the bit level, is highly efficient on the CPU and is well-suited for SIMD, making it exceptionally fast. Additionally, an advantage is that the XOR result of the current compressed log entry and the selected similar log entry is obtained in the XOR-based Similarity Search phase, which can be cached to save the XOR operation in the Preserving Original Characters phase. (2) Based on Observation 3, adjacent log entries are often more likely to be similar. Therefore, starting the XOR operation from the most adjacent log entries (i.e., log entries at the tail) rather than from the earliest ones (i.e. log entries at the head) is more likely to meet the similarity threshold.

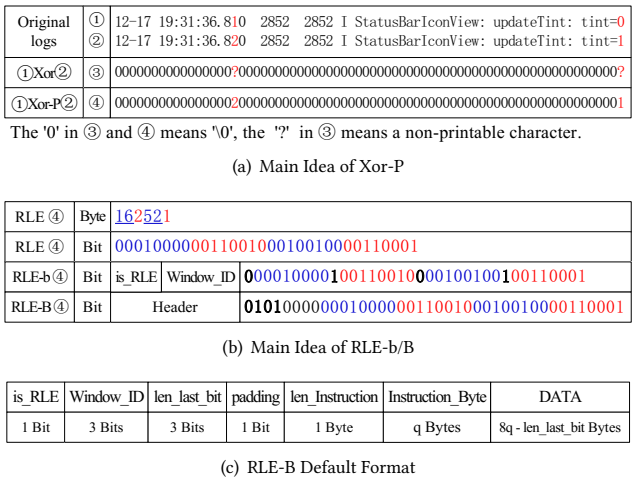


Figure 4: Components of the LogLite framework.

Preserving Original Characters. The current log entry being compressed is XOR-ed with the selected similar log entry to obtain an XOR result that contains many consecutive '\0' and some scattered non-'\0' characters. As shown in Figure 4(a), ② represents the current log entry being compressed and ① denotes the selected log entry from the corresponding queue. ② is XOR-ed with ①, resulting in the XOR result ③. For the result, XOR-P then retains all '\0' characters and replaces all non-'\0' characters with the characters at the corresponding indices of the current log entry, so the XOR-P result preserves some original characters. As shown in Figure 4(a), XOR-P transforms ③ into ④.

The design of Preserving Original Characters is based on the following two rationales: (1) Preserving Original Characters allows decompression without the need for XOR calculations, thus enhancing decompression speed. As shown in Figure 4(a), when compressing entry ②, we obtain ③ from ① XOR ②. To decompress ②, the calculation ② = ① XOR ③ must be performed once. However, for log entry ④, which preserves the original characters, decompression only requires replacing '\0' with the corresponding characters from ①, eliminating the need for XOR computation. (2) Compared to storing XOR results, Preserving Original Characters uses the same storage space but preserves more of the original characters of the log entry. Since these original characters are natural language, their distribution is more concentrated than that of characters obtained by XOR operations, which can improve compression ratios in subsequent general-purpose compression algorithms.

3.5 RLE-bit/Byte

After applying XOR-P, we obtain the XOR result ④ as shown in Figure 4(a). As the RLE ④ in Figure 4(b), the idea of Run-Length Encoding (RLE) is to record only the count of consecutive '\0' characters, while non-'\0' characters (i.e., the original characters from the current log) are fully retained. In Figure 4(b), it shows both the byte and bit representations of XOR-P result ④ after RLE.

In the RLE ④ of byte, the blue 16 and 52 are treated as integers and thereby require only one byte (8 bits) each for storage, while the red '2' and '1' are treated as characters and also require one byte (8 bits) each. Consequently, a log entry originally 70 bytes in size can be losslessly stored using only 4 bytes along with minimal additional metadata, thereby achieving a high compression ratio.

After being compressed and transmitted to a cloud server for log processing, logs may need to be parsed in real-time for anomaly detection and monitoring in certain applications, while in others, archiving collected logs suffices. Therefore, we design RLE-bit (RLE-b) and RLE-Byte (RLE-B), respectively. RLE-b outputs a bit stream, providing high compression ratios and fast processing, making it ideal for real-time log anomaly detection and monitoring. However, it is not compatible with further compression by general-purpose algorithms. In contrast, RLE-B outputs a byte stream, achieves competitive compression ratio and speed and supports general compression algorithms at the byte level for further compression, making it ideal for log archiving scenarios.

As shown in Figure 4(b), in the example of RLE-b ④, RLE-b only needs to record `is_RLE`, `Window_ID` and the streaming indicator bits (i.e., the bits are bolded) as additional metadata. These indicator bits specify whether the following 8 bits represent the count of consecutive '\0' characters or an original character from the log. This streaming design enables RLE-b to achieve high speeds.

As shown in Figure 4(b), in the example of RLE-B ④, RLE-B aligns data into a byte stream by grouping the indicator bits into bytes (i.e., `Instruction_Byte`). Since there are fewer than 8 bits, 4 zero bits are added as padding. The header contains supplementary details to facilitate lossless decompression. The standard RLE-B format is depicted in Figure 4(c). Here, `is_RLE` specifies if run-length encoding is utilized. `Window_ID` identifies the position within the L-Windows queue (with k typically set to 8). `len_last_bit` indicates the count of meaningful bits in the final byte of `Instruction_Byte`, enabling the decompression process to disregard padding zero bits. `Padding` consists of filler bits that ensure the initial four fields are byte-aligned. `len_Instruction` captures the byte length of `Instruction_Byte` (denoted as q), which inherently establishes that the byte length of `DATA` is $(8q - \texttt{len\_last\_bit})$, thereby allowing decompression to distinguish between `Instruction_Byte` and `DATA` segments.

3.6 Decompression

LogLite performs line-by-line streaming decompression, preserving the original log sequence order. Similar to most general compression, LogLite decompresses a log entry based on previously decompressed log entries.

To decompress the compressed logs shown in Figure 3, since $\log_1$ is the first log with a length of 70 that has not undergone XOR-P and RLE-b/B, LogLite creates a new queue in L-Windows according to the length of $\log_1$, directly outputs $\log_1$ and caches it in the new queue. $\log_2''$ is decoded using run-length decoding to obtain $\log_2'$. Then, using $Window\_ID$ (i.e., 0) and the length of $\log_2'$ (i.e., 70), LogLite finds $\log_1$ in the corresponding queue. The '\0' characters in $\log_2'$ are replaced by the characters at the same indices in $\log_1$, reconstructing the original log $\log_2$. $\log_2$ is output and also cached in the queue of length 70. $\log_3$ is a log that has not undergone XOR-P and RLE-b/B, so it is directly output and cached in the new queue of length 60. The remaining logs repeat the above process to achieve efficient streaming decompression.

3.7 Complexity

Given n log entries with an average length of AL, and the number of different lengths is NDL. During the compression, it is necessary to maintain an L-Windows, which can have a maximum of NDL queues, each queue containing a maximum of k logs having an average length of AL, where k is a constant. Therefore, the space complexity is $O(NDL \times k \times AL)$.

When compressing a log, at most k logs with an average length of AL need to be XORed, resulting in a time complexity of $O(k \times AL)$. The time complexity of the encoding of an RLE is $O(AL)$. The time complexity for compressing a log is $O((k+1) \times AL)$. Thus, the time complexity for compressing n logs is $O(n \times (k+1) \times AL)$.

In the worst-case scenario, where all log lengths are different, i.e., $NDL = n$, and $AL = \frac{n(n+1)}{2n} = O(n)$, the space complexity becomes $O(kn^2)$, and the time complexity becomes $O(kn^2)$. However, the characterization study in Table 1 indicates that NDL and AL are much smaller than n, approximated to constants. Therefore, in practical applications, the space complexity is very small, and the time complexity is equivalent to $O(n)$. Decompression is the inverse process of compression and shares the same complexity.

4 EXPERIMENTAL EVALUATION

To validate our observations and the effectiveness of the proposed LogLite solution, we conduct comprehensive experiments on 21 public and widely used log datasets, comparing it against SOTA general-purpose and log-specific compression baselines. Specifically, we focus on the following aspects: (1) The compression ratio, compression speed, and decompression speed of LogLite for both line-by-line log compression and file compression. (2) The throughput of LogLite in real-time compression scenarios. (3) The contribution of each component to the effectiveness of LogLite. (4) The impact of different configurations on LogLite's performance.

4.1 Experimental Setting

4.1.1 ***Datasets.*** We use two log datasets, detailed in Table 1.

(1) Unstructured log datasets: We include all 16 public unstructured TEXT log datasets from Loghub [59]. HDFS, Had, Spa, Zoo, and Ope are from Distributed Systems; BGL, HPC, and Thu are from Super Computers; Win, Lin, and Mac from Operating Systems; And and Hea are from Mobile Systems; Apa and SSH are from Server Applications; and Pro is from Standalone Software. The total size exceeds 60 GB.

(2) Semi-structured log datasets: We utilize all 5 public semi-structured JSON log datasets from μSlope [47], which are generated by running the HiBench [19] and YCSB [33] benchmarks. The total size exceeds 80 GB.

4.1.2 ***Environments.*** Our code is implemented in C++ and compiled with g++ 9.4.0 using the -Ofast optimization flag. All experiments are conducted on a machine equipped with an Intel(R) Xeon(R) Platinum 8336C CPU @ 2.30GHz, 125GB of main memory, and running Linux kernel version 5.15.0-124-generic.

Table 2: Line-by-line compression performance.(Speed is measured in MB/s)

		Datasets	TEXT																JSON				
			Apa	Lin	Zoo	Hea	HPC	And	Had	BGL	Mac	Ope	Pro	Spa	SSH	Thu	Win	HDFS	Mon	Coc	Ela	Spark	Pos
Compression Ratio	Specific	PBC	0.185	0.240	0.118	0.284	0.298	0.355	0.156	0.331	0.277	0.190	0.174	0.284	0.193	0.711	0.219	0.320	0.607	0.353	0.103	0.565	0.083
		PBC-F	0.106	0.159	0.075	0.174	0.201	0.251	0.105	0.168	0.197	**0.126**	**0.113**	0.158	0.119	0.382	0.114	**0.144**	0.235	0.225	0.080	0.398	**0.045**
		LogLite-B	0.112	0.175	0.071	0.174	0.154	0.250	0.103	0.127	0.160	0.166	0.163	0.105	0.108	0.116	0.105	0.232	0.024	0.119	0.028	0.096	0.072
		LogLite-b	**0.088**	**0.157**	**0.056**	**0.153**	**0.131**	**0.234**	**0.086**	**0.114**	**0.147**	0.159	0.144	**0.081**	**0.092**	**0.103**	**0.098**	0.218	**0.019**	**0.112**	**0.023**	**0.095**	0.067
	General	FSST	0.381	**0.441**	0.386	**0.414**	0.418	0.609	0.425	**0.346**	0.599	0.437	0.313	0.479	0.322	0.467	0.395	**0.326**	0.457	0.792	0.760	0.638	0.320
		LZ4-d	**0.350**	0.444	**0.250**	0.418	**0.353**	**0.569**	**0.305**	0.382	**0.414**	0.265	**0.310**	**0.358**	**0.302**	0.413	0.251	0.375	**0.163**	0.173	0.241	0.245	0.154
		Zstd-d	0.428	0.491	0.283	0.477	0.405	0.571	0.314	0.365	0.425	**0.236**	0.354	0.399	0.366	**0.408**	**0.243**	0.344	0.169	**0.116**	**0.195**	**0.179**	**0.136**
Compression Speed	Specific	PBC	53.6	68.5	129.0	60.7	65.3	52.3	101.0	32.9	94.1	181.9	32.8	52.5	57.0	121.3	105.7	97.2	398.1	225.7	433.4	194.1	167.2
		PBC-F	51.5	64.1	122.2	56.2	62.1	48.8	95.0	31.5	117.1	159.5	32.0	49.7	54.4	94.1	99.2	87.9	237.3	202.4	370.5	135.8	158.5
		LogLite-B	100.7	77.8	151.0	79.8	83.2	83.7	118.9	135.5	104.3	138.3	93.1	107.1	110.7	135.3	160.4	91.8	338.2	187.5	457.3	316.4	285.6
		LogLite-b	**219.1**	**132.5**	**333.1**	**137.3**	**145.8**	**122.6**	**233.3**	**203.7**	**167.7**	**183.6**	**148.7**	**226.3**	**213.8**	**217.4**	**217.9**	**127.2**	**650.1**	**260.1**	**737.8**	**321.7**	**406.8**
	General	FSST	**432.2**	**384.2**	**351.1**	**329.3**	**378.1**	**239.6**	**268.6**	**290.1**	**256.5**	**275.1**	**503.6**	**247.1**	**457.8**	**250.4**	**274.7**	**316.8**	**261.1**	**182.8**	**197.8**	**200.6**	**291.9**
		LZ4-d	29.6	33.1	50.2	33.2	35.4	38.0	56.8	46.8	46.2	75.7	33.6	40.5	23.5	45.9	63.9	47.2	99.0	207.2	179.4	177.4	109.6
		Zstd-d	8.4	8.6	12.8	8.5	9.1	10.7	15.6	12.3	13.3	25.3	10.9	10.6	10.3	12.3	17.2	12.6	29.4	112.7	84.7	101.6	34.7
Decompression Speed	Specific	PBC	**2580.8**	**2396.2**	**2578.4**	**2006.9**	**1947.3**	**1967.5**	**2900.1**	**1525.1**	**1532.1**	**1975.6**	**1949.5**	**2066.0**	**2676.8**	**1094.5**	**1211.6**	**2161.8**	**2640.7**	**2311.6**	**3405.3**	**4041.5**	**2543.9**
		PBC-F	702.4	628.8	462.1	236.4	571.3	650.8	1312.1	637.1	544.3	1012.9	650.0	1540.4	783.8	477.1	704.7	838.7	1535.5	2350.4	2871.7	1833.9	1430.2
		LogLite-B	496.3	379.3	640.6	341.8	409.4	321.0	530.6	571.3	405.0	468.3	412.9	522.4	506.1	561.7	732.8	369.9	1307.5	613.9	1251.3	755.1	877.9
		LogLite-b	493.2	371.7	680.6	356.3	391.2	342.6	539.7	586.9	412.1	485.1	413.4	544.7	514.3	581.0	573.0	378.7	1350.8	645.3	1440.4	765.0	822.8
	General	FSST	401.9	376.9	490.6	397.6	399.9	347.8	496.0	525.2	362.5	559.5	532.9	398.0	516.3	431.6	552.3	573.2	553.3	362.9	357.2	456.2	764.8
		LZ4-d	**635.5**	**608.0**	**978.2**	**606.8**	**665.5**	**665.3**	**1108.2**	**803.0**	**913.0**	**2021.5**	**758.2**	**740.3**	**818.2**	**801.2**	**1332.9**	**955.1**	**2635.7**	**3032.4**	**2668.5**	**3036.7**	**2845.6**
		Zstd-d	361.7	367.9	609.2	376.7	424.6	314.6	558.1	442.9	429.9	919.5	453.3	433.1	450.2	413.4	745.6	514.7	1274.3	1603.0	1481.0	1307.4	1444.5

4.1.3 ***Baselines.*** We compare our methods with the following well-known lossless compression algorithms, using default compression levels and parameters for all baselines.

- LZ (Lempel–Ziv) methods: The LZ family is the most widely used general-purpose compression technique, adopted by numerous database systems. Based on benchmark results [23, 27], we select the following three representative algorithms as baselines:
  - **LZ4** [9]: A lightweight compression algorithm with the fastest compression and decompression speed in the LZ family, often used in real-time scenarios.
  - **LZMA** [26]: Optimized for maximum compression ratio at the cost of speed, commonly used in archival scenarios.
  - **Zstd** [10]: Achieving Pareto optimality between speed and compression ratio, it is considered one of the most suitable compression methods for modern database systems [51].
- **FSST** [6]: A state-of-the-art lightweight compression method that supports line-by-line string compression.
- **PBC** [56]: A compression algorithm optimized for machine-generated data (e.g., KV, Logs, JSON), balancing general-purpose and domain-specific methods for high compression ratio and decompression speed.
  - For line-by-line compression, PBC can integrate FSST for improved compression ratio, denoted as **PBC-F**.
  - For file compression, PBC can further integrate Zstd or LZMA, denoted as **PBC-Z** and **PBC-L**, respectively.
- **LogShrink** [30]: A log-specific compression algorithm for unstructured logs, achieving the highest compression ratio, which incorporates LZMA for further compression.
- **LogReducer** [54]: A log-specific compression algorithm for unstructured logs, achieving Pareto optimality between compression ratio and speed, which also incorporates LZMA for further compression.
- **LogGrep** [49]: A state-of-the-art log-specific compression algorithm for unstructured logs that supports search without full decompression. It can further integrate Zstd or LZMA, denoted as **LogGrep-Z** and **LogGrep-L**, respectively.
- **CLP** [41]: A state-of-the-art compression algorithm relying on pre-defined rules for unstructured and semi-structured logs, which supports search without full decompression and incorporates Zstd for further compression.
- **μSlope** [47]: A state-of-the-art compression algorithm for semi-structured logs, supporting search without full decompression, which incorporates Zstd for further compression.
- The proposed LogLite and its variations (Open-sourced at [1]):
  - **LogLite-b**: A lightweight compression method proposed by us, outputting **bitstreams** and offering the best compression ratio and speed for line-by-line compression.
  - **LogLite-B**: A lightweight compression method proposed by us, outputting **bytestreams**, which can be further compressed by general-purpose compression algorithms.
  - **LogLite-BZ** and **LogLite-BL**: Extensions of LogLite-B with further compression using Zstd and LZMA, respectively.

4.2 Effectiveness and Efficiency

Based on our study of the entire life cycle of logs, we compare LogLite with other baselines using the following most important metrics: compression ratio, compression speed, and decompression speed. Specifically, the compression ratio is defined as follows, where a smaller compression ratio indicates better compression:

$$\text{Compression Ratio} = \frac{\textit{Compressed Data Size}}{\textit{Original Data Size}}$$

In all the following experiments, unless otherwise specified, LogLite is configured with a default window size of $k = 8$ and a similarity threshold of $\theta = 0.85$. In Table 2 and 3, bold values indicate the best performance within each compressor category (log-specific or general-purpose), while bold underlined values represent the overall best across all compressors.

4.2.1 ***Line-by-Line Compression.*** In many applications, log producers must transmit logs to other machines or cloud servers with high real-time performance for timely analysis and monitoring. To ensure this, we evaluate all baseline compression algorithms supporting line-by-line compression.

According to [6], the online compression methods in the LZ family struggle to compress single log entries. In most cases, their compression ratio is close to 1, or even exceeds 1, meaning no benefit can be gained from these compression methods. However, LZ4 and Zstd offer a special compression mode to support compression of small files. They first train a dictionary offline on sampled data, and then use this trained dictionary during online compression, allowing for line-by-line compression of logs. We evaluate LZ4 and Zstd with a pre-trained dictionary (trained by Zstd) at the default compression level, denoted **LZ4-d** and **Zstd-d**.

Notably, all baselines, including PBC, PBC-F, FSST, LZ4-d, and Zstd-d, require first collecting a sufficient amount of logs and offline training a dictionary on the sampled data to support line-by-line compression. In our experiment, they can sample sufficiently on each dataset. However, in practice, if the sampled data do not fully reflect the characteristics of the logs or if the characteristics of the logs change, the compression performance will degrade, requiring additional resources and time for resampling and retraining the dictionary. However, our proposed LogLite is a plug-and-play method that requires no offline training process, and can remain effective even if the characteristics of the logs change.

As shown in Table 2, in terms of compression ratio, LogLite-b outperforms all specific-purpose and general-purpose compression baselines. Compared to the second-best performer, PBC-F, LogLite-b achieves an average improvement of 20.4% in compression ratio on the TEXT dataset and 67.8% on the JSON dataset. Compared to LZ4-d, the best general compressor, LogLite-b achieves an average improvement of 64.7% in compression ratio across all datasets.

A general trend indicates that higher PSL values typically correspond to a better compression ratio for LogLite (e.g., Apa achieving a better result than Lin). In practice, the actual compression ratio is also influenced by the specific distribution of redundant characters and the encoding strategy. For example, Spa outperforms Had because the redundant characters in Spa are more contiguous, which reduces the number of RLE codes required. For datasets with a low PSL, the result is suboptimal but competitive (e.g., Ope and HDFS).

In terms of compression speed, LogLite-b demonstrates strong competitiveness, achieving the best performance on the JSON dataset and the second-best performance on the TEXT dataset, only trailing FSST having a poor compression ratio. Compared to FSST, LogLite-b achieves an average compression speed improvement of 109.5% on the JSON dataset. Compared to PBC-F, LogLite-b is on average 130.04% faster across all datasets. For decompression speed, although LogLite is on average 41.7% slower than PBC-F, it achieves competitive performance with FSST. This is because LogLite requires more computation for decompression than PBC-F's straightforward dictionary substitution.

4.2.2 ***File Compression.*** Before logs are archived to disk, they are typically collected and compressed as a large file, significantly reducing storage costs. This approach is particularly well-suited for the LZ family, as these algorithms excel at finding redundancy over large blocks of data. This is also a common application scenario for many log-purpose compression baselines. Although LogLite first compresses log entries on line-by-line, thanks to our design, LogLite exhibits a degree of orthogonality with the LZ family. As a result, algorithms like Zstd and LZMA can still further reduce redundancies effectively. Specifically, Zstd and LZMA can perform additional compression on the data files produced by LogLite-B's line-by-line compression (i.e., LogLite-BZ and LogLite-BL). Most log-purpose compression baselines support only one type of log format: either unstructured (TEXT) or semi-structured (JSON), but our method is applicable to both common log formats.

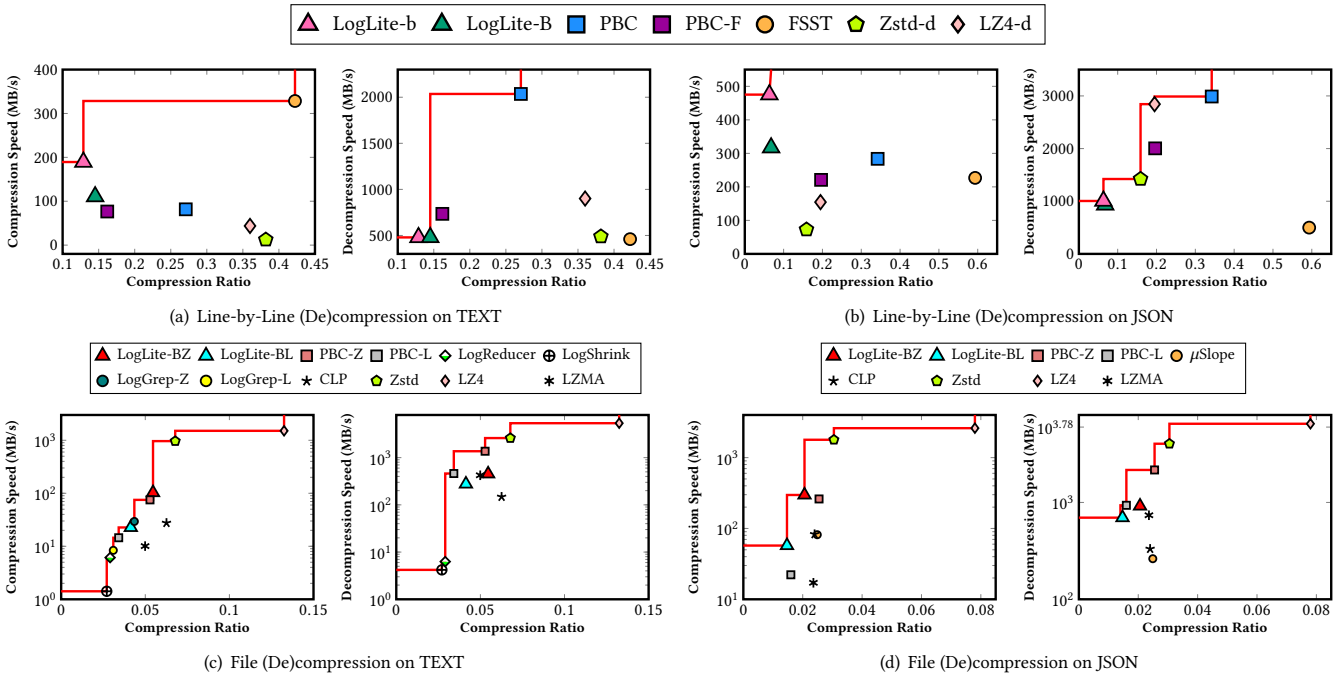
Table 3: File compression performance for semi-structured (JSON) log. (Speed is measured in MB/s)

Datasets			Mon	Coc	Ela	Spark	Pos
Compression Ratio	Specific	CLP	0.0091	0.0497	0.0073	0.0209	0.0328
		μSlope	0.0059	0.0412	0.0067	0.0199	0.0510
		PBC-Z	0.0253	0.0445	0.0074	0.0265	0.0238
		PBC-L	0.0152	**0.0258**	0.0042	0.0185	**0.0161**
		LogLite-BZ	0.0045	0.0398	0.0037	0.0240	0.0308
		LogLite-BL	**0.0026**	0.0284	**0.0018**	**0.0179**	0.0225
	General	LZ4	0.0388	0.0907	0.0325	0.1601	0.0681
		Zstd	0.0140	0.0485	**0.0107**	0.0359	0.0434
		LZMA	**0.0138**	**0.0343**	0.0114	**0.0241**	**0.0343**
Compression Speed	Specific	CLP	93.15	45.57	113.94	104.47	57.54
		μSlope	89.37	54.68	103.84	91.28	68.79
		PBC-Z	**349.48**	**224.35**	405.34	170.46	155.30
		PBC-L	24.17	10.17	44.78	12.61	19.19
		LogLite-BZ	328.90	167.84	**444.66**	**288.73**	**254.90**
		LogLite-BL	106.01	12.33	123.70	23.17	21.53
	General	LZ4	**4386.99**	**1754.00**	2997.00	997.00	**2890.00**
		Zstd	2579.15	859.00	**3095.00**	**1131.00**	1272.00
		LZMA	22.55	12.60	25.80	9.26	15.30
Decompression Speed	Specific	CLP	160.08	78.22	666.57	368.10	384.76
		μSlope	185.45	216.78	491.87	290.10	127.96
		PBC-Z	**1784.62**	**1892.38**	**2865.79**	**2268.43**	**2014.68**
		PBC-L	961.79	454.97	1586.72	828.58	839.34
		LogLite-BZ	1263.38	571.33	1215.86	718.83	822.42
		LogLite-BL	1157.86	297.41	1136.19	434.61	455.88
	General	LZ4	**5680.40**	**6722.00**	**7046.00**	**5046.00**	**7968.00**
		Zstd	4581.71	2545.00	6605.00	3427.00	3031.00
		LZMA	987.50	480.00	831.00	775.00	615.00

In the JSON datasets, as shown in Table 3, LogLite-BL achieves the best compression ratio, outperforming the second-best method, PBC-L, by 8.1% on average, and the best general-purpose compressor, LZMA, by 37.7% on average. Additionally, LogLite-BZ surpasses PBC-Z and μSlope by an average of 19.2% and 17.3%, respectively. In terms of compression speed, LogLite-BZ achieves the best results among specific-purpose methods, outperforming PBC-Z by an average of 14.2%. Similarly, LogLite-BL outpaces PBC-L and LZMA by an average of 1.6× and 2.4×, respectively. For decompression speed, although LogLite-L is on average 25.5% slower than PBC-L, it demonstrates competitive overall performance.

In the TEXT datasets, LogLite-BL's compression speed outperforms all specific compressors using LZMA for further compression, achieving a 2.7× improvement compared to LogReducer, better compression ratios than general compressors and competitive decompression speed. The overall comparison is demonstrated in Section 4.2.3, with detailed tables found in [46].

4.2.3 ***Pareto-Optimal Compression.*** To better illustrate the position of LogLite in compressing TEXT and JSON logs, we plot 8



(a) Line-by-Line (De)compression on TEXT

(b) Line-by-Line (De)compression on JSON

(c) File (De)compression on TEXT

(d) File (De)compression on JSON

Figure 5: Performance visualization of compared baselines with Pareto front (red lines).

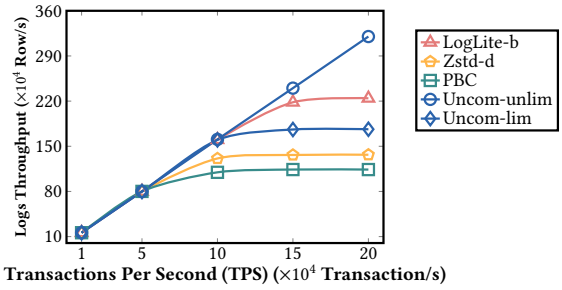


Figure 6: Throughput of logs processing.

figures based on the aforementioned experimental results over all our TEXT and JSON log datasets. These figures respectively depict the positions of compression ratio versus compression speed and compression ratio versus decompression speed for both line-by-line and file compression on TEXT and JSON logs. The top-left corner of each graph indicates better performance, and the red line represents the Pareto frontier. Methods positioned on this red line are Pareto-optimal. To distinguish LogLite from other baselines, we depict it as an upward triangle in the plots. As shown in Figure 5, LogLite achieves Pareto-Optimal in seven out of the eight figures. Notably, for line-by-line compression on JSON logs, LogLite achieves the best results in both compression ratio and compression speed.

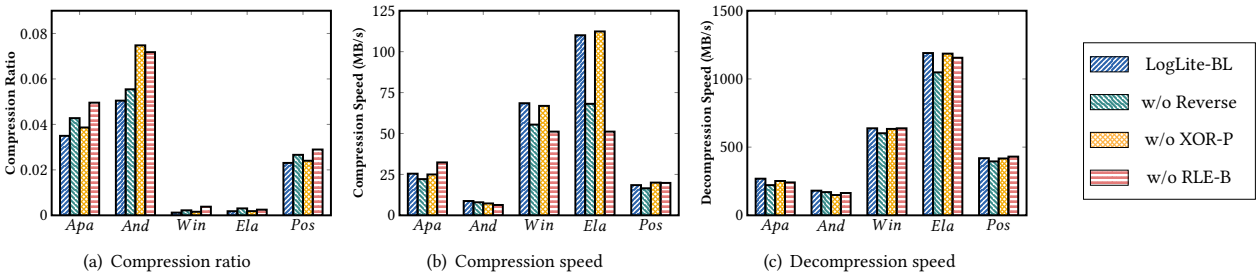
4.3 Case Study

A key application of LogLite is compressing logs line-by-line immediately after they are generated, reducing the resources needed for writing and sending logs. We integrate LogLite-b, PBC, and Zstd-d into PostgreSQL with configuration of `log_statement=all` and `log_min_messages=debug5` to enable the full logging strategy. During stress testing, the database generates a large volume of logs. These logs are compressed in real-time by a dedicated thread running the compression algorithm before being written to disk. We measure both the compression ratios and the throughput of logs from generation to their compressed output on disk.

The overall PostgreSQL log data compression ratios for LogLite-b, PBC, and Zstd-d are **0.1258**, **0.5802**, and **0.3255**, respectively. Notably, LogLite-b is plug-and-play, whereas PBC and Zstd-d require offline sampling and training of their respective dictionaries using pre-existing log data. Due to the dynamic nature of logs and the dictionary of PBC being trained on historical log data, PBC struggles to fully adapt to the log characteristics during compression, resulting in poor compression ratios in this scenario.

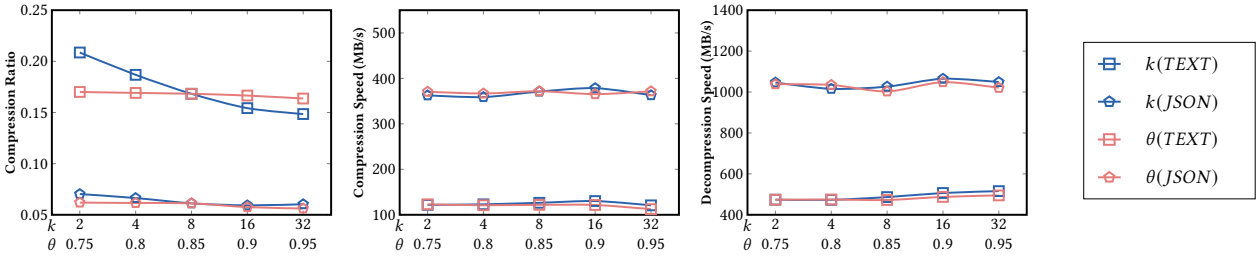
As shown in Figure 6, when the TPS is low, indicating a low log generation rate, all compression algorithms can handle the logs in real time, with throughput increasing as the TPS rises. However, when the system has to process logs generated by 10×10^4 transactions/s, both PBC and Zstd-d reach their performance bottlenecks, causing their throughput to plateau. In contrast, our method continues to work effectively. LogLite-b achieves a maximum throughput of 2.24 million rows per second, outperforming PBC and Zstd-d by 96.98% and 63.93%, respectively.

Theoretically, as long as disk write speed exceeds log generation rate, uncompressed log throughput will increase with log generation rate (i.e., Uncompressed-unlimited). However, in real-world environments, hardware I/O limitations or resource constraints cause throughput to hit a bottleneck. In the Uncompressed-limited case, we cap the write speed at 250 MB/s, corresponding to typical HDD speeds. When processing logs from 150,000 TPS, disk write speed becomes the bottleneck. However, LogLite-b compresses logs, requiring less than 50 MB/s of I/O, resulting in a 23.99% throughput improvement over Uncompressed-limited.



(a) Compression ratio (b) Compression speed (c) Decompression speed

Figure 7: Ablation study on 5 representative datasets.



(a) Compression ratio (b) Compression speed (c) Decompression speed

Figure 8: Effect of different window size k and similarity threshold θ.

4.4 Ablation Study

To evaluate each component's effectiveness in LogLite's design, we conduct an ablation study on LogLite-BL and its variants: without reverse lookup in L-Windows (w/o Reverse), without preserving original characters in XOR-P (w/o XOR-P), and without byte alignment in RLE-B (w/o RLE-B). We select 5 representative datasets, including 3 TEXT datasets (Apache, Android, Windows) and 2 JSON datasets (Elasticsearch, PostgreSQL). Compression ratio, compression speed, and decompression speed are presented in Figure 7.

Compared to w/o Reverse, the full LogLite-BL achieves an average improvement of 14.23% in compression ratio. This is because, although w/o Reverse identifies logs that exceed the default similarity threshold θ (i.e., 0.85), the logs found without reverse lookup have lower similarity scores in general. In terms of compression speed, the full LogLite-BL shows an average improvement of 35.83%, as reverse lookup reduces the number of searches needed to find similar logs. For decompression speed, the full LogLite-BL demonstrates an average improvement of 10.62%, as the inferior compression ratio in w/o Reverse requires more iterations to decode RLE and XOR.

Compared to w/o XOR-P, the full LogLite-BL achieves an average improvement of 20.92% in compression ratio. This is because w/o XOR-P applies XOR to all original characters, disrupting the character distribution in logs and making it harder for LZMA to achieve further compression. While preserving original characters requires additional operations, the use of efficient SIMD implementations limits the average slowdown in compression speed to just 0.13%. Furthermore, the full LogLite-BL achieves an average improvement of 2.23% in decompression speed, as its XOR-P design eliminates the need for XOR computation during decompression.

Compared to w/o RLE-B, the full LogLite-BL achieves an average improvement of 28.75% in compression ratio, along with average improvements of 11.14% in compression speed and 2.55% in decompression speed, which is because LZMA achieves better compression on byte-aligned streams compared to bit-aligned streams.

4.5 The Impact of Different Parameter Settings

This subsection examines the impact of key parameters (window size k and similarity threshold θ) on LogLite's performance. We evaluate LogLite-B with k values of 2, 4, 8, 16, and 32, and θ values of 0.75, 0.80, 0.85, 0.90, and 0.95. Figure 8 shows the average changes in compression ratio, compression speed, and decompression speed in the above 3 TEXT and 2 JSON datasets.

Increasing k improves the compression ratio, with a 28.81% gain for TEXT datasets and 16.58% for JSON datasets by finding more similar logs in larger windows requiring more memory. A stricter threshold θ yields a slight improvement: 3.82% for TEXT datasets and 9.37% for JSON datasets on average. The improvement varies by dataset, for example, the compression ratio increases by 10.89% on Windows when θ rises from 0.75 to 0.95, but only 0.52% on Elasticsearch. Compression and decompression speeds are minimally affected by changes in k and θ. While larger k or θ could increase search time and reduce compression speed, their improvement in compression ratio reduces encoding time, keeping the speed stable. Decompression speed tends to increase with larger k and θ, as the improved compression ratio reduces decoding time. In summary, larger k and θ typically improve compression ratio with minimal speed impact but require more memory and CPU.

5 RELATED WORK

5.1 General-Purpose Compression

General-purpose compression algorithms are the most widely used, as they can effectively compress data without relying on specific data distributions or requiring prior knowledge about the data.

Dictionary- and Statistics-based Compression. Compression algorithms like LZ77 [60], LZ78 [61], and their various derivatives, such as LZ4 [9], LZMA [26], Zstd [10], DEFLATE [17, 25, 52], Snappy [22], and Brotli [3], combine a dictionary matching phase with a fast entropy coding phase. The primary idea is to treat data as a byte stream within a sliding window, identify similar byte sequences in this window, and replace repeated bytes with shorter dictionary indices. This is followed by further compression using entropy coding techniques such as Huffman coding [20], Arithmetic coding [39], and Asymmetric Numeral Systems (ANS) [12], which rely on statistical analysis of the data to assign shorter codes to more frequent symbols, thereby improving compression performance. FSST [6] encodes strings using a symbol table, which maps 1-8 byte sequences onto single-byte codes.

General compression algorithms vary in speed and compression ratio by adopting different strategies. LZ4 [9] avoids entropy coding and simplifies dictionary matching, achieving exceptional speed, making it a top choice for general compression [6, 27] and widely used in Hadoop [45]. LZMA [26] uses a large window, complex context models, and near-optimal arithmetic coding to maximize compression ratio at the cost of speed, making it suitable for cold data archiving. Zstd [10] balances speed and ratio, reaching the Pareto frontier and being adopted in AWS Redshift [44] and RocksDB [21], and supports offline-trained dictionaries for small data.

5.2 Log Data Compression

Compared to general compression, log data compression takes full advantage of the semantic information of logs, particularly the structured components and the significant similarities within log records, to achieve better compression performance. However, most of these methods require accurate and appropriate prior knowledge of the log data, provided either manually or automatically.

Non-Parsing-based Compression. LogArchive [8] adaptively assigns log entries to different buckets using a similarity function to minimize heterogeneity within each bucket and compresses these buckets separately in parallel. Cowic [32] employs a semi-static dictionary model and Huffman coding to split log entries into multiple columns, building different models for each column for compression. MLC [16] compression technology identifies and organizes similar log records using Content-Defined Chunking (CDC) and a Jaccard distance-based bucket allocation strategy. It then encodes new log entries using delta compression and ultimately applies general-purpose compression for final compression.

Parsing-based Compression. Since logs typically follow certain patterns or templates, to fully leverage the semantic information of the fields in the logs and achieve higher compression ratios, parsing-based compression methods parse logs and extract patterns or templates using log parsers or regular expressions, albeit at the cost of efficiency, and stores shorter representations instead of the original and repeated patterns or templates. LogZip [35] parses templates from log samples and extracts all templates from raw log files through an iterative matching and extraction process, compressing logs into template IDs and variables. LogReducer [50] builds on LogZip by analyzing and optimizing it and applies specific encoding techniques for timestamps and numerical variables to achieve better compression ratios. In order to accelerate the template matching, LogReducer segments templates into tokens using delimiters and groups templates with identical token counts during parse tree construction, enabling rapid filtering of candidate templates based on an incoming log entry's token count. Building on LogReducer's parser, LogGrep [49] divides logs into subcomponents by identifying common patterns, even splitting variables for finer granularity, and uses tables to map and reconstruct original log messages for more efficient compression and search. LogGrep also leverages the observation that variable parts of the same runtime pattern often exhibit similar lengths to filter and accelerate searches on compressed logs. LogShrink [30] also adopts the parser of LogReducer and leverages a commonality and variability analyzer to identify repetitive patterns and variations in log data, and store similar data in a columnar format by separating fields, leading to high compression ratios. CLP [41] splits logs into three components (i.e. timestamps, static text, and variable values) and employs a dictionary to store and index repeated information, thus compressing logs and optimizing search efficiency. Unlike these methods for unstructured TEXT logs, μSlope [47] targets semi-structured JSON logs by merging parse trees to store schema structures, grouping logs with the same schema into structured tables for columnar storage. It uses dictionaries and type-specific encoding to reduce data redundancy, achieving efficient compression and fast searches. To save more storage resources during archiving, these compression methods often apply further compression using general-purpose compression like Zstd or LZMA.

5.3 Other Compression

For the compression of floating-point sequences, since a floating-point number is typically fixed at 32 or 64 bits and many floating-point sequences generated by sensors are nearly continuous, Gorilla [40], Chimp [31], and Elf [28] use XOR with the previous value to achieve lightweight compression. To bridge the gap between general-purpose compression and specific-purpose compression, PBC [56] supports data with patterns such as machine-generated KV data, logs, JSON, URLs, and UUIDs, etc. PBC extracts potential patterns from the data through clustering, thereby reducing redundancy in common subsequences, and achieves a high compression ratio, fast decompression speed, and competitive compression speed without any manually defined knowledge.

6 CONCLUSION

We present LogLite, a lightweight plug-and-play streaming lossless compression algorithm for TEXT and JSON logs. Based on our in-depth characterization study, it exploits the strong correlation between log length and similarity. Our evaluation demonstrates that LogLite achieves Pareto-optimal performance in most scenarios.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (2022YFB3103700), CCF-AFSG Research Fund (RF20220217), ARC Future Fellowship (FT210100303), ARC Discovery Project (DP230101445), Guangdong S&T Program under Grant 2024B01010 10002, National Natural Science Foundation of China (No.U2436208, No.62372129), and Project of Guangdong Key Laboratory of Industrial Control System Security (2024B1212020010).

REFERENCES

[1] 2025. LogLite. https://github.com/benzhaotang/LogLite

[2] Vaibhav Agrawal, Devanjal Kotia, Kamelia Moshirian, and Mihui Kim. 2018. Log-based cloud monitoring system for OpenStack. In *2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 276–281.

[3] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2018. Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems (TOIS)* 37, 1 (2018), 1–30.

[4] aliyun. 2018. aliyunIoT. https://developer.aliyun.com/article/637406

[5] Anunay Amar and Peter C Rigby. 2019. 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, 140–151.

[6] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

[7] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. 2021. Logstore: A cloud-native and multi-tenant log database. In *Proceedings of the 2021 International Conference on Management of Data*. 2464–2476.

[8] Robert Christensen and Feifei Li. 2013. Adaptive log compression for massive log data.. In *SIGMOD Conference*. 1283–1284.

[9] Yann Collet. 2011. LZ4: Fast Compression Algorithm. https://github.com/lz4/lz4

[10] Yann Collet and Murray Kucherawy. 2018. *Zstandard Compression and the application/zstd Media Type*. Technical Report.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[12] Jarek Duda, Khalid Tahboub, Neeraj J Gadgil, and Edward J Delp. 2015. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *2015 Picture Coding Symposium (PCS)*. IEEE, 65–69.

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

[14] Yao-Chung Fan, Yu-Chi Chen, Kuan-Chieh Tung, Kuo-Chen Wu, and Arbee LP Chen. 2015. A framework for enabling user preference profiling through wi-fi logs. *IEEE Transactions on Knowledge and Data Engineering* 28, 3 (2015), 592–603.

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

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

[17] Jean-loup Gailly and Mark Adler. 1992. GNU gzip. *GNU Operating System* (1992).

[18] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2018. 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*. 60–70.

[19] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*. IEEE, 41–51.

[20] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[21] Facebook Inc. 2012. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb

[22] Google Inc. 2011. Snappy: A Fast Compressor. https://github.com/google/snappy

[23] Insanity Industries. 2021. Pareto-optimal compression. https://insanity.industries/post/pareto-optimal-compression/

[24] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 61–71.

[25] Phil Katz. 1989. ZIP: A File Compression Standard. https://www.info-zip.org/

[26] Abraham Lempel and Jacob Ziv. 2001. LZMA Algorithm. http://www.7-zip.org/

[27] Roman Leshchinskiy. 2018. LZBench: Compression Benchmarking Tool. https://github.com/lemire/LZBench

[28] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-based lossless floating-point compression. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1763–1776.

[29] Xiaoyun Li, Pengfei Chen, Linxiao Jing, Zilong He, and Guangba Yu. 2022. Swisslog: Robust anomaly detection and localization for interleaved unstructured logs. *IEEE Transactions on Dependable and Secure Computing* 20, 4 (2022), 2762–2780.

[30] Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. 2024. Logshrink: Effective log compression by leveraging commonality and variability of log data. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[31] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3058–3070.

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

[33] Jimmy Lin et al. 2010. YCSB: A Workload Generation Framework for Cloud Databases. https://github.com/brianfrankcooper/YCSB

[34] Chunwei Liu, John Paparrizos, and Aaron J Elmore. 2024. Adaedge: A dynamic compression selection framework for resource constrained devices. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1506–1519.

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

[36] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering*. 2279–2290.

[37] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.. In *IJCAI*, Vol. 19. 4739–4745.

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

[39] R. Pasco. 1977. Source coding algorithms for fast data compression (Ph.D. Thesis abstr.). *IEEE Transactions on Information Theory* 23, 4 (1977), 548–548.

[40] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.

[41] Kirk Rodrigues, Yu Luo, and Ding Yuan. 2021. CLP: Efficient and scalable search on compressed text logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 183–198.

[42] Guoping Rong, Shenghui Gu, Haifeng Shen, He Zhang, and Hongyu Kuang. 2023. How Do Developers' Profiles and Experiences Influence their Logging Practices? An Empirical Study of Industrial Practitioners. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 855–867.

[43] Carl Martin Rosenberg and Leon Moonen. 2020. Spectrum-based log diagnosis. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[44] Amazon Web Services. 2012. Amazon Redshift: Data Warehousing Service. https://aws.amazon.com/redshift/

[45] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[46] Benzhao Tang, Shiyu Yang, Zhitao Shen, Wenjie Zhang, Xuemin Lin, and Zhihong Tian. 2025. LogLite: Lightweight Plug-and-Play Streaming Log Compression. arXiv:2507.10337 [cs.DB] https://arxiv.org/abs/2507.10337

[47] Rui Wang, Devin Gibson, Kirk Rodrigues, Yu Luo, Yun Zhang, Kaibo Wang, Yupeng Fu, Ting Chen, and Ding Yuan. 2024. μSlope: High Compression and Fast Search on Semi-Structured Logs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 529–544.

[48] Zhiyi Wang and Shimin Chen. 2017. Exploiting common patterns for tree-structured data. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 883–896.

[49] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. 2023. Loggrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 452–468.

[50] Junyu Wei, Guangyan Zhang, Yang Wang, Zhiwei Liu, Zhanyang Zhu, Junchao Chen, Tingtao Sun, and Qi Zhou. 2021. On the feasibility of parser-based log compression in Large-Scale cloud systems. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 249–262.

[51] Wikipedia. 2024. Zstandard. https://en.wikipedia.org/wiki/Zstd

[52] Wikipedia contributors. 2023. Deflate — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Deflate&oldid=1148886022

[53] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. 2018. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 127–138.

[54] Guangba Yu, Pengfei Chen, Pairui Li, Tianjun Weng, Haibing Zheng, Yuetang Deng, and Zibin Zheng. 2023. Logreducer: Identify and reduce log hotspots in kernel on the fly. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1763–1775.

[55] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *Proceedings of the 44th international conference on software engineering*. 623–634.

[56] Jiujing Zhang, Zhitao Shen, Shiyu Yang, Lingkai Meng, Chuan Xiao, Wei Jia, Yue Li, Qinhui Sun, Wenjie Zhang, and Xuemin Lin. 2023. High-Ratio Compression for Machine-Generated Data. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[57] Bolong Zheng, Yongyong Gao, Jingyi Wan, Lingsen Yan, Long Hu, Bo Liu, Yunjun Gao, Xiaofang Zhou, and Christian S. Jensen. 2023. DecLog: Decentralized Logging in Non-Volatile Memory for Time Series Database Systems. *Proc. VLDB Endow.* 17, 1 (2023), 1–14.

[58] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. 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*. 683–694.

[59] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. 2023. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*.

[60] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.

[61] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory* 24, 5 (1978), 530–536.