LOGPRISM: Unifying Structure and Variable Encoding for Effective Log Compression

Yang Liu*, Kaiming Zhang*, Zhuangbin Chen*†, Jinyang Liu‡, Zibin Zheng*
*Sun Yat-sen University, Zhuhai, China, {liuy2355, zhangkm7}@mail2.sysu.edu.cn,
{chenzhb36, zhzibin}@mail.sysu.edu.cn
‡The Chinese University of Hong Kong, Hong Kong, China, jyliu@cse.cuhk.edu.hk

arXiv:submit/7192585 [cs.SE] 24 Jan 2026

***Abstract*—The prevailing "*parse-then-compress*" paradigm in log compression fundamentally limits effectiveness by treating log parsing and compression as isolated objectives. While parsers prioritize semantic accuracy (i.e., event identification), they often obscure deep correlations between static templates and dynamic variables that are critical for storage efficiency. In this paper, we investigate this misalignment through a comprehensive empirical study and propose LOGPRISM, a framework that bridges the gap via *unified redundancy encoding*. Rather than relying on a rigid pre-parsing step, LOGPRISM dynamically integrates structural extraction with variable encoding by constructing a *Unified Redundancy Tree (URT)*. This hierarchical approach effectively mines "structure+variable" co-occurrence patterns, capturing deep contextual redundancies while accelerating processing through pre-emptive pattern encoding. Extensive experiments on 16 benchmark datasets confirm that LOGPRISM establishes a new state-of-the-art. It achieves the highest compression ratio on 13 datasets, surpassing leading baselines by margins of 4.7% to 80.9%, while delivering superior throughput at 29.87 MB/s (1.68×∼43.04× faster than competitors). Moreover, when configured in single-archive mode to maximize global pattern discovery, LOGPRISM outperforms the best baseline by 19.39% in compression ratio while maintaining a 2.62× speed advantage.**

***Index Terms*—Information Redundancy, Log Compression, Log Analysis, System Maintenance**

I. INTRODUCTION

Software systems generate logs to record runtime events, errors, and operational states, which are indispensable for system maintenance [1]–[8] and performance optimization [9]–[11]. However, the sheer volume of log data poses significant storage challenges [12], [13]. Modern large-scale systems can produce terabytes or even petabytes of logs daily [14]–[16]. Thus, efficient log compression has become a critical concern for managing storage costs while preserving the analytical value of historical log data [14], [17].

Logs typically follow a "template+variables" pattern, where static strings are interleaved with dynamic runtime parameters. General-purpose compression algorithms like gzip [18] and LZMA [19] fail to exploit the specific characteristics of log data, often resulting in suboptimal compression ratios [20]. To address this, recent research has focused on log-specific compression methods that leverage the inherent semi-structured nature of logs. By separating the template and variable components, parser-based compressors can replace repetitive templates with compact identifiers and compress variable streams with specialized encoding techniques, achieving significantly higher efficiency.

Despite these advancements, existing approaches face a fundamental limitation, i.e., the decoupling of log parsing [21], [22] and compression. Current workflows treat parsers as a black-box pre-processor, which aim to maximize semantic accuracy (i.e., identifying the correct templates) without regard for downstream compression effectiveness. To quantify the impact of this misalignment, we conduct a comprehensive empirical study evaluating four state-of-the-art compressors and nine parsers. Our findings reveal that semantically accurate parsers may produce templates that undermine compression performance, such as over-generalized templates that offload excessive entropy to the variable stream, or over-fitted templates that incur substantial dictionary overhead. Furthermore, the conventional "parse-then-compress" pipeline creates a boundary between the log structure and parameters, preventing the exploitation of deeper redundancies. Specifically, it ignores *template-variable correlations*, where specific variable values are strongly tied to a template and could be encoded as part of the structure, and *inter-variable correlations*, where variables within a single log entry co-occur in predictable patterns. By treating these components in isolation, existing methods fail to mine and encode these high-value aggregate patterns.

To overcome these limitations, we propose LOGPRISM, a log compression framework that unifies structural extraction and variable encoding. Instead of relying on a pre-defined parsing stage, LOGPRISM constructs a *Unified Redundancy Tree (URT)* that dynamically models both log structure and variable correlations in an integrated representation. Our approach employs a hierarchical redundancy mining strategy that progressively distills log data through three stages. First, we extract stable log tokens to construct a structural tree, establishing a compact skeleton for subsequent analysis. Second, we extend the skeleton by building variable subtrees to mine frequent "structure + variable" co-occurrence patterns, effectively bridging the gap between templates and parameters. Finally, we execute residual data processing to efficiently handle the remaining high-entropy "long-tail" variables using a specialized sorting and stream normalization pipeline.

This design maximizes compression ratios by capturing deep contextual redundancies while simultaneously accelerating processing speed. By filtering out the majority of high-frequency patterns in the early stages, LOGPRISM drastically reduces the computational load on the final, expensive residual

†Zhuangbin Chen is the corresponding author.

processing stage. We further enhance scalability through a parallel-aware architecture that supports fine-grained concurrency. We evaluate LogPrism on 16 benchmark datasets from LogHub [21], covering diverse system types and log formats. The results demonstrate that LogPrism establishes a new state-of-the-art in both effectiveness and efficiency, achieving the highest compression ratio on 13 out of 16 datasets and outperforming the leading baseline Denum by up to 25.25%, all while delivering the fastest compression speed (averaging 29.87 MB/s).

In summary, this paper makes the following contributions:

- We conduct the first comprehensive empirical study to quantify the impact of log parsers on compression, revealing the critical misalignment between parsing accuracy and compression efficiency.
- We propose the concept of unified redundancy encoding, a paradigm shift that co-designs structural extraction and variable encoding to exploit deep "structure+variable" correlations. Based on this concept, we design and implement LogPrism, a high-performance log compression framework featuring a unified redundancy tree and a parallel-aware architecture.
- We perform an extensive evaluation demonstrating that LogPrism significantly outperforms existing state-of-the-art methods in both compression ratio and speed.

II. Background and Motivation

A. Parser-based Log Compression

In the realm of log compression, parser-based methods represent the predominant approach. They leverage the inherent semi-structured nature of logs, typically composed of static format strings and dynamic parameters. The process begins with a critical prerequisite step, i.e., *log parsing*. As illustrated in Fig. 1, parsers like Drain [23] are employed decompose raw log messages, separating structured headers (e.g., timestamps, log levels) from the free-form body, and then segment the body into an invariant event template and its corresponding variables. This structured representation enables the core compression mechanism, where repetitive template text is replaced with compact identifiers (stored only once in a dictionary) and the extracted parameters are aggregated for subsequent encoding using specialized compression techniques. The resulting output, which is a highly regular stream of template IDs and parameter arrays with significantly reduced entropy, is then processed by general-purpose algorithms like LZMA or gzip to eliminate any remaining statistical redundancy. Several representative log compressors [24]–[27] exemplify different design philosophies within this framework.

The performance of these parser-based compressors is fundamentally tied to the quality of the log parsing stage [27], [28]. However, existing research mainly focuses on compression algorithms that operate on the parsed logs. Their designs, therefore, default to an assumption of ideal parsing quality. The parser is often treated as an independent component rather than an integral part of the compression pipeline whose performance demands critical evaluation. This has led to a common practice where an off-the-shelf parser is selected

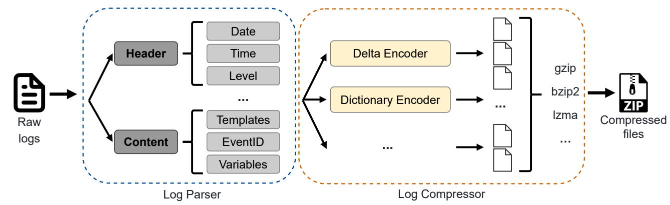

Fig. 1. The General Workflow of Parser-based Log Compression

based on secondary criteria like execution speed or ease of integration. However, there is no one-size-fits-all parser. Log parsers employ a variety of heuristics and algorithms, and their accuracy can vary significantly across different datasets with diverse log formats. Any errors introduced during parsing [22], [29]–[31] will inevitably propagate and undermine the effectiveness of the downstream compression algorithms. Despite this critical dependency, the interplay between parsing accuracy and compression efficiency remains largely unexplored. To systematically evaluate the impact of different log parsers on the performance of log compression, and to further reveal the critical role of parsing in the compression pipeline, we conduct a comprehensive empirical study in this paper.

B. Revisiting Log Parsers in Compression Pipelines

In this section, we perform an empirical study to investigate how the choice of log parsers quantitatively affects the performance of existing log compressors. We collect a set of representative log parsers and parser-based log compressors, and perform controlled experiments across multiple benchmark datasets that are widely used in log analysis domain.

1) Experiment Setup: We introduce the experiment setup of our study as follows:

Log Parser Selection: We select nine representative log parsers, i.e., Drain [23], AEL [32], IPLoM [33], LFA [34], LogSig [35], MoLFI [36], SHISO [37], Spell [38] and the parser implemented in LogReducer [25]. By covering this broad spectrum of algorithmic strategies, our study can generate heterogeneous template sets and provide a robust evaluation of their impact on downstream compression performance. These parsers span seven distinct methodological families, from simple heuristics to complex data-driven models. For instance, AEL represents the heuristic approach that uses lightweight, generic rules to distinguish static text from dynamic parameters, requiring minimal domain knowledge. In contrast, Drain and LogReducer employ fixed-depth parsing trees, where root-to-leaf paths define templates, enabling efficient online processing. Other parsers frame the task as a data mining problem. LFA applies frequent pattern mining and IPLoM iteratively partitions log messages based on token position and cardinality. Clustering-based parsers like LogSig and SHISO identify templates by computing pairwise message similarity and treat each resulting cluster's centroid as a template. Our selection also includes other novel strategies. Spell identifies templates based on the longest common subsequence algorithm, making it particularly effective for streaming data. MoLFI leverages evolutionary algorithms to iteratively evolve optimal template sets via genetic operations.

TABLE I
DETAILED STATISTICS OF BENCHMARK LOG DATASETS

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

Log Compressor Selection: To select representative log compressors, we review existing solutions [13], [14], [24]–[27], [39]–[41] in the literature and apply two filtering criteria. First, we exclude online compressors [13], [39]. Our study focuses on offline lossless compression for long-term storage, which prioritizes maximizing storage efficiency across the entire dataset. In contrast, online methods optimize for reducing real-time transmission overhead, often operating under tight resource constraints and with the possibility of lossy compression [39], [42], [43]. Second, we prioritize selecting the state-of-the-art approaches that represent the most effective methodologies in the field. This process yields four representative compressors: Logzip [24], LogReducer [25], LogShrink [26], and Denum [27]. While Denum is primarily a number-centric approach designed to bypass traditional template extraction, we include it because it relies on parser-based strategies like LogShrink to compress its non-numeric log content. Consequently, how the input text is parsed or tokenized remains a critical factor in Denum's overall performance.

These approaches feature a different design in log structure extraction and encoding strategies, providing a diverse testbed for our study. Logzip, as a pioneer in parser-based compression, employs iterative clustering to uncover latent structures, transforming logs into a hierarchical representation with template identification and parameter mapping. LogReducer advances this idea by focusing on inter-parameter correlations, utilizing an elastic numeric encoding scheme to dynamically select compact bit representations, complemented by delta encoding for sequential data. Similarly, LogShrink optimizes parameter storage by reorganizing data into columnar formats, grouping homogeneous types to create low-entropy streams ideal for dictionary encoding [44]. Finally, Denum introduces a number-centric approach that targets numerical data, using regular expressions to isolate numeric strings for categorization and differential encoding to minimize redundancy.

Dataset Selection: To ensure generalizability and robustness, our evaluation utilizes the widely recognized LogHub benchmark [21]. As detailed in Table I, this collection comprises 16 datasets spanning a broad spectrum of system types, including large-scale distributed systems (e.g., HDFS, Spark), supercomputers (BGL, Thunderbird), server applications (Apache, OpenSSH), operating systems (e.g., Linux, Windows), mobile platforms (Android, HealthApp), and standalone software (Proxifier). In total, the full benchmark contains over 77 GB of raw logs and approximately 378 million log entries. However, our preliminary experiments revealed that the Android and Windows datasets could not be processed by some parsers within a reasonable time budget due to their inherent complexity. Consequently, we exclude these two datasets from our empirical study. Our final evaluation is conducted on the remaining 14 datasets, which collectively account for over 50 GB of data and 262 million log entries, spanning diverse log formats and structural complexities.

Evaluation Metrics: To quantify how different log parsers impact the effectiveness of downstream compression, we employ the Compression Ratio (CR), a standard metric in data compression. It is defined as the ratio of the original log file size to the compressed file size. A higher value indicates better compression performance.

$$Compression\ Ratio\ (CR) = \frac{Original\ Log\ Size}{Compressed\ File\ Size}$$

2) Experiment Workflow: We design a controlled workflow for the study, which is structured into three phases, i.e., *log parsing*, *intermediate normalization*, and *log compression*.

In the initial phase, we apply the nine selected log parsers to generate structured template collections for each dataset. During this process, we observe significant inconsistencies in parser outputs regarding wildcard notation, delimiter usage, and file formats. For example, some parsers use $<*>$ to denote variables, while others use markers like `spec` or simply omit the variable tags. Such heterogeneity creates format incompatibilities that can prevent downstream compressors from correctly recognizing templates. To eliminate these discrepancies, we perform intermediate normalization in the second phase. This standardizes all parser outputs into a single canonical format, while preserving semantic content. Specifically, we unify all wildcard symbols, align metadata structures to a uniform schema, and convert outputs to a consistent file format. This ensures that compression performance differences reflect parsing quality rather than formatting artifacts. In the final phase, we evaluate the four selected compressors using these normalized templates. This presents a technical challenge, as most off-the-shelf log compressors are monolithic systems with tightly coupled parsing and compression modules. To address this, we systematically refactor their source code, decoupling data ingestion from the core compression logic. We develop dedicated input interfaces that allow us to inject our normalized templates, creating a controlled environment to test each compressor against templates from all nine parsers.

3) Experimental Results: Fig. 2 presents the compression ratio achieved by four log compressors when supplied with templates generated by nine different parsers. Our analysis reveals three key observations that fundamentally challenge the current parser-based paradigm for log compression.

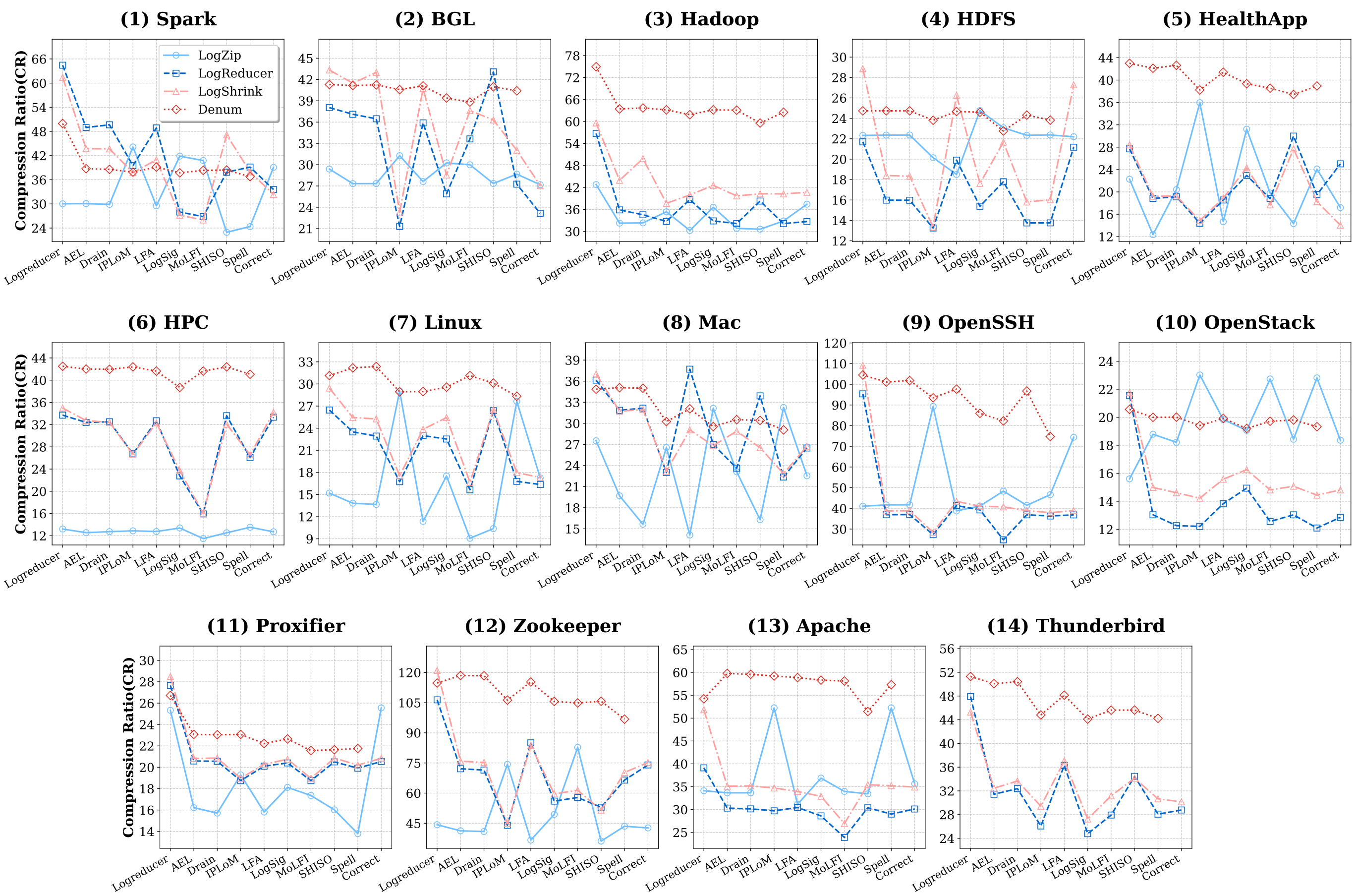

Fig. 2. Impact of Log Parser Selection on Downstream Compression Ratio across Different Datasets

Observation 1: Dramatic Performance Variance Induced by Parser Selection. For any given compressor, the choice of parser induces dramatic variance in the final compression ratio, often surpassing the inherent performance differences between the compressors themselves. Three compressors (LogZip, LogReducer, and LogShrink) exhibit particularly strong sensitivity. For example, on Zookeeper, LogShrink achieves a CR of 121.06 when using templates from LogReducer's parser, but this plummets to just 44.81 with the IPLoM parser's templates. Denum, which is not fully parser-based, also demonstrate notable performance fluctuations, confirming that the handling of non-numeric text remains critical. Such results provide clear evidence that parser selection plays a critical, yet previously overlooked, role in compression efficiency.

Observation 2: Default parsers are not always optimal. Counter-intuitively, our experiments reveal that compressors can achieve higher compression ratios using external parsers rather than their built-in ones. On BGL, for instance, LogReducer achieves a CR of 38.04 when using its own default parser, but this increases to 43.08 when paired with the SHISO parser. A similar trend is observed on the HealthApp dataset, where SHISO again outperforms LogReducer's native parsing logic (29.98 vs. 27.71). These findings suggest that the tight coupling of specific parsers with compressors in current designs can be suboptimal, preventing the compression algorithms from realizing their full potential on diverse datasets.

Observation 3: Parsing Accuracy Does Not Guarantee Compression Efficiency. Our results demonstrate that there is no universally optimal parser capable of consistently maximizing compression efficiency across all datasets and compressors. While Drain generally performs well, it is significantly outperformed by SHISO and LFA on Thunderbird. Conversely, SHISO excels on BGL but yields suboptimal results on HDFS and OpenStack. Even the ground-truth templates (labeled "Correct") are frequently outperformed by heuristic parsers. This variance indicates that a parser's effectiveness for compression is not intrinsic but depends on the complex interplay between data characteristics and the compressor's encoding strategy. The root cause of this volatility lies in a fundamental misalignment of objectives: parsers prioritize classification accuracy to maximize event clustering correctness, whereas compressors prioritize storage efficiency. Parsers emphasize semantic accuracy without considering the storage cost of the resulting templates and parameters. This leads to two typical inefficiencies that degrade compression performance despite high parsing accuracy:

- *Over-generalization (Coarse-grained Templates)*: Parsers prioritizing high matching rates with fewer templates often introduce excessive wildcards. On HDFS, for instance, LFA generates only 42 templates but with a total of 372 wildcards (8.8 per template on average). While this may yield high parsing accuracy, it offloads complexity to the

```
L1: Jan 26 15:32:30 combo kernel: audit(1119799950.864:693295): initialized
L2: Jan 26 15:32:31 combo kernel: audit(1119799950.865:693309): initialized
L3: Jan 26 15:32:33 combo httpd[3015]: received request; transaction_id: 5001
L4: Jan 26 15:32:35 combo database[3016]: query executed; transaction_id: 5001
L5: Jan 26 15:32:36 combo sshd(pam_unix)[3219]: authentication failure; user=test1 rhost=pokemon1.cs.edu uid=509 euid=0
L6: Jan 26 15:32:37 combo sshd(pam_unix)[3221]: authentication failure; user=test1 rhost=srv2.alfahost.nl uid=509 euid=0
L7: Jan 26 15:32:40 combo sshd(pam_unix)[3224]: authentication failure; user=test1 rhost=pokemon1.cs.edu uid=509 euid=0
L8: Jan 26 15:32:42 combo httpd[3015]: received request; transaction_id: 5002
L9: Jan 26 15:32:48 combo sshd(pam_unix)[3225]: authentication failure; user=root rhost=pc180.edu.tw uid=0 euid=0
L10: Jan 26 15:32:55 combo sshd(pam_unix)[3226]: authentication failure; user=root rhost=julia.arkos.de uid=0 euid=0
L11: Jan 26 15:32:58 combo sshd(pam_unix)[3229]: authentication failure; user=root rhost=pc180.edu.tw uid=0 euid=0
L12: Jan 26 15:33:07 combo sshd(pam_unix)[3231]: authentication failure; user=root rhost=julia.arkos.de uid=0 euid=0
```

Fig. 3. A 12-line Log Snippet as a Running Example

parameter stream. The compressor is forced to process large, noisy parameter sets that should have been static template text, severely degrading the compression ratio.

- *Over-fitting (Fine-grained Templates):* Conversely, overly sensitive clustering thresholds can partition semantically similar logs into numerous distinct templates. For example, some parsers produce over 28,000 templates for the HealthApp dataset, which contains only around 150 actual event types. This "template explosion" exponentially increases the dictionary size, creating a direct storage overhead that undermines compression efficiency despite high formal parsing precision.

In summary, our empirical study confirms that log parsing is not a mere preprocessing step but a dominant factor in the ultimate efficiency of log compression. The fundamental principle of exploiting log structural redundancy is sound, which enables parser-based approaches consistently and significantly outperforming general-purpose algorithms like gzip or LZMA. However, our findings reveal a critical flaw in its common decoupled implementation. We argue that unlocking optimal log storage performance requires a paradigm shift where structural extraction is not a prerequisite for compression but an integral, co-designed part of it.

C. A Motivating Example for Unified Redundancy Encoding

In this section, we illustrate the core concept of our approach based on the logs in Fig. 3. The idea is to move beyond the rigid dichotomy of "static strings vs. dynamic variables" by modeling log tokens (whether traditionally considered a static string or a variable) uniformly and encode them in aggregate based on their co-occurrence and dependencies. To this end, we propose a paradigm shift from the traditional *parse-then-compress* workflow and introduce a new methodology called *unified redundancy encoding*.

Consider the `sshd` log entries in Fig. 3 (L5-L7 and L9-L12). Since variable `user`, `rhost`, and `uid` take different values, a conventional parser may generate a log template like "`...authentication failure; user=<*> rhost=<*> uid=<*> euid=0`". When compressing these logs, e.g., L5, existing compressors would first store an identifier for this template and then independently encode the variable values (`test1`, `pokemon1.cs.edu`, and `509`). In this process, two types of correlations are ignored. The first is *template-variable correlation*. For pure log compression purposes, the template ID could include certain variables if they are strongly tied to the template body, eliminating separate parameter processing. Although parsers may occasionally inline frequent variables (e.g., `euid=0` in Fig. 3) into the template, this behavior is inconsistent and not measurable. The second is *inter-variable correlation* within each log entry, which allows for the collective processing of variables. For instance, user `test1` deterministically maps to uid `509`, and user `root` maps to uid `0`. This differs from prior correlation mining methods [26], [27] that mainly model relationships across log entries (e.g., incremental user IDs).

Our method is more holistic and context-aware. It recognizes that the value set {`test1`, `pokemon1.cs.edu`, `509`} (L5 and L7) is a frequent pattern occurring within the context of the aforementioned template. Therefore, our method aggregates both parts as "`...authentication failure; user=test1 rhost=pokemon1.cs.edu uid=509 euid=0`" and encodes it with a single ID. Similar frequent patterns include {`root`, `pc180.edu.tw`, `0`} (L9 and L11) and {`root`, `julia.arkos.de`, `0`} (L10 and L12), while only `srv2.alfahost.nl` (L6) needs to be handled independently. The advantage of this unified redundancy encoding is significant: whereas existing approaches require storing one template ID plus three separate variable IDs, we need only a single ID to represent the entire, highly correlated token sequence. This principle of co-designing structural extraction and pattern encoding enables us to discover and exploit deep contextual redundancies in log data.

A straightforward way to implementing this idea is to build a prefix tree (i.e., a trie) over the sequences of log tokens and assign IDs to frequent paths, thereby identifying recurring log (sub)sequences. However, this naive construction presents two fundamental limitations. First, high-cardinality fields induce many branches, causing the number of nodes to grow with the product of distinct values per position. This results in prohibitive storage consumption and increased lookup latency due to structural inefficiency. Second, a trie only encodes prefixes. If an infrequent token appears early, the match stops and the rest of the log line is no longer compressible, even when the suffix is highly regular (e.g., `uid=509 euid=0` after a rare `rhost` in L6). These constraints motivate our design of LogPrism, an effective log compressor which moves beyond pure prefix matching and supports compact, gap-tolerant aggregation of frequent token sequences.

III. Methodology

In this section, we introduce LogPrism, our log compression framework that constructs a *Unified Redundancy Tree (URT)* to jointly model log structure and variable correlations. Since a naive prefix tree over raw logs would suffer from combinatorial branch explosion, LogPrism employs a hier-

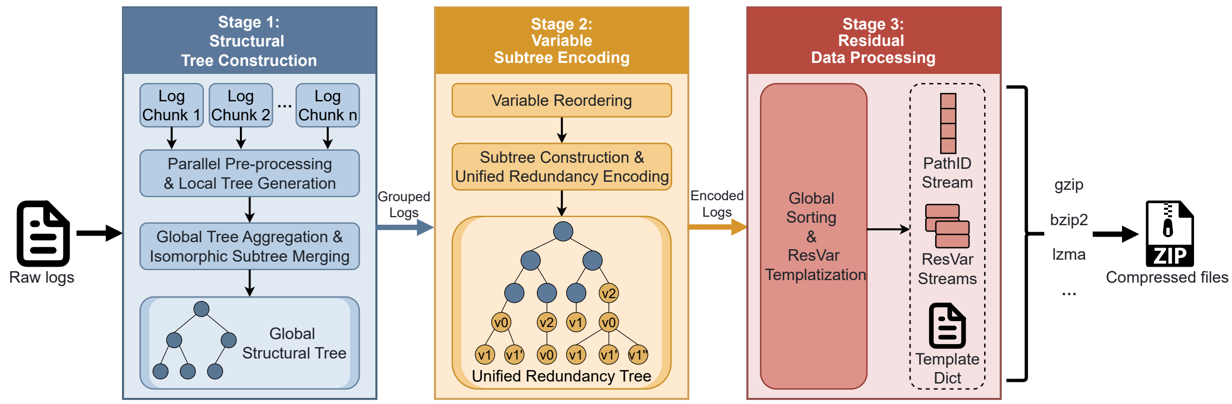

Fig. 4. An Overview of the LogPrism Compression Framework

archical redundancy mining strategy to build a structurally efficient URT. We progressively integrate log tokens into the URT based on their frequency and stability through three stages, i.e., *Structural Tree Construction*, *Variable Subtree Encoding*, and *Residual Data Processing*, as shown in Fig. 4. The first stage builds the structural skeleton of the URT based on regular, fixed tokens. Each path in this tree represents a group of log entries that are structurally similar, regardless of the log events they represent. The second stage integrates frequent variable tokens by expanding the skeleton's terminal nodes into subtrees. Within each subtree, we mine deep correlations between the structural skeleton and specific variable values, allowing a single ID to represent complex, recurring "structure-variable" patterns. The third stage handles the remaining "long-tail" tokens (i.e., outliers that are too rare or random) by isolating and encoding them via specialized schemes suited for high-entropy data. The entire compression pipeline concludes by feeding all compressed data into a general-purpose compressor (e.g., LZMA) to exploit remaining byte-level redundancy.

It is essential to note that LogPrism's hierarchical compression pipeline is fundamentally different from the decoupled "parse-then-compress" workflow. LogPrism's ultimate goal is to unify the encoding of both structural and variable tokens into an end-to-end redundancy pattern, rather than treating them as separate entities for isolated compression.

A. Structural Tree Construction

The primary objective of this stage is to construct the foundational skeleton of the URT by extracting stable patterns from the raw logs. To ensure high throughput and a low memory footprint, we implement this process within a parallel streaming architecture. The log dataset is partitioned into multiple chunks, each processed concurrently by a dedicated worker thread to build local structures. As depicted in Fig. 4, this construction phase proceeds through two key steps: 1) *Parallel Pre-processing and Local Tree Generation*, where logs are tokenized, filtered, and organized into local prefix trees; and 2) *Global Tree Aggregation and Isomorphic Subtree Merging*, which integrates these local trees into a unified global structure while dynamically refining the topology.

1) Parallel Pre-processing and Local Tree Generation: The pipeline within each worker thread involves two operations, i.e., the pre-processing of the assigned log chunk to filter volatile content, and the construction of a local structural tree.

To prevent the "branch explosion" associated with high-cardinality data, we identify and separate two categories of rapidly-changing tokens.

- *Globally Patterned Metadata*: This category includes common header fields like dates, timestamps, and process IDs (PIDs) [21]. These tokens, while highly variable, exhibit predictable syntactical patterns. Storing them in a string-based tree is highly inefficient. Thus, we identify them using predefined regular expressions and extract their values into separate, highly compressible columnar streams, preserving the global order of the logs. As illustrated in Fig. 5(a), these tokens are replaced with specific placeholders (e.g., $\langle X \rangle$ for month, $\langle dt \rangle$ for timestamp, $\langle P \rangle$ for PID) in raw logs.
- *Unstructured Numeric Tokens*: This category targets tokens within free-text log contents that are likely to be variables. Based on a simple and effective heuristic from prior research [27], we treat tokens containing numeric characters as potential variables. We apply this rule to tokens not captured by the metadata regex and uniformly replace the matched ones by the wildcard placeholder $\langle * \rangle$, as shown in Fig. 5. The original values of these tokens are collected sequentially into a `varList` associated with the specific log entry. In this process, two types of misclassifications can happen. First, static tokens containing numbers (e.g., `node1`) are generalized as variables. Since LogPrism treats static and dynamic tokens as a holistic entity, these tokens can be re-integrated into the structure during correlation mining stage (Sec. III-B). Second, string-only variables (e.g., `user=root`) are treated as static, causing potential tree fragmentation. This is explicitly addressed via isomorphic subtree merging in the subsequent step.

After pre-processing, the resulting log messages are used to construct a local prefix tree. For example, dividing the logs from Fig. 3 into two chunks (L1-L6 and L7-L12) yields the independent trees shown in Fig. 6(a) and Fig. 6(b), respectively. In this structure, every terminal node marks the end of a log message. Crucially, a terminal node is not necessarily a leaf node, since one log entry can be a complete prefix of another. Each path from the root to a terminal node uniquely

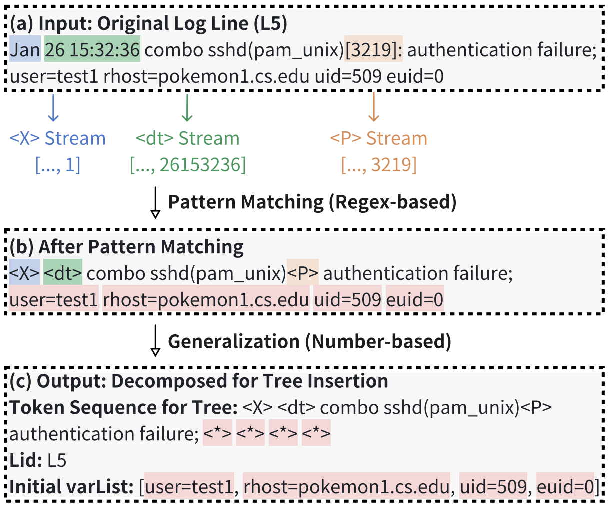

Fig. 5. The Pre-processing Pipeline for Log Entry L5

identifies a group of log entries sharing the same structural pattern. To index these groups, a `Record` Object is maintained at every such terminal node, aggregating the Line IDs (`Lid`) and extracted `varList`(s) for all corresponding logs.

2) Global Tree Aggregation and Isomorphic Subtree Merging: Once the worker threads generate their local structures, the main thread integrates them into a single global tree and refines the topology. We employ an on-arrival merge strategy to minimize synchronization overhead. As soon as a worker thread completes its chunk, its local prefix tree is merged into the global one. As depicted in Fig. 6(c), this merge process aggregates information along identical paths and adds new branches where structures differ (e.g., the `user=root` branch from the second chunk). Concurrently, the columnar data streams from each thread are appended to global files, preserving the original order of the log dataset. For numerical streams like $\langle dt \rangle$, we apply a specialized compression pipeline consisting of Delta Encoding (to store value differences), ZigZag Encoding (to efficiently represent negative numbers), and Varint Encoding (for variable-length integer representation) to generate a final compact binary format.

Although the number-based heuristic for identifying volatile fields in the first step is effective, it can potentially misclassify purely string-based variables. For instance, `user=test1` is correctly generalized to $\langle * \rangle$ (due to `"1"`), but `user=root` would be incorrectly identified as a stable token. This inconsistency creates unnecessary branches and fragments the tree. To resolve this, we introduce a correction process named *isomorphic subtree merging*. It operates on the principle that if a "static" branch and a "variable" branch lead to subtrees that are topologically isomorphic, they serve the same structural role and should be treated uniformly as a variable.

The merging process is implemented as a post-order (bottom-up) traversal of the global tree. At each branching node, it computes a unique structural signature for every child subtree. If a node contains a linear path, its signature is simply the concatenation of its tokens. For nodes with multiple branches, we lexicographically sort the signatures of all outgoing branches before concatenating them. Any sibling nodes sharing identical structural signatures will be considered isomorphic, and the algorithm performs a merge operation. Fig. 7 demonstrates this iterative merging. The traversal moves up from the leaves and, as shown in Fig. 7(a), encounters its first branching node `user=root`. It compares the child nodes $\langle * \rangle$ (which represents numeric `rhosts`) and `rhost=julia.arkos.de` by computing the structural signature for the path under each. Since both yield the same sequence of $\langle * \rangle$ nodes (as highlighted by the green dashed box), they are topologically identical. This isomorphism indicates that `rhost=julia.arkos.de` should also be a variable token (i.e., $\langle * \rangle$). The system then merges these two paths by (i) inserting the value `"rhost=julia.arkos.de"` into the corresponding position of the `varList` (the red arrow line) for all logs traversing that path, and (ii) merging the `Record` Object (i.e., `Lids` and updated `varLists`) from the string branch into the sibling wildcard branch. The result is the more generalized structure in Fig. 7(b). As the traversal continues up to the next branching node, `failure;`, the system performs another similar merge as shown in Fig. 7(c).

The choice of a bottom-up traversal is critical for accurate path mergings. By processing the tree from the leaves upward, the algorithm ensures that sibling nodes sharing the longest common prefixes are evaluated for merging first. Such paths are more likely to be truly isomorphic [23]. It also ensures that any structural inconsistencies at deeper levels are resolved first, recursively generalizing the tree into its most compact form. In case of incorrect mergings, LogPrism can separate the outlier variables during the correlation mining stage (Sec. III-B).

Finally, we traverse the global tree to index the structural contexts. Every terminal node is assigned a globally unique `pathID`. As shown in Fig. 8, this ID acts as a *structural context identifier*, representing a group of logs that share the exact same skeletal structure. For example, all `sshd` logs (L5-L7 and L9-L12) in the example are mapped to `pathID=3`. The core role of the `pathID` is to provide clear log groupings for the next stage of variable correlation analysis, where each log is represented by its original line number (`Lid`), its structural context identifier (`pathID`), and its `varList`.

B. Variable Subtree Encoding

The construction of the structural context tree leverages only stable tokens, resulting in a highly compact skeleton where all variable tokens are aggregated at the terminal nodes. To capture the deep correlations between the log structure and its parameters, we can conceptually extend the URT by attaching a prefix subtree to each terminal node, using the `varLists` of the associated log entries as input paths. By doing so, we unify the structural skeleton and variables into a single, continuous representation. However, this presents two critical challenges. *First, a naive sequential insertion of variables inevitably triggers a secondary branch explosion.* As illustrated in Fig. 9(a), if variables are processed in their original order (e.g., $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$, a high-cardinality variable may appear early in the sequence (e.g., a `rhost` at v_1), forcing the subtree to branch at the top layers. Consequently,

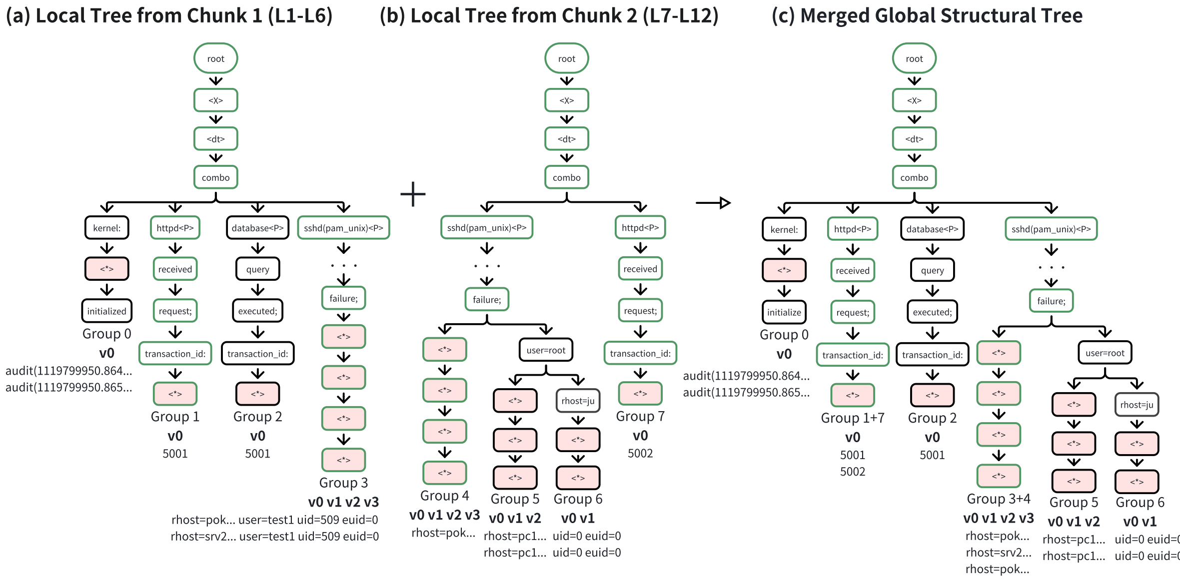

Fig. 6. The Parallel Pre-processing and Streaming Merge Process of the Structural Tree

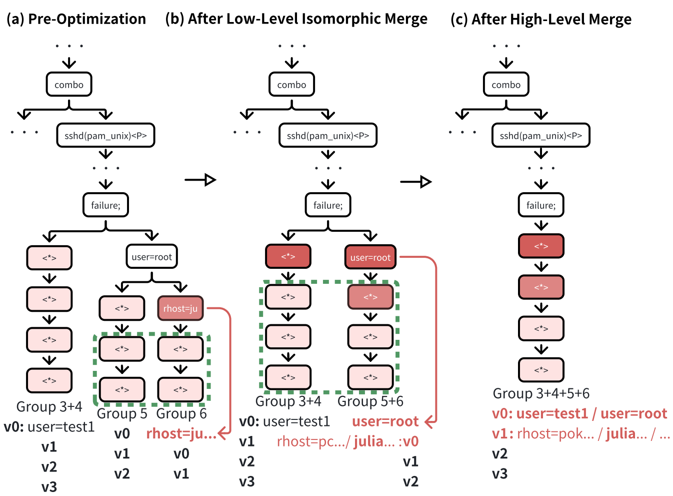

Fig. 7. The Iterative, Bottom-up process of Isomorphic Subtree Merging

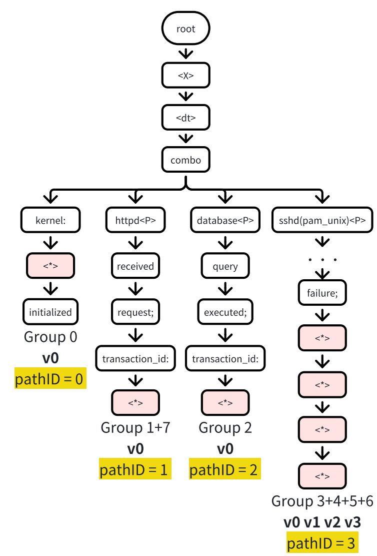

Fig. 8. The Final Global Structural Tree with `pathID` Assignments

even if later variables form a highly frequent pattern (e.g., `uid=509` $\rightarrow$ `euid=0`), it is fragmented into multiple separate branches, preventing efficient compression. *Second, we require a robust mechanism to distinguish meaningful patterns from random noise.* Since variables are high-entropy, assigning a unique ID to every combination would result in a prohibitively large dictionary and compromise the compression benefits. Therefore, we must selectively encode only frequent, co-occurring variable combinations (the "signal") into the URT, while actively filtering out rare or random values (the "noise") to be handled as residuals (Sec. III-C).

1) Variable Reordering for Efficient Subtree Construction: To mitigate the secondary branch explosion, LogPrism restructures the variable subtree topology via a two-step optimization process. This involves a stability-based sort that prioritizes low-entropy variables for insertion. By forcing frequent patterns to share a common prefix, this strategy effectively pushes high-cardinality branching to the deeper levels of the tree. Consequently, correlated variable sequences are materialized as deep, continuous paths (Fig. 8) rather than fragmenting prematurely.

The optimization begins with *high-frequency filtering*, which prunes infrequent variables, as shown in Fig. 10(a). For each variable position within a log group, we compute the frequency of every unique value and discard any that fall below a pre-defined threshold α (set to 1% of the total log count in the group), e.g., the rare `rhost=srv2.alfahost.nl`. If all values at a given position are filtered out, the entire position is excluded from subsequent analysis. For instance, in the `httpd` log group (`pathID=1`) shown in Fig. 8, the sole variable v_0 (`transaction_id`) is removed entirely because both of its values (`5001` and `5002`) are too infrequent.

The second step is *variable reordering*, which establishes the optimal insertion order for the remaining variables. The

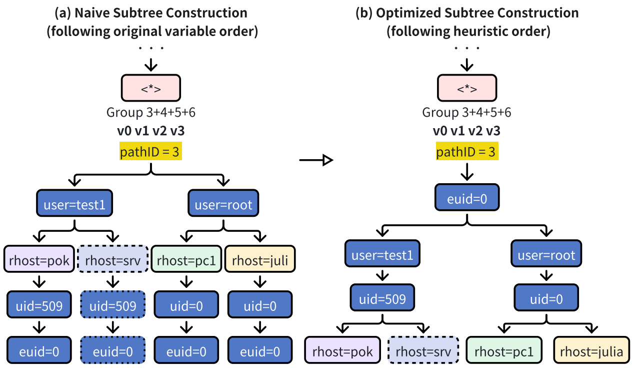

Fig. 9. The Motivation and Effect of Variable Reordering

reordering is determined by a heuristic that prioritizes stability. We calculate two key metrics for each variable position, i.e., *total frequency* (the sum of occurrences of all its values) and *discriminative power* (the number of its unique values). Variables are then sorted in descending order of total frequency. Any ties are resolved by sorting in ascending order of discriminative power, which favors variables with fewer unique values. In the `sshd` example, after filtering, v_1 has the lowest total frequency, i.e., six, while the other positions share a total frequency of seven. Among those, v_3 has the lowest discriminative power of one. This heuristic thus yields the optimal construction order of $v_3 \rightarrow v_0 \rightarrow v_2 \rightarrow v_1$, which is the crucial input for the final subtree construction phase.

2) Subtree Construction and Unified Redundancy Encoding: With the established variable order, this phase constructs the variable subtrees to identify and encode co-occurrence patterns. While the reordering was guided by aggregate statistics, it does not guarantee that individually frequent variables appear together in the same log entry. For instance, a specific `user` and `rhost` might both be globally frequent, but if they never co-occur (e.g., that user never logs in from that host), they do not form a valid pattern. Therefore, this stage validates these patterns by iterating through each log's `varList`, ensuring that compression identifiers are assigned only to variable combinations that genuinely exist in the data.

The construction process is illustrated in Fig. 10(b). As each log's `varList` is traversed according to the optimal order (i.e., $v_3 \rightarrow v_0 \rightarrow v_2 \rightarrow v_1$), a path is extended in the subtree. Every node in this tree maintains a `cnt` attribute, a key metric that records the number of logs whose reordered variables share that specific prefix. For logs like L5 and L7, which are composed entirely of frequent variables, a complete path is formed, and the `cnt` value of every node along that path is incremented. In contrast, the traversal for log L6 terminates prematurely because its variable `rhost=srv2.alfahost.nl` was filtered out as a low-frequency value. Consequently, L6 only increments the cnt values for its matched prefix (`euid=0` $\rightarrow$ `user=test1` $\rightarrow$ `uid=509`). This distinction is crucial, as the `cnt` attribute now precisely tracks the frequency of both complete and partial patterns. Next, a pruning operation is performed, which acts as a second layer of filtering, removing paths composed of individually frequent variables but whose combination is rare. Any node whose `cnt` falls below a pre-defined pruning threshold β (set to 0.5% of the total log count in the group) is removed. While no nodes are pruned in our simplified example (Fig. 10), this is crucial for eliminating noise in complex datasets.

The most critical step is the assignment of new universal `pathID`s. To maximize compression efficiency, identifiers are assigned only to nodes that represent the termination of a high-frequency aggregate pattern. This ensures that every `pathID` corresponds to a meaningful, recurring token combination, preventing an explosion in dictionary size due to rare partial matches. We term these nodes "stable endpoints" and identify them using the `cnt` attribute under two conditions. First, any leaf node is inherently a stable endpoint, as it represents the explicit termination of a pattern that has already survived the high-frequency filtering process (i.e., β). For example, the `rhost=pokemon1.cs.edu` node in Fig. 10(b) is assigned `pathID=5` because it marks the end of the pattern for logs L5 and L7. Second, a non-leaf node is designated a stable endpoint if the difference between its `cnt` and the sum of its children's `cnt` values is not less than the threshold β. This "residual count" represents the number of log entries whose pattern matches exactly up to this node but does not continue to any of its high-frequency children. By enforcing the threshold, we ensure that we only create a new ID if a substantial number of logs terminate at this specific intermediate point. For instance, the `uid=509` node has a `cnt` of 3, while its only child has a `cnt` of 2. The residual count of 1 indicates that one log (L6) terminates its match here. If this residual meets the threshold, the node is marked as a stable endpoint and assigned `pathID=4`, allowing LOGPRISM to hierarchically encode both complete patterns and frequent sub-patterns.

The final encoded output, shown in Fig. 10(c), is a highly compact representation of the log data. Log messages that fully match a frequent path, such as L7 and L9, are represented by a single `pathID` (5 and 6, respectively). Logs that only partially match, like L6, are represented by the `pathID` of their longest matched prefix (`pathID=4`), while the unmatched token (`rhost=srv2.alfahost.nl`) is preserved as a residual variable in the log's `varList` for processing in the third stage. This process is the core manifestation of our unified redundancy encoding concept. Crucially, the new `pathID` is a logical extension of the initial structural `pathID`. It represents the complete "structure + variable" collective pattern. Unlike traditional methods that require one identifier for the template and separate ones for each variable, our approach uses a single `pathID` to represent the entire high-frequency combination. For fully matched logs, this identifier replaces both the original structure and all of its variables, achieving a significant gain in compression efficiency.

C. Residual Data Processing

After the first two stages have captured high-frequency collective patterns, the remaining data constitutes the "long-tail" data, i.e., outlier variables characterized by high entropy and weak correlation. This final stage is designed to efficiently compress these residual components by leveraging simpler, linear patterns that may exist. Our approach is guided by

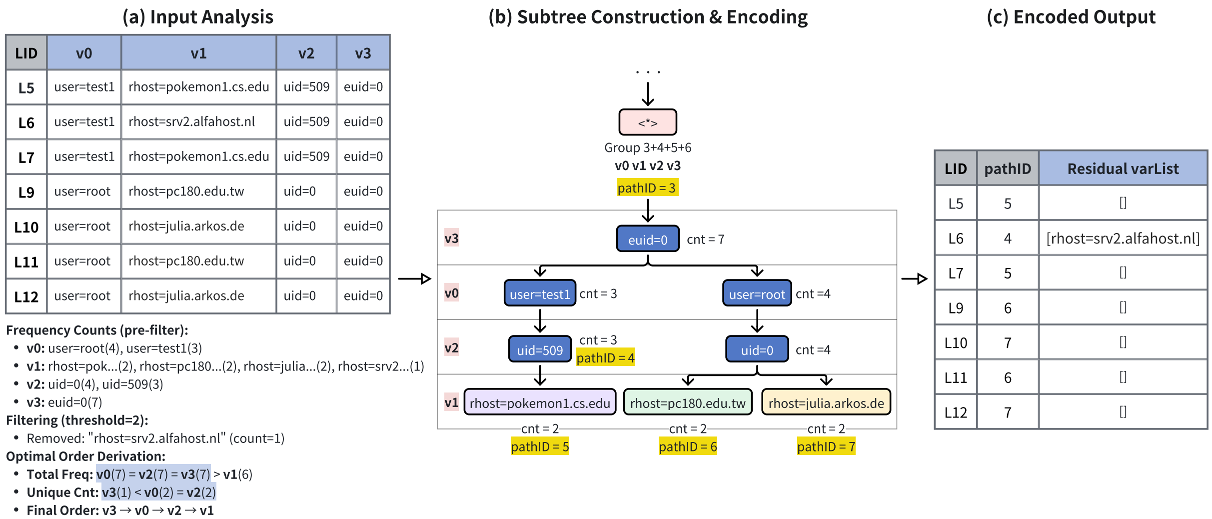

Fig. 10. The Variable Subtree Encoding Pipeline for the `sshd` Log Group (`pathID=3`)

the principle of efficient residual handling: after the URT has filtered out complex correlations, this final pipeline can focus on simpler sequential regularities.

1) Global Sorting for Temporal Coherence: The structural grouping in the first stage, while effective for mining template-based redundancy, inevitably fragments the natural time order of logs. This can obscure valuable patterns that span across different structural groups. As shown in Fig. 11(a), logs from different sources (e.g., `httpd` and `database`) may share a sequential `transaction_id`. To uncover these dependencies, the core of this stage is a *global sorting pipeline*. We aggregate the residual information from all log groups and perform a global re-sort based on the original Line ID (`Lid`). This operation restores the dataset's temporal coherence, re-aligning dispersed entries like L3, L4, and L8 and exposing the incremental numeric sequence (`5001`, `5001`, `5002`), which is now highly amenable to delta compression.

2) Just-in-Time Residual Templatization: The residual variables after the second stage are a heterogeneous mix of complex strings, atomic identifiers, and pure numbers. This diversity prevents efficient columnar compression. To resolve this, we employ a *just-in-time templatization* strategy that iterates through the sorted queue and parses each variable into homogeneous components.

For each residual variable, a regex-based extraction mechanism decomposes it into its invariant static fragments and dynamic numeric parts. We then borrow the core principle from Denum [27] to classify each extracted numeric string based on its intrinsic features (e.g., length, first digit) and generate a corresponding placeholder. This unified process gracefully handles three distinct scenarios:

- **Complex Numeric Variables**: For a composite variable like `audit(1119799950.864:693295):` in log L1, the process generates a generalized template from its static parts and placeholders (e.g., `audit(`$\langle jb\_\rangle$`.`$\langle c\_\rangle$`:`$\langle fg\_\rangle$`):`) and dispatches each numeric value to its respective columnar stream.
- **Atomic String Variables**: For a variable that cannot be further deconstructed, such as `rhost=srv2.alfahost.nl` from log L6, the entire string is treated as an indivisible atomic template and assigned a unique template ID.
- **Pure Numeric Variables**: For simple numbers like `5001`, a minimalist placeholder template (e.g., $\langle df\_\rangle$) is generated, and the value is appended to the corresponding numeric stream.

The result of this process is a set of dense, homogeneous columnar streams, as shown in Fig. 11(b). The `pathID` of each log is written to the main stream, followed by the template IDs for its residual variables in the `ResVar` streams. Concurrently, all extracted numeric values are written to their specialized Value Streams, which are then compressed using Delta Encoding followed by ZigZag and Varint encoding.

Although this stage borrows the principle of numeric classification from Denum, our hierarchical data distillation strategy is fundamentally different. Denum performs an indiscriminate, global numeric/non-numeric split from the outset, which can prematurely destroy valuable contextual correlations. In contrast, LogPrism has already processed the vast majority of high-frequency tokens (including many numerics) via unified redundancy encoding in the first two stages. Consequently, the computationally intensive parsing logic of this stage is applied only to a minimal subset of true residual data. This targeted approach provides a dual advantage: a higher compression ratio, as it preserves the "structure + variable" correlations, and faster compression speed, by confining the expensive variable processing to a much smaller dataset.

The pipeline concludes by aggregating all generated components, i.e., the URT, the residual template dictionary, and the various columnar data streams, into a single compact archive, which is finally processed by a general-purpose compressor (e.g., LZMA) to exploit remaining byte-level redundancy.

D. Decompressor

The decompression process of LogPrism is the precise inverse of compression, achieving lossless log reconstruction.

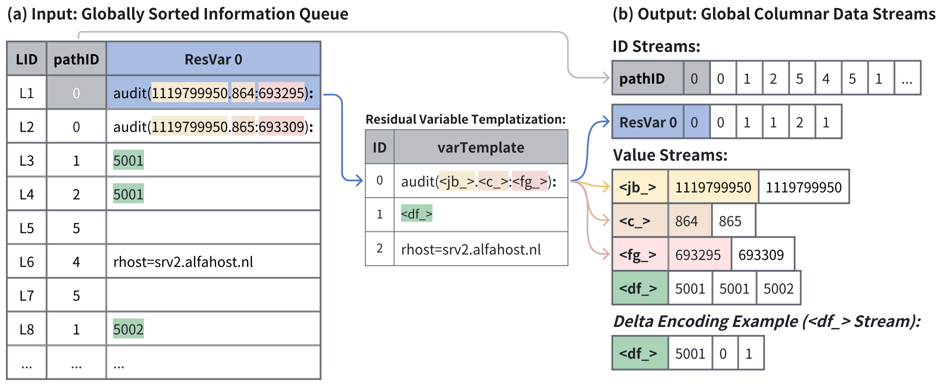

Fig. 11. The Residual Data Processing Pipeline in Stage 3

The process begins by loading the metadata from the compressed archive, including the URT, the residual template dictionary, and all columnar data streams. The core of the reconstruction is a sequential traversal of the main `pathID` stream, reconstructing logs line by line. Since this stream was written in the order of the original log entries, processing it sequentially naturally restores the original log order.

For each `pathID` read from the stream, a two-stage reconstruction process is performed. First, the `pathID` is used to perform a reverse lookup in the URT. By tracing backward from the `pathID` node to the root, we instantly reconstruct the log's complete high-frequency "structure + variable" pattern. For a log that was fully matched during compression (e.g., L5), this single lookup is sufficient to recover all its tokens, including both the structural and variable tokens. For a log with residual variables (e.g., L1 or L6), this lookup only recovers the matched portion, and the remaining variables are then reconstructed by reading their template IDs from the `ResVar` streams and resolving them against the residual template dictionary. At this point, a semi-reconstructed log line is formed, containing all static and high-frequency variable tokens but still holding placeholders for dynamic values (e.g., $\langle dt\rangle$, $\langle P\rangle$, and $\langle jb\_\rangle$). Therefore, the second step performs global placeholder substitution. For each placeholder encountered, the decompressor reads the next available value from the corresponding columnar data stream and substitutes it in place. This multi-layered mechanism guarantees 100% accuracy, as all data streams are consumed in the exact order in which they were generated, ensuring a perfect reconstruction of the original log file.

IV. Evaluation

We conduct a comprehensive evaluation to demonstrate the effectiveness and efficiency of LogPrism. Our evaluation is designed to answer four critical research questions:

- **RQ1:** How does LogPrism's compression ratio compare to state-of-the-art log compressors?
- **RQ2:** How does LogPrism's compression speed compare to state-of-the-art log compressors?
- **RQ3:** What are the individual performance contributions of LogPrism's hierarchical stages?
- **RQ4:** What is the impact of input data granularity on LogPrism's performance?

A. Experimental Settings

1) Datasets: Consistent with our previous empirical study (Sec.II-B), we utilize the LogHub benchmark [21] for evaluation. However, while the empirical study involved only 14 datasets, this evaluation incorporates the full suite of 16 datasets. The additional datasets (i.e., Android and Windows) were previously excluded due to the scalability limitations of certain parsers. In this evaluation phase, all selected compressors successfully handle these datasets using their default parsers, allowing for a comprehensive assessment across the entire benchmark.

2) Evaluation Metrics: We employ two fundamental metrics in the field of log compression: the **Compression Ratio (CR)** to quantify effectiveness (higher values indicate better compression) and the **Compression Speed (CS)** to measure efficiency (higher values indicate faster processing). The CR was previously defined in Sec. II. The CS is defined as:

$$Compression\ Speed\ (CS) = \frac{Original\ File\ Size\ (MB)}{Compression\ Time\ (s)}$$

3) Baselines: We compare LogPrism against comprehensive baselines spanning both state-of-the-art log-specific compressors and well-known general-purpose compressors.

- **Log-Specific Compressors:** We evaluate LogPrism against the same four state-of-the-art methods studied in our empirical study (Sec. II-B). These include LogZip [24], the pioneering parser-based method; LogReducer [25] and LogShrink [26], which represent advanced approaches optimized for parameter correlation and variability; and Denum [27], the leading number-centric compressor that operates without traditional parsing. This selection ensures a comprehensive comparison against the best existing techniques across different design paradigms.
- **General-Purpose Compressors:** We include four compressors in this category: *gzip* (a traditional compressor based on the DEFLATE algorithm, known for good speed but moderate compression ratios), *bzip2* (which uses the Burrows-Wheeler Transform for better compression at the cost of lower speed), *LZMA* (a dictionary-based algorithm typically offering the best compression ratios but is relatively slow), and *PPMd* (which compresses by analyzing

character sequences and predicting probabilities, excelling on plain text but degrading significantly on logs with high-entropy numeric variables).

For metrics independent of the experimental environment (CR), we directly cite the results from the original papers of the respective log compressors. For environment-dependent metrics (CS), we re-ran all available open-source tools in our unified environment to ensure fair comparison.

4) Implementation: All experiments were conducted on a Linux server with an AMD EPYC 9224 CPU (24 cores/48 threads, 3.70 GHz) and 251 GB RAM, running Ubuntu 22.04 LTS (kernel 6.8.0). LOGPRISM is implemented in C++ and compiled with g++ following the C++20 standard. Regular expression matching relies on the PCRE2 library. Similar to Denum, the `PCRE2_CODE_UNIT_WIDTH` was set to 8.

To align with the evaluation settings of Denum and related work, all input log data for RQ1-RQ3 experiments were split into 100K-line chunks for parallel compression. Each compressor uses 4 threads for processing each chunk. To ensure fair architectural comparison, our main evaluation of **LOGPRISM** constrains the internal operations of each chunk-processing thread to be single-threaded, matching the execution model of the baselines. To demonstrate the full potential of our parallel-aware design, we also evaluate an enhanced configuration, denoted as **LOGPRISM-P**, where each of the 4 chunk-processing threads utilizes an internal pool of 4 worker threads. The total time to compress all chunks is recorded. LOGPRISM employs the 'tar' utility to package all generated intermediate files, which are then compressed with 'lzma' to form the final archive.

B. RQ1: The Compression Ratio of LOGPRISM

Table II presents the compression ratio comparison across all benchmarks. LOGPRISM achieves the highest compression ratio on 13 of 16 datasets, establishing a new state-of-the-art in compression effectiveness.

Comparison with general-purpose compressors: Regarding general-purpose tools, LOGPRISM demonstrates substantial superiority over all of them. Compared to gzip, LOGPRISM achieves a 5.60× higher average compression ratio, reaching up to 27.42× on specific datasets. Against the stronger LZMA baseline, improvements range from 1.57× (Proxifier) to 5.97× (OpenSSH). Similarly, the ratio over bzip2 ranges from 1.26× (Proxifier) to 7.23× (Windows), and over PPMd, it ranges from 1.17× (Proxifier) to 7.99× (Windows). These results confirm that exploiting log-specific structure yields significant compression gains.

Comparison with log-specific compressors: LOGPRISM advances the state-of-the-art among log-specific methods. Compared to LogReducer, CR improvements reach up to 58.12%. Against LogShrink, the maximum gain is 53.31%. Notably, even compared to the current leading method Denum, LOGPRISM achieves substantial improvements of up to 25.25% on Linux and 16.34% on Thunderbird. The LogZip result for Thunderbird is marked as unavailable (–) because, as reported in the LogShrink paper [26], LogZip failed to complete parsing within one week.

This systematic performance advantage of LOGPRISM stems from a fundamental architectural shift. Traditional methods (whether the "parse-then-compress" workflows or Denum's "numeric/non-numeric" split) perform an irreversible token categorization early in the process. This leads to the loss of contextual correlations that spans these category boundaries. In contrast, LOGPRISM's unified redundancy encoding paradigm co-designs structural extraction with pattern encoding. This enables representing entire, highly correlated "structure + variable" token sequences with single path identifiers, where other methods require one template ID plus multiple independent variable IDs. Our hierarchical data distillation strategy prioritizes encoding high-value aggregate patterns while preserving full context, allowing it to exploit deep redundancies in log data.

On the three remaining datasets where LOGPRISM does not achieve the highest CR (Hadoop, HDFS, Spark), performance remains highly competitive. On Hadoop and HDFS, the differences from optimal are minimal (0.80% and 3.33%, respectively). The larger gap on Spark reflects the characteristics of the dataset, i.e., Spark logs are dominated by simple, repetitive templates with single continuously changing numeric values (e.g., over a million records like `"Update row <*>"`) [27]. In such scenarios, the deep correlation mining of LOGPRISM's second stage offers less advantage compared to methods specifically optimized for simple numeric streams. Nevertheless, as we demonstrate in RQ2, LOGPRISM achieves a better performance balance in these cases with significantly higher compression speed.

Summary for RQ1: LOGPRISM surpasses existing general-purpose and log-specific compressors in compression ratio on 13 out of 16 datasets, establishing a new state-of-the-art in compression effectiveness.

C. RQ2: The Compression Speed of LOGPRISM

We evaluate LOGPRISM's efficiency in two configurations: **LOGPRISM**, which restricts internal processing to a single thread per chunk for a fair comparison, and **LOGPRISM-P**, which activates our internal fine-grained parallelism. A critical aspect of this benchmark is the timing methodology. We measure the processing time for LOGPRISM (both configurations) and Denum on a strictly end-to-end basis. In contrast, for parser-based baselines (LogZip, LogReducer, LogShrink), we adhere to the conventions established in their respective papers, which exclude the often time-consuming parsing and template generation phases. Consequently, LOGPRISM's performance is achieved under a significantly stricter measurement standard than the parser-based baselines.

Table III presents the experimental results, which reveal two key conclusions. First, in the direct comparison (single-threaded internal model), LOGPRISM emerges as the fastest compressor, outperforming all baselines across all 16 datasets. With an average speed of 29.87 MB/s, it surpasses the next fastest competitor, Denum (17.83 MB/s), by 1.68×. The superior efficiency stems from LOGPRISM's hierarchical processing strategy. By efficiently discovering and encoding

TABLE II
EXPERIMENTAL RESULTS OF COMPRESSION RATIO

Dataset	gzip	LZMA	bzip2	PPMd	LogZip	LogReducer	LogShrink	Denum	LOGPRISM
Android	7.742	18.857	12.787	19.370	25.165	20.776	21.857	32.494	**32.552**
Apache	21.308	25.186	29.557	31.688	30.375	43.028	55.940	58.517	**64.250**
BGL	12.927	17.637	15.461	18.927	32.655	38.600	42.385	41.804	**46.907**
Hadoop	20.485	36.095	32.598	32.110	35.008	52.830	60.091	**78.546**	77.915
HDFS	10.636	13.559	14.059	19.155	26.666	22.634	**27.319**	25.670	26.410
HealthApp	10.957	13.431	13.843	15.337	12.632	31.694	39.072	44.472	**50.116**
HPC	11.263	15.076	12.756	14.822	27.208	32.070	35.878	45.275	**45.399**
Linux	11.232	16.677	14.695	18.508	23.368	25.213	29.252	30.449	**38.137**
Mac	11.733	22.159	18.074	28.469	26.306	35.251	39.860	40.789	**44.762**
OpenSSH	16.828	18.918	22.865	31.977	42.606	86.699	103.175	101.654	**112.910**
OpenStack	12.158	14.437	15.231	17.429	17.258	16.701	22.157	22.238	**23.267**
Proxifier	15.716	18.982	23.619	25.489	21.493	25.501	27.029	27.288	**29.759**
Spark	17.825	19.908	26.497	30.614	20.825	59.470	**59.739**	59.470	52.496
Thunderbird	16.462	27.309	25.428	33.026	–	49.185	48.434	63.824	**74.252**
Windows	17.798	202.568	67.533	61.083	310.596	342.975	456.301	481.350	**488.020**
Zookeeper	25.979	27.667	36.156	38.931	47.373	94.562	116.981	135.251	**142.313**

frequent "structure + variable" patterns in the second stage, we drastically reduce the volume of data requiring processing by the third stage that is more computationally intensive. Second, the results for LOGPRISM-P validate the significant benefits of our parallel-aware design. By enabling internal, fine-grained parallelism, LOGPRISM-P achieves an average speed of 41.55 MB/s, representing a further 39.1% average speedup over the single-threaded LOGPRISM configuration.

The observed variation in LOGPRISM's speed across datasets is primarily driven by dataset size, which dictates the effectiveness of chunk-level parallelization. For example, on the small Linux dataset (25,567 lines), the volume is insufficient to fill even a single 100K-line processing chunk, precluding the standard multi-threaded acceleration used by all compressors. Despite this constraint, the standard LOGPRISM configuration achieves the best performance (8.55 MB/s) among all baselines. Moreover, LOGPRISM-P provides substantial additional speedup by leveraging the co-design of hierarchical workload reduction and internal fine-grained parallelism. This ensures high efficiency even in edge cases. On the Linux dataset, LOGPRISM-P (10.23 MB/s) is significantly faster than Denum (4.97 MB/s) because its hierarchical strategy minimizes the computational overhead, while the internal worker threads maximize CPU utilization within the single active chunk.

Summary for RQ2: LOGPRISM achieves state-of-the-art end-to-end compression speed. This efficiency stems from the synergistic effect of its hierarchical processing strategy, which minimizes computational overhead, and its parallel-aware architecture, which maximizes resource utilization.

D. RQ3: Ablation Analysis of LOGPRISM*'s Different Stages*

We perform a progressive ablation study to isolate and quantify the contribution of each stage in LOGPRISM's hierarchical design. Specifically, we focus on Stage 2 (Variable Subtree Encoding), which implements our core unified redundancy encoding paradigm. We evaluate three configurations across all 16 datasets: (1) LZMA, a general-purpose compressor serving as a universal baseline; (2) LOGPRISM (S1+S3), a structural baseline that executes Stage 1 and Stage 3 but bypasses Stage 2, representing an advanced parser-based compressor that treats structure and variables as separate entities; and (3) LOGPRISM, the complete model, which is compared against LOGPRISM (S1+S3) to measure the specific performance gain provided by mining "structure + variable" correlations. Both the last two configurations utilize LZMA for the final compression of their output. The results for CR and CS are detailed in Tables IV and V, respectively.

Compression Ratio Analysis: The results demonstrate a clear, step-wise improvement in effectiveness. The LOGPRISM (S1+S3) model is itself a high-performance compressor. By leveraging the structural analysis of Stage 1 and global sorting pipeline of Stage 3, it achieves a 2.48× higher average CR than LZMA. Furthermore, the integration of Stage 2 delivers a decisive performance boost. The full LOGPRISM model achieves an additional 6.82% increase in average CR over the already strong LOGPRISM (S1+S3) variant. This gain is most prominent on datasets with deep variable correlations, such as Android (+27.34%) and Thunderbird (+24.01%). The only exception is the Windows dataset, which shows a very small decrement (-0.57%). This is due to its dominance by flat, disjoint variable patterns (e.g., unique version hashes) where the metadata overhead outweighs the encoding gains. Nonetheless, the full model outperforms the structural baseline on all other datasets. This confirms the validity of our unified redundancy encoding design, i.e., by treating variable combinations as part of a compressible pattern rather than independent random values, LOGPRISM unlocks massive redundancy that the traditional "template vs. variable" dichotomy ignores.

Compression Speed Analysis: The ablation study also reveals the critical role of Stage 2 as a performance accelerator. Despite adding another analysis step, the full LOGPRISM model is 25.93% faster on average than the structurally simpler LOGPRISM (S1+S3) variant. This counter-intuitive result validates our hierarchical data distillation strategy. In the

TABLE III
EXPERIMENTAL RESULTS OF COMPRESSION SPEED (MB/S)

Dataset	**LogZip**	**LogReducer**	**LogShrink**	**Denum**	**LOGPRISM**	**LOGPRISM-P**
Implementation	Python	C++	C++ and Python	C++	C++	C++
Android	0.068	19.323	4.123	24.380	**31.189**	36.672
Apache	0.737	2.347	1.537	6.369	**12.002**	17.551
BGL	0.874	26.738	2.571	23.575	**33.234**	50.625
Hadoop	0.901	12.882	4.401	24.453	**40.539**	52.840
HDFS	0.701	23.598	3.466	25.193	**29.281**	35.288
HealthApp	0.736	7.937	2.754	17.866	**22.239**	29.103
HPC	0.644	9.391	3.599	25.216	**32.257**	37.868
Linux	0.687	1.249	0.941	4.969	**8.550**	10.228
Mac	0.009	5.450	2.141	6.710	**11.715**	16.097
OpenSSH	0.715	14.773	3.335	25.572	**51.369**	70.016
OpenStack	0.537	13.039	4.018	12.352	**20.960**	29.396
Proxifier	0.716	1.328	0.742	6.000	**9.657**	10.585
Spark	0.550	21.233	3.185	26.034	**42.052**	59.005
Thunderbird	–	18.656	4.036	19.830	**38.539**	69.213
Windows	1.357	18.483	6.330	28.783	**79.220**	116.152
Zookeeper	0.842	4.429	2.280	8.000	**15.111**	24.192
Average	0.694	12.554	3.091	17.831	**29.870**	41.552

TABLE IV
ABLATION STUDY OF COMPRESSION RATIO

Dataset	**LZMA**	**LOGPRISM (S1+S3)**	**LOGPRISM**
Android	18.857	25.564	**32.552**
Apache	25.186	61.887	**64.250**
BGL	17.637	46.548	**46.907**
Hadoop	36.095	75.170	**77.915**
HDFS	13.559	24.282	**26.410**
HealthApp	13.431	46.932	**50.116**
HPC	15.076	43.880	**45.399**
Linux	16.677	37.356	**38.137**
Mac	22.159	37.373	**44.762**
OpenSSH	18.918	95.969	**112.910**
OpenStack	14.437	21.868	**23.267**
Proxifier	18.982	26.664	**29.759**
Spark	19.908	50.656	**52.496**
Thunderbird	27.309	59.876	**74.252**
Windows	202.568	**490.787**	488.020
Zookeeper	27.667	118.531	**142.313**

TABLE V
ABLATION STUDY OF COMPRESSION SPEED (MB/S)

Dataset	**LZMA**	**LOGPRISM (S1+S3)**	**LOGPRISM**
Android	24.876	26.717	**36.672**
Apache	7.665	13.453	**17.551**
BGL	16.108	49.072	**50.625**
Hadoop	22.102	44.937	**52.840**
HDFS	15.715	31.145	**35.288**
HealthApp	9.436	23.156	**29.103**
HPC	11.700	33.401	**37.868**
Linux	4.592	8.818	**10.228**
Mac	8.428	12.950	**16.097**
OpenSSH	15.873	46.553	**70.016**
OpenStack	11.647	20.578	**29.396**
Proxifier	6.696	8.750	**10.585**
Spark	20.757	50.734	**59.005**
Thunderbird	26.845	51.360	**69.213**
Windows	63.706	91.402	**116.152**
Zookeeper	6.581	14.907	**24.192**
Average	17.045	32.996	**41.552**

LOGPRISM (S1+S3) configuration, every variable identified in Stage 1 must be processed by Stage 3, which relies on computationally expensive regex templatization. In contrast, the full LOGPRISM model only supplies Stage 3 with a smaller set of "true long-tail residuals." By handling the majority of high-frequency variables in the efficient Stage 2, LOGPRISM drastically reduces the workload of the most time-consuming part of the pipeline, thereby lowering overall computational overhead and increasing throughput.

Summary for RQ3: The ablation study confirms that LOGPRISM's performance breakthrough is driven by its hierarchical design. While Stages 1 and 3 provide a solid structural baseline, the core innovation (i.e., Stage 2 Variable Subtree Encoding) delivers a decisive performance boost to both compression ratio and speed.

E. RQ4: Robustness Analysis: The Impact of Data Granularity

This question investigate the impact of data granularity on LOGPRISM's performance, which is a critical operational parameter. Our primary experiments utilized a fixed chunk size of 100K lines, a standard baseline setting that maximizes parallelism but limits pattern discovery to local windows. This design involves an inherent trade-off: each chunk builds an independent URT, preventing the sharing of patterns across the full dataset. This can potentially obscure very long-range redundancies. To explore this speed-vs-compression trade-off, we configure LOGPRISM to operate in single-archive (non-chunked) mode, allowing it to construct a single global URT over the entire dataset. We compare this configuration against Denum, the leading state-of-the-art baseline that also supports

TABLE VI
ROBUSTNESS ANALYSIS OF SINGLE-ARCHIVE MODE PERFORMANCE

Dataset	Compression Ratio		Speed (MB/s)	
	Denum	LOGPRISM	Denum	LOGPRISM
Android	46.191	**52.429**	5.416	**15.294**
Apache	58.562	**64.269**	5.810	**12.369**
BGL	44.284	**47.429**	6.100	**16.345**
Hadoop	**92.899**	90.590	8.643	**27.186**
HDFS	**32.919**	32.198	5.230	**7.956**
HealthApp	47.279	**53.574**	7.072	**12.728**
HPC	45.542	**46.350**	6.369	**16.276**
Linux	30.688	**38.216**	4.582	**8.456**
Mac	43.633	**50.266**	5.245	**11.171**
OpenSSH	106.984	**120.116**	7.426	**27.962**
OpenStack	21.898	**23.394**	5.129	**12.576**
Proxifier	27.141	**29.739**	5.484	**9.894**
Spark	**65.125**	56.196	5.398	**13.237**
Thunderbird	70.161	**85.097**	4.263	**15.900**
Windows	3407.766	**4174.176**	6.421	**26.897**
Zookeeper	135.846	**142.290**	7.556	**18.013**
Average	267.307	**319.146**	6.009	**15.766**

global operation. The results, presented in Table VI, demonstrate that global pattern discovery significantly boosts LOGPRISM's compression effectiveness while maintaining highly competitive speeds.

Compression Ratio Analysis: In single-archive mode, LOGPRISM achieves a 278.40% increase in average CR over its own chunked performance, with the gain on the Windows dataset reaching a massive 755.33%. When compared to Denum operating in the same global configuration, LOGPRISM outperforms it on 13 of the 16 datasets. The improvements are particularly significant on complex logs such as Windows (+22.49%) and Thunderbird (+21.29%). On two of the remaining three datasets (Hadoop, HDFS), LOGPRISM remains highly competitive, following Denum with a small difference of less than 2.50%. The only notable exception is the Spark dataset, where Denum's specialized handling of specific numeric sequences proves more efficient. Overall, these results validate that LOGPRISM scales effectively to leverage global context to achieve superior compression.

Compression Speed Analysis: LOGPRISM maintains an absolute speed advantage even when building a single global model, whose average speed of 15.77 MB/s is 2.62× faster than Denum (6.01 MB/s). This sustained high performance is attributable to LOGPRISM's hybrid parallel architecture. While standard chunk-based compressors (like Denum) lose parallelism when processing a single global block, LOGPRISM retains fine-grained concurrency: the pre-processing of logs in Stage 1 and the subtree construction for distinct structural groups in Stage 2 continue to execute in parallel. Furthermore, the architectural benefit identified in RQ3, where Stage 2 acts as a high-throughput filter to reduce the workload of Stage 3, remains fully effective. Particularly, even in this globally optimized mode, LOGPRISM remains significantly faster than the default chunked configurations of LogReducer (12.55 MB/s) and LogShrink (3.09 MB/s).

Summary for RQ4: LOGPRISM's design offers both robustness and flexibility, providing users with a clear speed-vs-compression trade-off: the default chunked mode maximizes speed, while the single-archive mode maximizes the compression ratio at a reasonable efficiency cost. In both operational paradigms, LOGPRISM's performance advantages significantly surpass baseline models.

V. RELATED WORK

A. General-purpose Compression Algorithms

General-purpose compressors provide a baseline for evaluating specialized methods. These algorithms can be categorized into three primary families. Dictionary-based approaches, exemplified by LZMA, achieve compression by replacing repeated byte sequences with references to a dictionary. Prediction-based methods, such as PPMd, leverage statistical models to encode characters based on their preceding context. Block-sorting algorithms, notably bzip2, utilize the Burrows-Wheeler Transform to cluster similar characters, thereby enhancing subsequent encoding efficiency.

While these methods exhibit strong performance on generic text data, they fundamentally operate on undifferentiated byte streams without awareness of log structure. Consequently, their effectiveness diminishes significantly when confronted with logs containing high-entropy variables such as unique request IDs and dynamic numerical values, as they cannot exploit the inherent semi-structured redundancy that spans log entries.

B. Log-Specific Compression Methods

1) Parser-Based Approaches: The parser-based paradigm, which represents the dominant methodology in contemporary log compression research, follows a two-stage workflow. In the first stage, a log parser analyzes the input to extract fixed templates and separate variable parameters. Templates are typically inferred from statistical analysis of a representative log sample. In the second stage, these components are compressed independently, with template identifiers being dictionarized and parameter sequences processed separately.

This paradigm has undergone continuous refinement. LogZip [24] established the foundational framework, utilizing the Drain parser combined with an iterative clustering algorithm for pattern mining. LogReducer [25] advanced the field by introducing specialized techniques for numerical parameters, including delta encoding for timestamps, correlation identification, and elastic encoding schemes. LogShrink [26] extended this analysis to encompass all parameter types, employing longest common subsequence techniques and entropy-based analysis to identify commonality and variability patterns. Beyond compression effectiveness, several systems have integrated query capabilities into the compressed representation. CLP [14] and LogGrep [41] enable efficient search operations directly on compressed log data, demonstrating the potential for compression methods to support downstream analytical tasks.

However, the parser-based paradigm carries an inherent limitation: the rigid separation of templates and parameters occurs early in the processing pipeline. This irreversible categorization can destroy contextual correlations that span structural and variable tokens, potentially limiting compression effectiveness when these cross-boundary patterns dominate the redundancy structure.

2) Non-Parser-Based Approaches: Non-parser-based methods eschew explicit structural extraction, instead applying unified compression strategies to entire log lines. Early work in this category includes LogArchive [45], which groups similar logs using similarity functions, and MLC [46], which identifies redundancy through block-level deduplication. The most recent advancement in this category is the number-centric paradigm, exemplified by Denum [27]. Rather than parsing logs into templates and parameters, Denum performs a global binary classification of all tokens into numeric and non-numeric categories. It then applies specialized compression techniques optimized for each category, leveraging the arithmetic properties of numeric sequences while handling non-numeric strings as independent streams.

While this approach avoids the template inference overhead of parser-based methods, it still performs an early, global token categorization. This design choice can lead to the loss of correlation information that crosses the numeric/non-numeric boundary, particularly when the redundancy structure involves co-occurring patterns of both types.

Our work introduces a fundamentally different paradigm that unifies structural extraction and pattern encoding through hierarchical redundancy mining. Rather than pre-committing to fixed categories, our approach progressively distills log data through multiple stages, prioritizing the encoding of high-frequency aggregate patterns that preserve full contextual information. This allows the compression of entire "structure+variable" sequences with single identifiers, exploiting deep redundancies in log data.

VI. Conclusion

This paper reevaluates the prevailing "parse-then-compress" paradigm in log storage, identifying the rigid decoupling of structure extraction and data encoding as a fundamental bottleneck. Our empirical analysis confirms that high parsing accuracy does not guarantee compression efficiency. Instead, it often obscures deep "template-variable" and inter-variable correlations essential for maximizing storage density. To address this limitation, we introduce LogPrism, a framework that resolves this misalignment through Unified Redundancy Encoding. By dynamically modeling log structure and variable patterns within a Unified Redundancy Tree (URT), LogPrism effectively bridges the gap between static templates and dynamic parameters. Leveraging a hierarchical redundancy mining strategy and fine-grained parallelism, our approach simultaneously optimizes for compression ratio and throughput. Extensive experiments on 16 benchmark datasets validate this design, with LogPrism achieving state-of-the-art compression ratios on 13 datasets and delivering compression speeds 1.68$\times$ faster than the nearest competitor. These findings demonstrate that co-designing parsing and compression is critical for unlocking the full potential of log data reduction in large-scale systems.

Data Availability

The source code of LogPrism is publicly available on https://github.com/Lycc42/LogPrism.

References

[1] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*, pp. 207–218, IEEE Computer Society, 2016.

[2] Z. Chen, J. Liu, W. Gu, Y. Su, and M. R. Lyu, "Experience report: Deep learning-based system log analysis for anomaly detection," *CoRR*, vol. abs/2107.05908, 2021.

[3] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 1285–1298, ACM, 2017.

[4] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel* (J. Fürnkranz and T. Joachims, eds.), pp. 37–46, Omnipress, 2010.

[5] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010* (J. C. Hoe and V. S. Adve, eds.), pp. 143–154, ACM, 2010.

[6] W. Yuan, S. Lu, H. Sun, and X. Liu, "How are distributed bugs diagnosed and fixed through system logs?," *Inf. Softw. Technol.*, vol. 119, 2020.

[7] J. Pu, Y. Li, Z. Chen, J. Liu, Z. Jiang, J. Chen, R. Shi, Z. Zheng, and T. Zhang, "Errorprism: Reconstructing error propagation paths in cloud service systems," *CoRR*, vol. abs/2509.26463, 2025.

[8] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019* (M. Dumas, D. Pfahl, S. Apel, and A. Russo, eds.), pp. 683–694, ACM, 2019.

[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 Comput. Surv.*, vol. 54, no. 6, pp. 130:1–130:37, 2022.

[10] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu, Y. Dang, F. Gao, P. Zhao, B. Qiao, Q. Lin, D. Zhang, and M. R. Lyu, "Towards intelligent incident management: why we need it and how we make it," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020* (P. Devanbu, M. B. Cohen, and T. Zimmermann, eds.), pp. 1487–1497, ACM, 2020.

[11] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (K. Keeton and T. Roscoe, eds.), pp. 603–618, USENIX Association, 2016.

[12] R. Ding, H. Zhou, J. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *Proceedings of the 2015 USENIX Annual Technical Conference, USENIX ATC 2015, July 8-10, Santa Clara, CA, USA* (S. Lu and E. Riedel, eds.), pp. 139–150, USENIX Association, 2015.

[13] Z. Chen, J. Pu, and Z. Zheng, "Tracezip: Efficient distributed tracing via trace compression," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, pp. 411–433, 2025.

[14] K. Rodrigues, Y. Luo, and D. Yuan, "CLP: efficient and scalable search on compressed text logs," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021* (A. D. Brown and J. R. Lorch, eds.), pp. 183–198, USENIX Association, 2021.

[15] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, “The mystery machine: End-to-end performance analysis of large-scale internet services,” in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014* (J. Flinn and H. Levy, eds.), pp. 217–231, USENIX Association, 2014.

[16] S. Yang, S. J. Park, and J. K. Ousterhout, “Nanolog: A nanosecond scale logging system,” in *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (H. S. Gunawi and B. C. Reed, eds.), pp. 335–350, USENIX Association, 2018.

[17] R. Wang, D. Gibson, K. Rodrigues, Y. Luo, Y. Zhang, K. Wang, Y. Fu, T. Chen, and D. Yuan, “μslope: High compression and fast search on semi-structured logs,” in *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (A. Gavrilovska and D. B. Terry, eds.), pp. 529–544, USENIX Association, 2024.

[18] P. Deutsch, “DEFLATE compressed data format specification version 1.3,” *RFC*, vol. 1951, pp. 1–17, 1996.

[19] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[20] K. Yao, H. Li, W. Shang, and A. E. Hassan, “A study of the performance of general compressors on log files,” *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 3043–3085, 2020.

[21] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, “Tools and benchmarks for automated log parsing,” in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019* (H. Sharp and M. Whalen, eds.), pp. 121–130, IEEE / ACM, 2019.

[22] Z. Jiang, J. Liu, J. Huang, Y. Li, Y. Huo, J. Gu, Z. Chen, J. Zhu, and M. R. Lyu, “A large-scale evaluation for log parsing techniques: How far are we?,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024* (M. Christakis and M. Pradel, eds.), pp. 223–234, ACM, 2024.

[23] 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 2017, Honolulu, HI, USA, June 25-30, 2017* (I. Altintas and S. Chen, eds.), pp. 33–40, IEEE, 2017.

[24] 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 *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pp. 863–873, IEEE, 2019.

[25] 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 2021, February 23-25, 2021* (M. K. Aguilera and G. Yadgar, eds.), pp. 249–262, USENIX Association, 2021.

[26] X. Li, H. Zhang, V. Le, and P. Chen, “Logshrink: Effective log compression by leveraging commonality and variability of log data,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pp. 23:1–23:12, ACM, 2024.

[27] S. Yu, Y. Wu, Y. Li, and P. He, “Unlocking the power of numbers: Log compression via numeric token parsing,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024* (V. Filkov, B. Ray, and M. Zhou, eds.), pp. 919–930, ACM, 2024.

[28] Z. A. Khan, D. Shin, D. Bianculli, and L. C. Briand, “Impact of log parsing on deep learning-based anomaly detection,” *Empir. Softw. Eng.*, vol. 29, no. 6, p. 139, 2024.

[29] Z. A. Khan, D. Shin, D. Bianculli, and L. C. Briand, “Guidelines for assessing the accuracy of log message template identification techniques,” in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 1095–1106, ACM, 2022.

[30] Z. Li, C. Luo, T. Chen, W. Shang, S. He, Q. Lin, and D. Zhang, “Did we miss something important? studying and exploring variable-aware log abstraction,” in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pp. 830–842, IEEE, 2023.

[31] Y. Fu, M. Yan, J. Xu, J. Li, Z. Liu, X. Zhang, and D. Yang, “Investigating and improving log parsing in practice,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022* (A. Roychoudhury, C. Cadar, and M. Kim, eds.), pp. 1566–1577, ACM, 2022.

[32] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, “Abstracting execution logs to execution events for enterprise applications (short paper),” in *Proceedings of the Eighth International Conference on Quality Software, QSIC 2008, 12-13 August 2008, Oxford, UK* (H. Zhu, ed.), pp. 181–186, IEEE Computer Society, 2008.

[33] A. Makanju, N. Zincir-Heywood, and E. E. Milios, “Clustering event logs using iterative partitioning,” in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009* (J. F. E. IV, F. Fogelman-Soulié, P. A. Flach, and M. J. Zaki, eds.), pp. 1255–1264, ACM, 2009.

[34] M. Nagappan and M. A. Vouk, “Abstracting log lines to log event types for mining software system logs,” in *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings* (J. Whitehead and T. Zimmermann, eds.), pp. 114–117, IEEE Computer Society, 2010.

[35] L. Tang, T. Li, and C. Perng, “Logsig: generating system events from raw textual logs,” in *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011* (C. Macdonald, I. Ounis, and I. Ruthven, eds.), pp. 785–794, ACM, 2011.

[36] S. Messaoudi, A. Panichella, D. Bianculli, L. C. Briand, and R. Sasnauskas, “A search-based approach for accurate identification of log message formats,” in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018* (F. Khomh, C. K. Roy, and J. Siegmund, eds.), pp. 167–177, ACM, 2018.

[37] M. Mizutani, “Incremental mining of system log format,” in *2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, pp. 595–602, IEEE Computer Society, 2013.

[38] M. Du and F. Li, “Spell: Streaming parsing of system event logs,” in *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain* (F. Bonchi, J. Domingo-Ferrer, R. Baeza-Yates, Z. Zhou, and X. Wu, eds.), pp. 859–864, IEEE Computer Society, 2016.

[39] G. Yu, P. Chen, P. Li, T. Weng, H. Zheng, Y. Deng, and Z. Zheng, “Logreducer: Identify and reduce log hotspots in kernel on the fly,” in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pp. 1763–1775, IEEE, 2023.

[40] J. Chen, “Log compression via redundancy eliminating at word and numerical levels,” in *IEEE International Conference on Software Services Engineering, SSE 2024, Shenzhen, China, July 7-13, 2024*, pp. 115–122, IEEE, 2024.

[41] J. Wei, G. Zhang, J. Chen, Y. Wang, W. Zheng, T. Sun, J. Wu, and J. Jiang, “Loggrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns,” in *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023* (G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, eds.), pp. 452–468, ACM, 2023.

[42] L. Zhang, Z. Xie, V. Anand, Y. Vigfusson, and J. Mace, “The benefit of hindsight: Tracing edge-cases in distributed systems,” in *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023* (M. Balakrishnan and M. Ghobadi, eds.), pp. 321–339, USENIX Association, 2023.

[43] Z. Chen, Z. Jiang, Y. Su, M. R. Lyu, and Z. Zheng, “Tracemesh: Scalable and streaming sampling for distributed traces,” in *17th IEEE International Conference on Cloud Computing, CLOUD 2024, Shenzhen, China, July 7-13, 2024* (R. N. Chang, C. K. Chang, J. Yang, N. L. Atukorala, Z. Jin, M. Sheng, J. Fan, K. Fletcher, Q. He, T. Kosar, S. Sarkar, S. Venkateswaran, S. Wang, X. Liu, S. Seelam, C. Narayanaswami, and Z. Zong, eds.), pp. 54–65, IEEE, 2024.

[44] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *IEEE Trans. Inf. Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[45] R. Christensen and F. Li, “Adaptive log compression for massive log data,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (K. A. Ross, D. Srivastava, and D. Papadias, eds.), pp. 1283–1284, ACM, 2013.

[46] 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, Tianjin, China, August 23-26, 2016*, pp. 1358–1365, IEEE, 2016.