LogNexus: Effective Log Compression via Unified Redundancy Encoding

YANG LIU, Sun Yat-sen University, China
KAIMING ZHANG, Sun Yat-sen University, China
ZHUANGBIN CHEN*, Sun Yat-sen University, China
ZIBIN ZHENG, Sun Yat-sen University, China

State-of-the-art log compressors typically rely on a decoupled "*parse-then-compress*" workflow, where parsing is optimized for semantic accuracy (i.e., event identification) rather than storage efficiency. Through a comprehensive empirical study, we reveal that this architectural decoupling prevents the exploitation of deep correlations between static templates and dynamic variables. To address it, we propose LogNexus based on the principle of *unified redundancy encoding*, a new log compression paradigm that co-designs structural extraction and variable encoding. LogNexus constructs a *Unified Redundancy Tree (URT)* using a hierarchical strategy that progressively mines frequent "structure+variable" patterns in logs. Such a design captures deep contextual redundancies ignored by traditional methods while minimizing computational overhead by pre-emptively encoding dominant patterns. Extensive evaluation on 16 benchmark datasets demonstrates that LogNexus establishes a new state-of-the-art. It achieves the highest compression ratio on 14 datasets (outperforming baselines by 9.48%∼89.13%) and the fastest speed (1.51×∼40.06× faster than competitors). Furthermore, when configured in non-chunked mode to maximize global pattern discovery, LogNexus boosts its compression ratio by 285.13%, which is 27.08% higher than the best baseline, while retaining a 2.43× speed advantage. The decompression audit further shows that LogNexus successfully restores every token on all 16 datasets.

CCS Concepts: • **Information systems → Information storage technologies**; • **Software and its engineering → Software reliability**; **Software maintenance tools**.

Additional Key Words and Phrases: Information Redundancy, Log Compression, Log Analysis

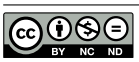
ACM Reference Format:
Yang Liu, Kaiming Zhang, Zhuangbin Chen, and Zibin Zheng. 2026. LogNexus: Effective Log Compression via Unified Redundancy Encoding. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA172 (October 2026), 23 pages. https://doi.org/10.1145/3832263

1 Introduction

Software systems generate logs to record runtime events, errors, and operational states, which are indispensable for system maintenance [3, 4, 12, 17, 30, 36, 41, 42, 44] and performance optimization [2, 6, 16, 43]. However, the sheer volume of log data poses significant storage challenges [5, 10]. Modern large-scale systems can produce terabytes or even petabytes of logs daily [7, 31, 37]. Thus, efficient log compression has become a critical concern for managing storage costs while preserving the analytical value of historical log data [1, 31, 33].

*Zhuangbin Chen is the corresponding author.

Authors' Contact Information: Yang Liu, Sun Yat-sen University, Zhuhai, China, liuy2355@mail2.sysu.edu.cn; Kaiming Zhang, Sun Yat-sen University, Zhuhai, China, zhangkm7@mail2.sysu.edu.cn; Zhuangbin Chen, Sun Yat-sen University, Zhuhai, China, chenzhb36@mail.sysu.edu.cn; Zibin Zheng, Sun Yat-sen University, Zhuhai, China, zhzibin@mail.sysu.edu.cn.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA172
https://doi.org/10.1145/3832263

Logs typically follow a "template+variables" pattern, where static strings are interleaved with dynamic runtime parameters. General compression algorithms like gzip [9] and LZMA [46] fail to exploit the specific characteristics of log data, often resulting in suboptimal compression ratios [38]. 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 [18, 45] 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 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. Such early splitting is not unique to parser-based pipelines. Recent number-centric designs such as Denum [39] likewise route numeric and non-numeric tokens through separate paths, leaving cross-boundary co-occurrences invisible.

To overcome these limitations, we propose LogNexus, a log compression framework that unifies structural extraction and variable encoding. We define *unified redundancy encoding* as a compression paradigm that represents recurring log patterns by jointly encoding stable structural tokens and correlated variable values, rather than separating them into independent template and variable streams. For example, as illustrated by the `sshd` entries in Fig. 2 (L5–L7), the structural context `authentication failure;` repeatedly co-occurs with variable values such as `user=test1` and `uid=509`; parser-based compressors typically encode this case as one template ID plus separate variable values, whereas LogNexus can encode the recurring structure-variable combination as one pattern. Instead of relying on a pre-defined parsing stage, LogNexus constructs a *Unified Redundancy Tree (URT)* that models log structure and variable correlations together. 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" patterns, effectively bridging the gap between templates and parameters. Finally, we 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. By filtering out frequent patterns early, LogNexus drastically reduces the load on the expensive residual stage.

Extensive evaluation on 16 datasets from LogHub [45] shows that LogNexus establishes a new state-of-the-art in both effectiveness and efficiency. It achieves the highest compression ratio on 14 datasets (outperforming baselines by 9.48%~89.13%) and the fastest speed (1.51×~40.06× faster than competitors). Furthermore, we address a critical trade-off inherent in baselines, which rely on chunked processing to enable parallelism with limited global pattern discovery. When in non-chunked mode, LogNexus improves its compression ratio by 285.13%, 27.08% higher than Denum in the same mode. Remarkably, LogNexus's speed in this computationally intensive mode (14.58 MB/s)

remains comparable to Denum's default chunked performance (17.83 MB/s). For decompression, LogNexus successfully restores all tokens across all datasets, whereas the baselines yield incorrect or incomplete restorations on a substantial portion of these datasets.

In summary, this paper makes the following contributions:

- We conduct the first 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 LogNexus, an effective log compression framework featuring a unified redundancy tree and a parallel-aware architecture.
- We perform an extensive evaluation demonstrating that LogNexus significantly outperforms existing state-of-the-art methods in both compression ratio and speed.

2 Background and Motivation

2.1 Log Compression Techniques

Log compression techniques can be broadly grouped into *general-purpose compressors* (e.g., gzip [9] and LZMA [46]), which treat logs as opaque byte streams, and *log-specific compressors*, which first transform logs into a more compressible representation before invoking a general-purpose backend. The latter category spans parser-based compressors that decompose logs into templates and variables, search-oriented stores [31, 34], similarity-based methods [8, 13], and the recent number-centric paradigm [39]. A more detailed discussion of existing approaches is given in Sec. 5.

Among them, parser-based methods represent the predominant approach, which leverage the semi-structured nature of logs. The process begins with a critical prerequisite step, *log parsing*, where parsers decompose raw log messages into invariant templates and corresponding variables. This structured representation enables efficient compression: repetitive template text is replaced with compact identifiers (stored once in a dictionary), and extracted parameters are encoded using specialized techniques. The resulting stream of template IDs and parameter arrays, with significantly reduced entropy, is then processed by general algorithms like LZMA or gzip. Several representative compressors [22, 24, 35, 39] exemplify different design philosophies within this framework.

The performance of these compressors is fundamentally tied to the outcome of the parsing stage [21, 39]. However, existing research mainly focuses on compression algorithms, with an assumption of ideal parsing quality. Parsers are often treated as independent components rather than integral parts of the compression pipeline. Yet, there is no one-size-fits-all parser. Parsers employ diverse heuristics and algorithms with accuracy varying significantly across datasets [14, 18, 20, 23]. Any parsing errors will inevitably propagate and undermine the effectiveness of the downstream compression algorithms. Moreover, their common reliance on sampling for template extraction precludes comprehensive template coverage and introduces performance variance determined by sample quality. Despite these critical limitations, how parser-produced representations affect compression efficiency remains insufficiently understood. To systematically evaluate how different log parsers impact compression performance, we conduct a comprehensive empirical study.

2.2 Revisiting Log Parsers in Compression Pipelines

2.2.1 Experiment Setup. We introduce the experiment setup of our study as follows:

Log Parser Selection: We select nine representative parsers, i.e., Drain [15], AEL [19], IPLoM [26], LFA [29], LogSig [32], MoLFI [27], SHISO [28], Spell [11], and LogReducer's parser [35], covering a broad spectrum of algorithmic strategies. Our focus on mostly classical parsers reflects common practice: existing parser-based compressors are commonly built on off-the-shelf parsers or embedded variants with similar template/variable outputs. Their outputs therefore represent the

front-end decompositions that current compressors actually receive. The selected parsers span seven methodological families, which helps us generate heterogeneous representations for studying downstream compression behavior. AEL represents the heuristic approach using lightweight rules to distinguish static text from dynamic parameters. Drain and LogReducer employ fixed-depth parsing trees where root-to-leaf paths define templates. Some parsers frame the task as a data mining problem. LFA applies frequent pattern mining while IPLoM iteratively partitions log messages based on token position and cardinality. LogSig and SHISO use clustering by computing pairwise message similarity. Other novel designs are included. Spell identifies templates via longest common subsequence, and MoLFI leverages evolutionary algorithms to evolve optimal template sets.

Log Compressor Selection: Following existing log compression studies, we select four state-of-the-art compressors, i.e., LogZip [24], LogReducer [35], LogShrink [22], and Denum [39]. These approaches feature a different design in log structure extraction and encoding strategies. LogZip, a pioneer in parser-based compression, uses 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 select compact bit representations, complemented by delta encoding for sequential data. LogShrink optimizes parameter storage by reorganizing data into columnar formats, grouping homogeneous types to create low-entropy streams ideal for dictionary encoding [47]. Denum introduces a number-centric approach that targets numerical data, using regex to isolate numeric tokens for categorization and differential encoding. For this empirical study, we use its optional mode, in which the remaining non-numeric content is processed by a parser-based compressor (i.e., LogShrink) after the numeric tokens are extracted and encoded. Consequently, parser choice can affect Denum's final compression ratio under this setting.

Dataset Selection: Our study employs the widely-used LogHub benchmark [45], which comprises 16 datasets spanning a broad spectrum of system types, e.g., large-scale distributed systems, supercomputers. In total, the full benchmark contains around 378 million log entries (over 77 GB). However, our experiments revealed that the Android and Windows datasets could not be processed by certain parsers within a reasonable time budget due to their inherent complexity. Thus, our 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, defined as $\text{CR} = \frac{Original\ File\ Size}{Compressed\ File\ Size}$. A higher value indicates better compression performance.

2.2.2 Experimental Results. Fig. 1 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.

Observation 1: Dramatic Performance Variance Induced by Parser Selection. The choice of parser induces dramatic variance in compression ratios, often surpassing the inherent performance differences between compressors themselves. 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 only 44.81 with the IPLoM parser's templates. Denum, which is not fully parser-based, also demonstrates 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, compressors can achieve higher compression ratios using external parsers rather than their built-in ones. On BGL, LogReducer achieves a CR of 38.04 when using its own default parser, but this increases to

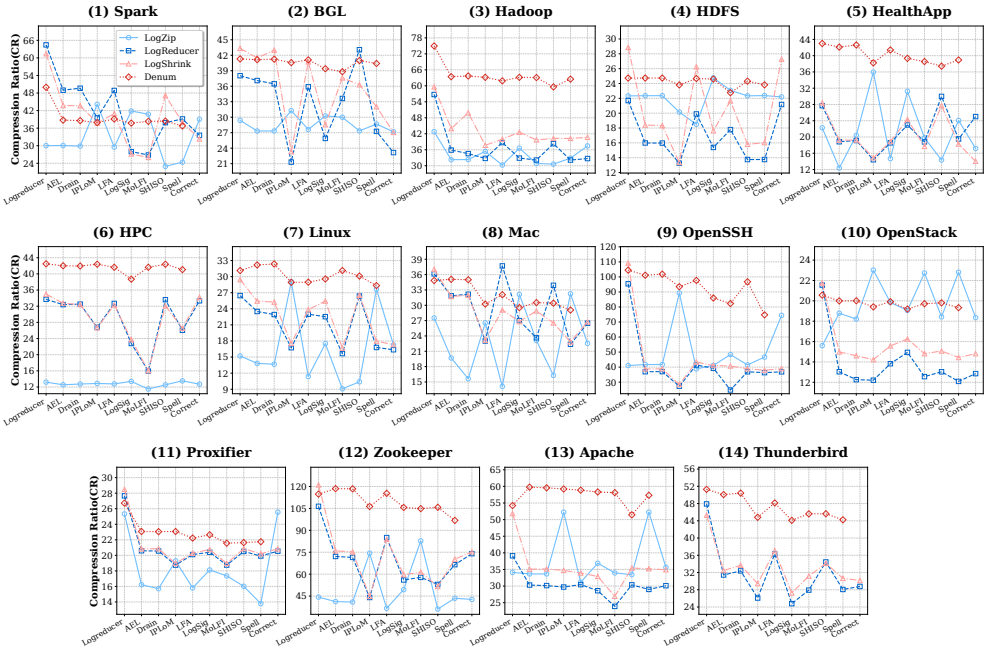


Fig. 1. Impact of Log Parser Selection on Downstream Compression Ratio across Different Datasets

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). This suggests 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. There is no universally optimal parser that consistently maximizes 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 depends on the complex interplay between data characteristics and the compressor's encoding strategy. The root cause lies in a fundamental misalignment of objectives: parsers prioritize classification accuracy to maximize event clustering correctness, whereas compressors prioritize storage efficiency. This leads to two typical inefficiencies that degrade compression performance despite high parsing accuracy. *Over-generalization (coarse-grained templates)* occurs when parsers prioritizing high matching rates introduce excessive wildcards. On HDFS, LFA generates only 42 templates with 372 wildcards (8.8 per template). While this may yield high parsing accuracy, it forces compressors to process large, noisy parameters that should have been static text. Conversely, *over-fitting (fine-grained templates)* occurs when overly sensitive clustering thresholds partition semantically similar logs into numerous distinct templates. For instance, AEL produces over 28,000 templates for HealthApp, which contains only ~150 actual event types. This "template explosion" exponentially increases dictionary size, creating direct storage overhead.

In summary, our empirical study confirms that log parsing is not a mere preprocessing step but a dominant factor in log compression. The fundamental principle of exploiting log structural redundancy is sound, which enables parser-based approaches to significantly outperform general

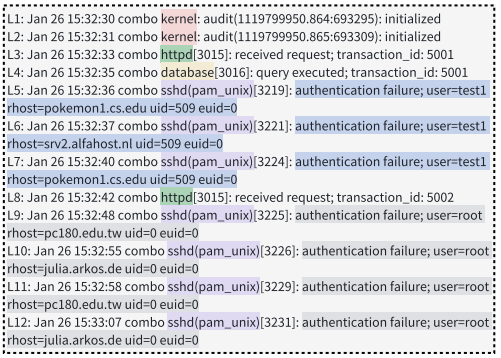


Fig. 2. A 12-line Log Snippet as a Running Example

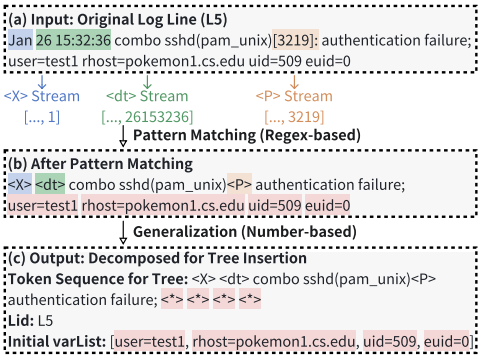


Fig. 3. Pre-processing Pipeline for Log Entry L5

compression algorithms. However, our findings reveal a critical flaw in its 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. Thus, the study is not intended to optimize parsing accuracy itself, but to show that the representation before backend compression determines which structural and variable redundancies remain visible.

2.3 A Motivating Example for Unified Redundancy Encoding

We illustrate the core concept of our approach based on the logs in Fig. 2. The idea is to move beyond the rigid dichotomy of "static strings vs. dynamic variables" by modeling log tokens uniformly and encoding 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*.

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 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*. 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`) 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 [22, 39] 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 template context. Thus, we aggregate both parts as "`...authentication failure; user=test1 rhost=pokemon1.cs.edu uid=509 euid=0`" and encodes it with a single ID. Similar 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 independent handling. The advantage of this unified redundancy encoding is significant: whereas existing approaches require one template ID plus three variable IDs, we need only a single ID for 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 naive implementation of this idea is to build a prefix tree (trie) over the sequences of log tokens and assign IDs to frequent paths, thereby identifying recurring log (sub)sequences. However, this

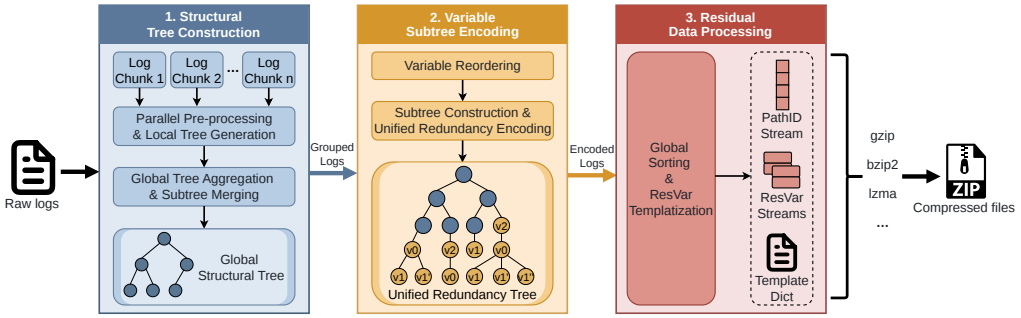


Fig. 4. Overview of the LogNexus Compression Framework

presents two fundamental limitations. First, high-cardinality fields induce many branches, causing node count to grow with the product of distinct values per position. This results in prohibitive storage and lookup latency due to structural inefficiency. Second, a trie only encodes prefixes. If an infrequent token appears early, 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 LogNexus, an effective log compressor which moves beyond pure prefix matching and supports compact, gap-tolerant aggregation of frequent token sequences.

3 Methodology

In this section, we introduce LogNexus, our log compression framework that constructs a *Unified Redundancy Tree (URT)* to jointly model log structure and variable correlations. The URT is the prefix-tree representation that realizes unified redundancy encoding: stable-token paths provide structural contexts, and variable subtrees extend these contexts with frequent correlated variable values. To avoid excessive branching in naive prefix trees, LogNexus employs a hierarchical redundancy mining strategy to build a structurally efficient URT. As shown in Fig. 4, 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*. The first stage builds the structural skeleton of the URT based on regular, fixed tokens, where each path groups structurally similar log entries. The second stage integrates frequent variables by expanding terminal nodes into subtrees. Within each subtree, we mine deep correlations between the structural skeleton and variables, allowing a single ID to represent recurring "structure-variable" patterns. The third stage handles remaining "long-tail" tokens (e.g., random variables) via specialized encoding 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.

Motivated by the decomposition limitation discussed in Sec. 2, LogNexus's hierarchical compression pipeline is fundamentally different from prior designs that perform an early irreversible split before encoding redundancy. Parser-based compressors separate log templates from variables and encode template identifiers and variable streams independently. Denum, while bypassing traditional parsing for numerical fields, still separates numeric and non-numeric content into different encoding paths. Once such a boundary is introduced, recurring cross-boundary patterns, e.g., a specific variable value that repeatedly appears under a specific structure, cannot be encoded as one unit. In contrast, LogNexus keeps stable tokens and frequent correlated variable values in the same URT path, making the compression unit a complete structure-variable pattern rather than isolated template or variable components.

3.1 Structural Tree Construction

This stage constructs the foundational skeleton of the URT by extracting stable patterns from the raw logs. To ensure high efficiency, 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. As depicted in Fig. 4, this phase proceeds through two 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 Subtree Merging*, which integrates these local trees into a unified global structure while dynamically refining the topology.

3.1.1 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 excessive branching caused by highly diverse tokens, we identify and separate two categories of rapidly-changing tokens.

- *Patterned Metadata Fields*: This category includes common header fields like dates, timestamps, and PIDs [45]. They are highly variable but exhibit predictable syntactical patterns. We identify them using regular expressions and extract their values into separate, highly compressible columnar streams, preserving global log order. For fields extracted by regular expressions, LogNexus uses pattern-specific placeholders whose decoders preserve the format information needed to restore the token form, such as the original widths of timestamp components. This avoids the common leading-zero problem in numeric extraction, where values such as `09` and `9` become indistinguishable if both are stored only as integers. As shown in Fig. 3(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).
- *Dynamic Numeric Tokens*: This category targets tokens within log contents that are likely variables. We apply a heuristic [39] to treat any token containing numeric characters as a potential variable. These tokens are replaced by the wildcard $\langle * \rangle$ (Fig. 3), with their original values preserved in a `varList`. Although this approach may misclassify static tokens containing numbers (e.g., `node1`) or string-only variables (e.g., `user=root`), such issues are resolved during the correlation mining stage (Sec. 3.2) and the subsequent subtree merging step, respectively.

After pre-processing, the resulting log messages are used to construct a local prefix tree. For example, dividing the logs from Fig. 2 into two chunks (L1-L6 and L7-L12) yields the independent trees shown in Fig. 5(a) and Fig. 5(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 root-to-terminal path uniquely identifies a group of log entries sharing the same structural pattern. To index these groups, a `Record` object is maintained at every terminal node, aggregating Line IDs (`Lid`) and `varList`(s) for all corresponding logs.

3.1.2 Global Tree Aggregation and 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. 5(c), this 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 metadata streams extracted by individual worker threads are appended to global files, preserving the original log order. For numerical streams like $\langle dt \rangle$, we apply a specialized compression pipeline 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

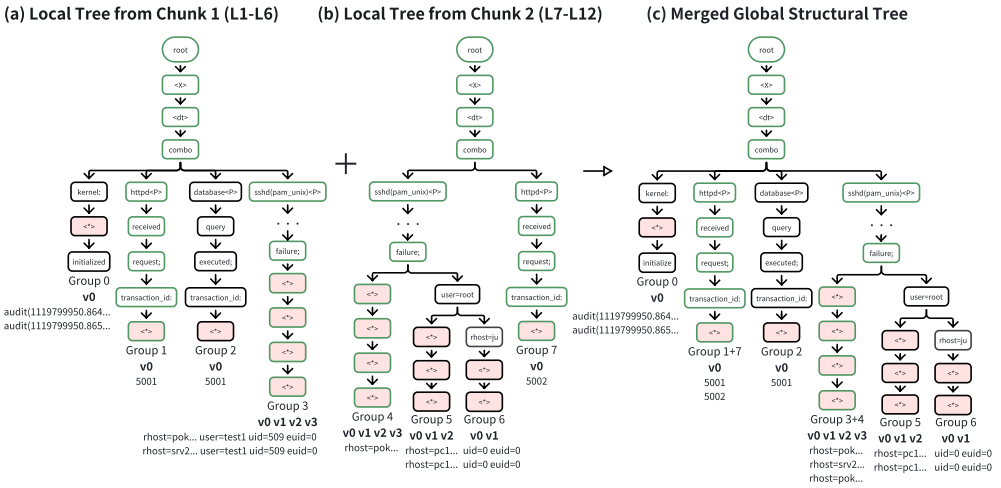


Fig. 5. The Parallel Pre-processing and Streaming Merge Process of the Structural Tree

generalized to ⟨∗⟩ (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 *subtree merging*. It operates on the principle that if a static subtree and a wildcard subtree have the same descendants, their root labels serve the same structural role and should be treated uniformly as variables.

The merging employs a post-order (bottom-up) traversal of the global tree. At each branching node, we compute a unique structural signature for each child subtree by concatenating its descendant tokens (with lexicographical sorting of multiple branches). Any sibling subtrees sharing identical structural signatures are treated as having the same downstream structure, and the algorithm merges them. Fig. 6 demonstrates this iterative merging. Traversing upward from the leaves to the first branching node `user=root` (Fig. 6(a)), the system compares the child node ⟨∗⟩ (representing numeric `rhosts`) with `rhost=julia.arkos.de` by computing the structural signature for the path under each. Since both yield the same sequence of ⟨∗⟩ nodes (highlighted by the green dashed box), they have the same downstream structure. This matching structure indicates that `rhost=julia.arkos.de` should also be a variable token (i.e., ⟨∗⟩). 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. 6(b). As the traversal continues up to the next branching node, `failure;`, the system performs another merge as shown in Fig. 6(c). In case of incorrect merges, LogNexus separates outlier variables during the correlation mining stage (Sec. 3.2).

Finally, we traverse the global tree to index the structural contexts. Every terminal node is assigned a globally unique `pathID`. As shown in Fig. 7, this ID acts as a *structural context identifier*, representing a group of logs that share the 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`.

3.2 Variable Subtree Encoding

The structural context tree leverages only stable tokens, where all variable tokens are aggregated at the terminal nodes. To capture the deep correlations between the log structure and its parameters,

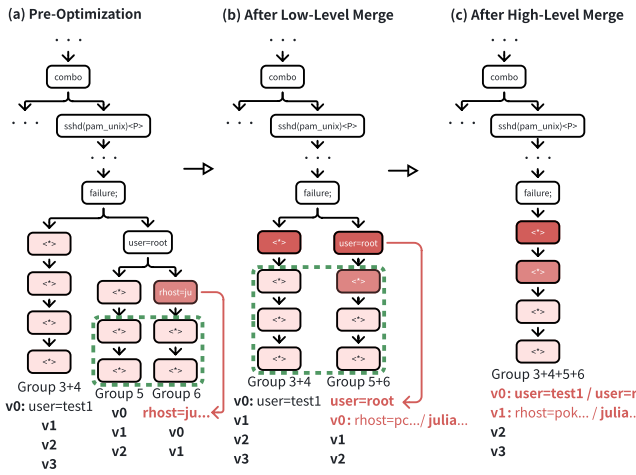


Fig. 6. The Iterative Process of Subtree Merging

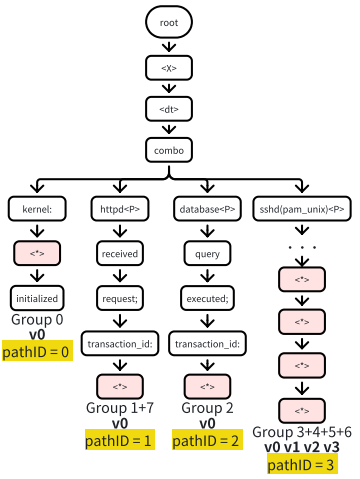


Fig. 7. The Global Structural Tree

we can extend the URT by attaching a prefix subtree to each terminal node, using the `varLists` as input paths. However, if variables are naively inserted into these subtrees in their original order, two critical limitations arise. *First, naive sequential insertion of variables causes excessive branching inside the variable subtree.* As shown in Fig. 8(a), if variables are processed in original order (e.g., $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$), a highly diverse variable may appear early in the sequence (e.g., a `rhost` at v_1), forcing the subtree to branch at the top layers. Consequently, even if later variables form a highly frequent pattern (e.g., `uid=509` → `euid=0`), it is fragmented into multiple separate branches. This means common values such as `euid=0` and `uid=509` cannot be represented once on a shared path, but are repeated under separate `rhost` branches. *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. 3.3).

3.2.1 Variable Reordering for Efficient Subtree Construction. To mitigate this excessive branching, LogNexus restructures the variable subtrees by sorting variable positions according to stability, i.e., placing frequent variables with fewer distinct values earlier. By forcing frequent patterns to share a common prefix, this strategy pushes branches caused by highly diverse variables to the deeper levels of the tree (Fig. 8(b)).

The optimization begins with *high-frequency filtering*, which prunes infrequent variables, as shown in Fig. 9(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 τ, e.g., the rare `rhost=srv2.alfahost.nl`. If all values at a position are filtered out, the entire position is excluded. For instance, in the `httpd` log group (`pathID=1`) shown in Fig. 7, the sole variable v_0 (`transaction_id`) is removed entirely because both of its values (`5001` and `5002`) are too infrequent. In this work, we use τ as our universal threshold for different filtering operations to screen out outliers, which avoids the overhead of managing multiple distinct thresholds and reduces overall system complexity. The second step is *variable reordering*, which establishes the optimal insertion order for the remaining variables. We calculate two metrics for each variable position, i.e., *total frequency* (the sum of occurrences of all its values) and *discriminative power* (the number of

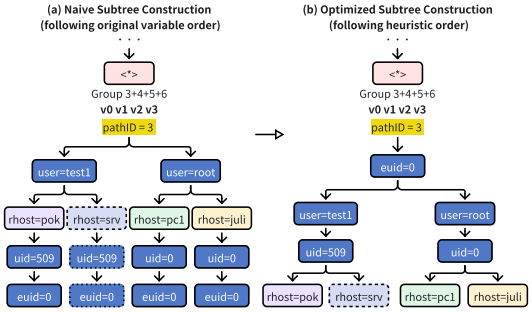


Fig. 8. The Motivation and Effect of Variable Reordering

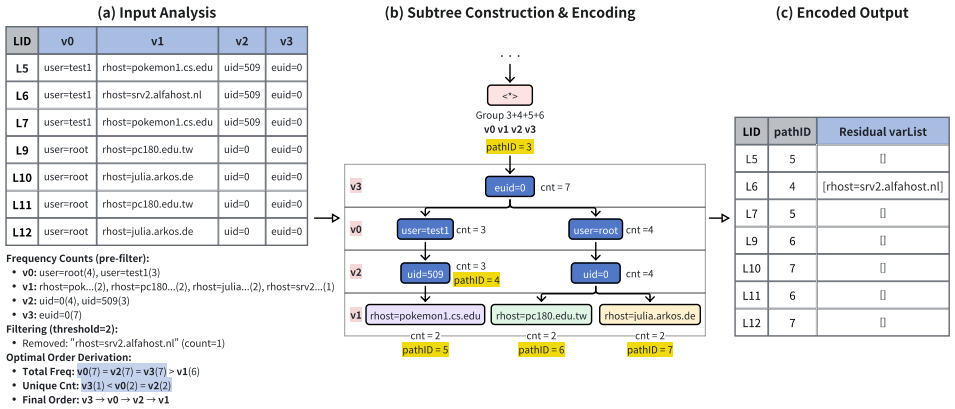


Fig. 9. The Variable Subtree Encoding Pipeline for the sshd Log Group (pathID=3)

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 $v_3 \rightarrow v_0 \rightarrow v_2 \rightarrow v_1$.

3.2.2 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 is guided by frequency and diversity statistics computed for each variable position, 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 they may never co-occur (e.g., that user never logs in from that host). Thus, 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. 9(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 maintains a cnt attribute, recording 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 → user=test1 → 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 the pruning threshold τ is removed. While no nodes are pruned in our simplified example (Fig. 9), this is crucial for eliminating noise in complex datasets.

The most critical step is the assignment of new universal `pathIDs`. 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. 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., τ). For example, the `rhost=pokemon1.cs.edu` node in Fig. 9(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 τ. 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. For instance, the `uid=509` node has a `cnt` of 3, while its only child has a `cnt` of 2. This 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 LogNexus to hierarchically encode both complete patterns and frequent sub-patterns.

The final encoded output, shown in Fig. 9(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, achieving a significant gain in compression efficiency.

3.3 Residual Data Processing

After the first two stages capture frequent collective patterns, the remaining data constitutes "long-tail" outlier variables with 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 the principle of efficient residual handling: after the URT has filtered out complex correlations, this final pipeline can focus on simpler sequential regularities.

3.3.1 Global Sorting for Temporal Coherence. The structural grouping in the first stage, while effective for mining template-based redundancy, inevitably fragments the time order of logs. This can obscure valuable patterns that span different structural groups. As shown in Fig. 10(a), logs from different sources (e.g., `httpd` and `database`) may share a sequential `transaction_id`. To uncover these dependencies, this stage employs 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 restores the temporal coherence, realigning dispersed entries like L3, L4, and L8 and exposing the incremental numeric sequence (`5001`, `5001`, `5002`), which is highly amenable to delta compression.

3.3.2 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

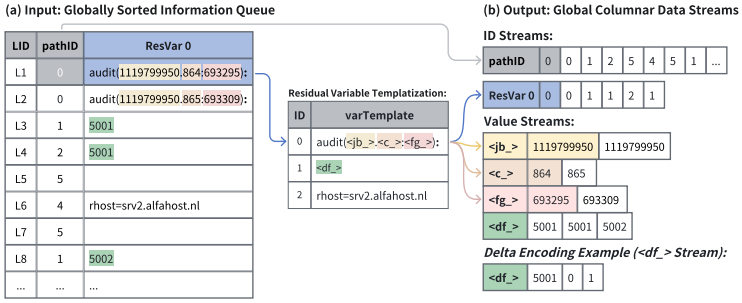


Fig. 10. The Residual Data Processing Pipeline in Stage 3

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 borrow the core principle from Denum [39] to classify each extracted numeric string based on its intrinsic features (e.g., length, first digit) and generate a corresponding placeholder. The format-preserving rule (Sec. 3.1.1) is used here: a placeholder is emitted only when the original token form can be restored from the stored value and its format metadata. Otherwise, LogNexus stores the original token string rather than only the normalized numeric value. This process handles three distinct scenarios:

- **Complex Numeric Variables**: For a composite variable (e.g., `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 numeric values to columnar streams.
- **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 is a set of dense, homogeneous columnar streams, as shown in Fig. 10(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 compressed using Delta Encoding followed by ZigZag and Varint encoding.

Although this stage borrows the principle of numeric classification from Denum, our hierarchical redundancy mining strategy is fundamentally different. Denum performs an indiscriminate, global numeric/non-numeric split from the outset, which can destroy valuable contextual correlations. In contrast, LogNexus has already processed the majority of high-frequency tokens (including many numerics) in the first two stages. Thus, Stage 3 operates only on the residual variables left after structural grouping and frequent structure-variable encoding, rather than on the full log stream.

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 compressor (e.g., LZMA) to exploit remaining byte-level redundancy.

3.4 Decompression

The decompression process of LogNexus is the precise inverse of compression and reconstructs the original token sequence. It begins by loading the metadata from the compressed archive (URT, residual template dictionary, and all columnar data streams). The core 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, reconstruction has two stages. First, the `pathID` is used to perform a reverse lookup in the URT. By tracing backward from the `pathID` node to the root, we 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, the decompressor reads the next value from the corresponding columnar stream and applies its restoration rule, using the recorded format metadata when needed, before substitution. Since the streams are read in their written order, this restores the original token sequence.

4 Evaluation

We conduct a comprehensive evaluation to demonstrate the effectiveness and efficiency of LogNexus. Our evaluation is designed to answer the following three important research questions:

- **RQ1:** How does the overall performance of LogNexus compare to state-of-the-art methods?
- **RQ2:** What is the impact of input data granularity on LogNexus's performance?
- **RQ3:** What are the individual performance contributions of LogNexus's hierarchical stages?

4.1 Experiment Settings

4.1.1 Datasets. Consistent with our previous empirical study (Sec. 2.2), we utilize the LogHub benchmark [45] for evaluation. Differently, this evaluation includes the Android and Windows datasets (16 datasets in total), as all compressors successfully handle them with their default parsers.

4.1.2 Evaluation Metrics. Besides the **Compression Ratio (CR)** defined in Sec. 2, we use **Compression Speed (CS)** and **Decompression Speed (DS)** for efficiency. We define CS = $\frac{Original\ File\ Size}{Compression\ Time}$ and DS = $\frac{Restored\ File\ Size}{Decompression\ Time}$.

4.1.3 Baselines. We evaluate LogNexus against the same four state-of-the-art log-specific compressors studied in our empirical study (Sec. 2.2). These include LogZip [24], the pioneering parser-based method; LogReducer [35] and LogShrink [22], which represent advanced approaches optimized for parameter correlation and variability; and Denum [39], a number-centric log compressor. The parser-swapping study in Sec. 2.2 is diagnostic rather than a tuning step for RQ1. Each baseline follows its standard setting from the original paper or released implementation. Choosing the best parser per compressor and dataset after observing Fig. 1 would create a post-hoc oracle setting.

4.1.4 Implementation. All experiments were run on an Ubuntu 22.04 LTS server with an AMD EPYC 9224 CPU (24 cores/48 threads, 3.70 GHz) and 251 GB RAM. LogNexus is implemented in C++ and uses PCRE2 with 8-bit code units. Following prior work, RQ1 and RQ3 split input logs into 100K-line chunks and use four chunk-processing threads for each compressor. The default **LogNexus** keeps each chunk-processing thread internally single-threaded. To demonstrate the full potential of our parallel-aware design (Sec. 3.1.1), we also evaluate **LogNexus-P**, where each chunk-processing thread uses 4 internal worker threads. The total time to compress all chunks is recorded. LogNexus packs intermediate files with `tar` and compresses them with `lzma`.

4.2 RQ1: Overall Performance of LogNexus

Compression Ratio Analysis: As shown in Table 1, LogNexus achieves the highest CR on 14 of 16 datasets. Compared with Denum, LogNexus improves the average CR by 9.47%, with gains reaching 25.54% on Linux and 24.23% on Thunderbird. This result is consistent with LogNexus's design: it keeps stable structural tokens and correlated variable values in the same URT path, while split-based methods encode them in separate streams. On the two datasets where LogNexus does not achieve the highest CR, HDFS and Spark, the results remain competitive. On HDFS, the difference from the best result is 5.16%. The larger gap on Spark (8.42%) 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 <*>`") [39]. In this case, Stage 2 has less advantage over methods optimized for simple numeric streams.

Compression Speed Analysis: We measure LogNexus and Denum end-to-end from raw log input to final archive. For parser-based baselines, we follow the timing convention in their original papers, which excludes their parsing and template generation steps. Thus, LogNexus is evaluated under a stricter timing scope; including parsing and template generation would further reduce the reported compression speed of parser-based baselines.

As shown in Table 1, under the default LogNexus configuration, LogNexus is faster than all baselines on all 16 datasets, averaging 26.92 MB/s and outperforming Denum by 1.51×. This result is consistent with the Stage-2 workload reduction analyzed in RQ3. LogNexus-P further reaches 36.24 MB/s on average, a 34.6% speedup over LogNexus, and is the fastest configuration on all 16 datasets, showing the benefit of internal parallelism beyond chunk-level processing.

Decompression Speed and Restoration Analysis: Because decompression speed is meaningful only with correct restoration, Table 2 reports decompression speed together with restoration verification. PASS means that all restored tokens match the original log in order, FAIL means that a restored log is produced but not all tokens match, and N/R means that the released artifact could not reconstruct all chunks, so no complete restored log was available for comparison with the original log. We exclude LogZip from the decompression table because its public implementation does not provide an official decompression tool, which avoids drawing conclusions from a non-official restoration procedure. As an auxiliary check, our best-effort restoration from its generated archive matched the original tokens on 5/16 datasets under the same 100K-line chunk setting.

In Table 2, LogNexus restores every token on all 16 datasets and achieves an average DS of 16.40 MB/s. This is 1.89× faster than LogReducer and 3.94× faster than LogShrink. LogReducer and LogShrink restore every token on only 6/16 and 1/16 datasets, respectively. Denum has higher raw output throughput on average, but its restored outputs do not match the original tokens in our runs. For rows marked FAIL, the differences show concrete patterns: some outputs lose the lexical form of numeric fields (LogReducer, LogShrink, and Denum), such as restoring `09` as `9`; others change numerical values (LogReducer and LogShrink), such as restoring `2831866760` as `-1463100536`; and others break token-stream restoration (LogReducer, LogShrink, and Denum), including token substitution or misalignment in LogReducer and LogShrink, token-count differences in Denum on 9/16 datasets, and unresolved placeholders such as `<a>` and `<b>` in Denum's restored Spark logs.

Summary for RQ1: LogNexus establishes a new state-of-the-art in both effectiveness and efficiency. It surpasses existing log-specific compressors in compression ratio on 14 of 16 datasets. Furthermore, it achieves the fastest end-to-end speed, driven by the synergistic effect of its hierarchical encoding strategy and parallel-aware architecture. For decompression, LogNexus is the only evaluated method that restores every token on all 16 datasets.

Table 1. Experimental Results of Compression Ratio and Speed

Dataset	Compression Ratio					Compression Speed (MB/s)					
	LogZip	LogRed.	LogShr.	Denum	LogNexus	LogZip	LogRed.	LogShr.	Denum	LogNexus	LogNexus-P
Andr	25.165	20.776	21.857	32.494	**32.567**	0.068	19.323	4.123	24.380	**27.022**	34.116
Apac	30.375	43.028	55.940	58.517	**65.342**	0.737	2.347	1.537	6.369	**12.302**	15.256
BGL	32.655	38.600	42.385	41.804	**47.586**	0.874	26.738	2.571	23.575	**30.922**	44.962
Hado	35.008	52.830	60.091	78.546	**80.626**	0.901	12.882	4.401	24.453	**41.085**	48.588
HDFS	16.666	22.634	**27.319**	25.670	25.909	0.701	23.598	3.466	25.193	**25.306**	28.356
Heal	22.632	31.694	39.072	44.472	**49.430**	0.736	7.937	2.754	17.866	**19.306**	24.728
HPC	27.208	32.070	35.878	45.275	**46.004**	0.644	9.391	3.599	25.216	**25.392**	29.249
Linu	23.368	25.213	29.252	30.449	**38.226**	0.687	1.249	0.941	4.969	**8.986**	10.669
Mac	26.306	35.251	39.860	40.789	**43.703**	0.009	5.450	2.141	6.710	**10.901**	15.407
OSSH	42.606	86.699	103.175	101.654	**110.929**	0.715	14.773	3.335	25.572	**49.132**	70.971
OStk	17.258	16.701	22.157	22.238	**23.265**	0.537	13.039	4.018	12.352	**20.098**	22.926
Prox	21.493	25.501	27.029	27.288	**30.081**	0.716	1.328	0.742	6.000	**8.732**	10.395
Spar	20.825	59.470	**59.739**	59.470	54.709	0.550	21.233	3.185	26.034	**38.304**	53.668
Thun	–	49.185	48.434	63.824	**79.288**	–	18.656	4.036	19.830	**36.435**	63.654
Wind	310.596	342.975	456.301	481.350	**539.070**	1.357	18.483	6.330	28.783	**59.431**	82.094
Zook	47.373	94.562	116.981	135.251	**144.491**	0.842	4.429	2.280	8.000	**17.376**	24.745
Avg	46.636	61.074	74.092	80.568	**88.202**	0.672	12.554	3.091	17.831	**26.921**	36.237

Andr (Android), Apac (Apache), Hado (Hadoop), Heal (HealthApp), Linu (Linux), OSSH (OpenSSH)
OStk (OpenStack), Prox (Proxifier), Spar (Spark), Thun (Thunderbird), Wind (Windows), Zook (Zookeeper)

Table 2. Decompression Speed (DS, MB/s) with Restoration Verification Results

Dataset	LogRed.		LogShr.		Denum		LogNexus	
	DS	Ver.	DS	Ver.	DS	Ver.	DS	Ver.
Android	5.356	FAIL	3.250	FAIL	27.610	FAIL	30.144	**PASS**
Apache	5.686	FAIL	3.278	FAIL	24.854	FAIL	8.284	**PASS**
BGL	10.517	**PASS**	N/R	N/R	30.130	FAIL	35.542	**PASS**
Hadoop	9.918	FAIL	3.901	FAIL	31.903	FAIL	12.303	**PASS**
HDFS	13.603	**PASS**	N/R	N/R	34.583	FAIL	23.441	**PASS**
HealthApp	8.808	FAIL	N/R	N/R	25.772	FAIL	8.620	**PASS**
HPC	9.197	**PASS**	4.253	**PASS**	30.806	FAIL	7.225	**PASS**
Linux	2.380	FAIL	1.683	FAIL	14.984	FAIL	4.663	**PASS**
Mac	5.328	FAIL	2.346	FAIL	25.357	FAIL	6.262	**PASS**
OpenSSH	10.741	**PASS**	N/R	N/R	33.900	FAIL	43.503	**PASS**
OpenStack	14.649	**PASS**	6.631	FAIL	32.808	FAIL	12.488	**PASS**
Proxifier	1.582	FAIL	N/R	N/R	13.992	FAIL	3.231	**PASS**
Spark	8.652	**PASS**	3.368	FAIL	32.601	FAIL	10.372	**PASS**
Thunderbird	10.581	FAIL	4.901	FAIL	27.547	FAIL	20.242	**PASS**
Windows	15.299	FAIL	8.004	FAIL	39.738	FAIL	26.205	**PASS**
Zookeeper	6.396	FAIL	N/R	N/R	28.538	FAIL	9.873	**PASS**
Avg.	8.668	6/16	4.162	1/16	28.445	0/16	16.400	**16/16**

Table 3. Robustness Analysis of Single-Archive Mode Performance

Dataset	CR		CS	
	Denum	LogNexus	Denum	LogNexus
Android	46.191	**50.646**	5.416	**11.457**
Apache	58.562	**65.084**	5.810	**18.191**
BGL	44.284	**48.216**	6.100	**13.890**
Hadoop	92.899	**94.019**	8.643	**23.847**
HDFS	**32.919**	32.286	5.230	**7.556**
HealthApp	47.279	**52.357**	7.072	**9.782**
HPC	45.542	**46.509**	6.369	**12.950**
Linux	30.688	**38.181**	4.582	**9.281**
Mac	43.633	**49.152**	5.245	**12.121**
OpenSSH	106.984	**117.418**	7.426	**27.901**
OpenStack	21.898	**23.542**	5.129	**14.157**
Proxifier	27.141	**30.074**	5.484	**9.961**
Spark	**65.125**	58.246	5.398	**12.971**
Thunderbird	70.161	**90.829**	4.263	**15.499**
Windows	3407.766	**4494.089**	6.421	**19.369**
Zookeeper	135.846	**144.402**	7.556	**14.321**
Average	267.307	**339.691**	6.009	**14.578**

4.3 RQ2: Performance Impact of Data Granularity

This question investigates the impact of data granularity on performance and resource use. The baseline setting uses a chunk size of 100K lines, which maximizes parallelism and keeps memory low but limits pattern discovery to local windows. This 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-compression-memory trade-off, we configure LogNexus to operate in single-archive (non-chunked) mode, allowing it to

Table 4. CPU and Memory Footprint Analysis (Memory is peak RSS in MB)

Dataset	Chunked mode								Single-archive mode			
	LogReducer		LogShrink		Denum		LogNexus		Denum		LogNexus	
	Cores	Mem.	Cores	Mem.	Cores	Mem.	Cores	Mem.	Cores	Mem.	Cores	Mem.
Android	2.44	1651.51	2.20	1654.65	3.58	**354.94**	3.45	377.95	0.99	1538.17	0.89	**1206.98**
Apache	1.96	1649.93	3.01	1653.08	0.97	42.47	0.96	**31.46**	0.98	42.47	1.34	**36.07**
BGL	2.72	1671.95	1.55	1671.95	3.57	607.06	3.64	**582.96**	0.99	5644.42	1.12	**4822.63**
Hadoop	2.87	1659.37	2.83	1662.52	2.37	306.50	2.20	**230.28**	0.98	360.76	1.23	**348.63**
HDFS	3.17	1689.26	1.60	1662.52	3.82	**431.69**	3.43	593.17	0.96	**12383.28**	1.29	13564.45
HealthApp	2.45	1653.08	3.29	1660.94	2.20	198.90	2.31	**174.75**	0.98	**230.04**	1.26	264.63
HPC	2.57	1646.79	2.24	1654.65	3.37	273.92	2.46	**269.24**	0.99	326.63	1.19	**321.42**
Linux	1.72	1646.79	2.68	1651.51	0.94	26.74	0.86	**20.45**	0.84	26.74	1.09	**20.45**
Mac	1.84	1649.93	2.39	1656.23	1.15	125.83	1.21	**87.33**	0.94	132.12	1.05	**104.74**
OpenSSH	2.75	1648.36	1.92	1651.51	3.09	284.74	3.05	**215.65**	0.94	555.11	1.43	**451.24**
OpenStack	2.55	1673.53	1.92	1670.38	1.98	382.35	1.58	**295.93**	0.96	429.64	1.37	**384.34**
Proxifier	2.57	1645.22	2.91	1649.93	0.93	23.59	0.92	**20.45**	0.95	23.59	0.93	**20.45**
Spark	1.70	1678.25	1.71	2180.65	3.92	341.19	3.84	**308.08**	0.99	25754.27	1.44	**18919.18**
Thunderbird	2.19	1722.29	1.47	2468.73	3.92	862.56	3.61	**621.67**	0.96	318362.50	1.64	**165021.45**
Windows	1.53	1708.13	1.54	3296.08	3.94	**512.64**	3.48	631.89	0.96	174731.35	1.37	**135194.91**
Zookeeper	1.91	1648.36	2.30	1657.80	0.97	75.50	0.92	**45.11**	0.92	75.50	0.97	**62.24**
Average	2.31	1665.17	2.22	1843.95	2.54	303.16	2.37	**281.65**	0.96	33788.54	1.23	**21296.49**

construct a single global URT over the entire dataset. We compare this configuration against Denum, the state-of-the-art baseline that also supports global operation. Table 3 reports the single-archive results. Following Denum's chunk-size analysis [39], we also check BGL, HDFS, and Spark with chunks of 100K, 300K, 1M, and 3M lines under the same chunked LogNexus setting. As shown in Table 6, the average CR increases from 42.735 at 100K to 47.404 at 3M, while the average CS decreases from 31.511 MB/s to 21.383 MB/s. This confirms the expected trade-off and supports using 100K as a fair and efficient default rather than as the setting with the highest CR.

Compression Ratio Analysis: In single-archive mode, LogNexus achieves a 285.13% increase in average CR, with the gain on the Windows dataset reaching 733.67%. When compared to Denum in the same global configuration, LogNexus outperforms it on 14 of the 16 datasets. The improvements are particularly significant on complex logs such as Windows (+31.88%) and Thunderbird (+29.46%). On HDFS, LogNexus remains close to Denum with a difference of 1.92%. The main exception is Spark, where Denum's specialized handling of specific numeric sequences is more efficient.

Compression Speed Analysis: LogNexus maintains an absolute speed advantage in global mode, with an average speed of 14.58 MB/s, 2.43× faster than Denum. This performance stems from LogNexus's hybrid parallel architecture. While standard chunk-based compressors lose parallelism when processing a single global block, LogNexus 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. Even in global mode, LogNexus remains faster than Denum in the same setting and faster than the parser-based baselines reported in RQ1 under their default timing convention.

Resource Footprint Analysis: To address deployment cost, we measure CPU utilization and peak resident memory. Table 4 reports all 16 datasets in both settings. In chunked mode, LogNexus has the lowest average peak memory among the four methods, reducing peak memory by 7.10% compared with Denum, 83.08% compared with LogReducer, and 84.73% compared with LogShrink. Under the same four-thread chunked setting, its average CPU usage is close to the other methods. In single-archive mode, LogNexus reduces average peak memory by 36.97% compared with Denum. This setting still raises LogNexus's average peak memory from 281.65 MB to 21296.49 MB, mainly because Thunderbird and Windows require a single global model over much larger inputs.

Table 5. Ablation Study of Compression Performance (Ratio and Speed)

Dataset	Compression Ratio				Speed (MB/s)			
	LZMA	S1+LZMA	S1+S3+LZMA	Full	LZMA	S1+LZMA	S1+S3+LZMA	Full
Android	18.857	21.901	25.341	**32.567**	24.876	21.189	20.973	**34.116**
Apache	25.186	46.768	62.211	**65.342**	7.665	11.001	10.172	**15.256**
BGL	17.637	35.643	46.992	**47.586**	16.108	24.118	40.218	**44.962**
Hadoop	36.095	67.305	76.318	**80.626**	22.102	29.557	32.479	**48.588**
HDFS	13.559	19.062	23.900	**25.909**	15.715	17.887	21.895	**28.356**
HealthApp	13.431	23.112	46.150	**49.430**	9.436	11.477	14.975	**24.728**
HPC	15.076	28.210	44.283	**46.004**	11.700	15.205	19.803	**29.249**
Linux	16.677	33.401	36.847	**38.226**	4.592	9.138	7.257	**10.669**
Mac	22.159	33.689	37.308	**43.703**	8.428	9.803	10.099	**15.407**
OpenSSH	18.918	65.048	90.120	**110.929**	15.873	37.400	37.800	**70.971**
OpenStack	14.437	19.634	21.799	**23.265**	11.647	15.559	15.627	**22.926**
Proxifier	18.982	24.664	26.835	**30.081**	6.696	7.887	6.102	**10.395**
Spark	19.908	35.033	51.386	**54.709**	20.757	39.702	41.885	**53.668**
Thunderbird	27.309	49.350	58.915	**79.288**	26.845	34.366	34.771	**63.654**
Windows	202.568	349.213	530.719	**539.070**	63.706	63.410	62.131	**82.094**
Zookeeper	27.667	69.360	121.987	**144.491**	6.581	11.574	10.528	**24.745**
Average	31.779	57.587	81.319	**88.202**	17.045	22.454	24.170	**36.237**

Summary for RQ2: LogNexus's design offers both robustness and flexibility, giving users a clear speed-compression-memory trade-off: the default chunked mode keeps memory low, while the single-archive mode maximizes the compression ratio with more memory.

4.4 RQ3: Ablation Analysis of LogNexus's Different Stages

We perform a progressive ablation study to quantify the contribution of each stage in LogNexus's hierarchical design. The evaluation follows the path LZMA → S1+LZMA → S1+S3+LZMA → Full. LZMA is the general compression baseline. S1+LZMA keeps Stage 1 only, which builds the URT structural skeleton and groups structurally similar logs before final LZMA compression. S1+S3+LZMA further enables Stage 3, which losslessly handles the remaining rare and unresolved values after Stage 1 while still bypassing Stage 2. Full LogNexus then adds Stage 2 (Variable Subtree Encoding), the core implementation of unified redundancy encoding that mines recurring "structure+variable" correlations. All three LogNexus variants use LZMA as the final compression backend. Table 5 reports results under the same chunked and parallel setting as RQ1.

Compression Ratio Analysis: Table 5 shows that each stage contributes to compression ratio. Averaged across datasets, Stage 1 improves CR over LZMA by 83.25% by grouping structurally similar logs before backend compression. Adding Stage 3 further improves CR over S1+LZMA by 34.33% by transforming residual values into more regular streams. Full LogNexus adds Stage 2 and improves CR over S1+S3+LZMA by 11.48%, indicating that unified redundancy encoding captures recurring structure-variable co-occurrences beyond structural grouping and residual handling.

Compression Speed Analysis: Table 5 also shows a progressive speed improvement. Stage 1 improves speed over LZMA by 31.74% because structural grouping produces more regular streams for backend compression. Adding Stage 3 further improves speed over S1+LZMA by 7.64%, suggesting that residual-value regularization reduces the backend compression burden enough to offset its extra processing cost. Full LogNexus adds Stage 2 and improves speed over S1+S3+LZMA by 49.93%, showing that Stage 2 is the main speed accelerator in the hierarchy.

Case Study Discussion: To connect the aggregate ablation results with concrete workload changes, we inspect OpenSSH and HDFS. Stage 1 extracts 1,596,090 variables on OpenSSH, and Stage 2 encodes 1,342,009 of them into variable-subtree `pathIDs`, covering 84.08% of variables and 88.60% of variable bytes. Only 254,081 variables remain for Stage 3, which explains why Full LogNexus

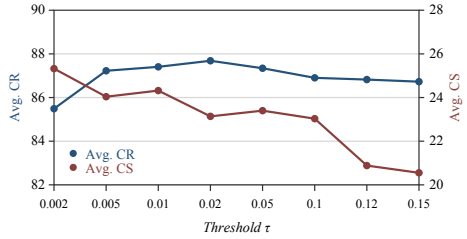


Fig. 11. Sensitivity Analysis of LogNexus on τ

Table 6. Chunk-Size Sensitivity of LogNexus

Chunk	CR			CS (MB/s)		
	BGL	HDFS	Spark	BGL	HDFS	Spark
100K	47.586	25.909	54.709	30.922	25.306	38.304
300K	49.755	27.683	56.719	28.740	24.452	37.868
1M	51.491	29.505	57.936	21.077	22.597	35.902
3M	52.222	31.605	58.385	13.344	21.256	29.550

improves over S1+S3+LZMA from 90.120 to 110.929 in CR and from 37.800 MB/s to 70.971 MB/s in speed. HDFS is a conservative case: Stage 2 encodes 49.94% of Stage-1 variables, but these variables cover only 26.39% of variable bytes. Full LogNexus still improves HDFS from 23.900 to 25.909 in CR and from 21.895 MB/s to 28.356 MB/s in speed, but the gain is smaller because much of the residual byte mass remains in string patterns handled by Stage 3 and the backend compressor.

Summary for RQ3: The ablation validates LogNexus's hierarchical design. Stage 1 contributes structural grouping, Stage 3 contributes residual-value regularization, and Stage 2 further improves the already strong S1+S3+LZMA baseline by mining unified structure-variable redundancy.

4.5 Threshold Sensitivity

To examine the sensitivity of LogNexus to the frequency threshold τ, we vary it over eight values spanning a broad range from 0.002 to 0.15, and rerun the chunk-level compression experiment on all 16 datasets. As shown in Fig. 11, the average CR rises when τ increases from 0.002 to 0.005, then remains within a narrow range from 86.73 to 87.68 across $\tau \in [0.005, 0.15]$. The highest fixed-threshold average CR is 87.68 at $\tau = 0.02$. Compression speed generally decreases as τ increases, which is consistent with the fact that a higher threshold keeps fewer values in the URT and leaves more residual values for Stage 3 and the backend compressor. We therefore use $\tau = 0.02$ as a representative fixed threshold because it gives the highest average CR among fixed thresholds while maintaining throughput comparable to nearby settings.

5 Related Work

General-purpose Compression Approaches. These algorithms eliminate statistical redundancy in data for compression, which 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.

Log-specific Compression Approaches. The dominant methodology in log compression is the parser-based paradigm [22, 24, 31, 34, 35], which decouples structure extraction from variable encoding. LogZip established this framework by combining the Drain parser with iterative clustering. Subsequent works focused on optimizing the separated components: LogReducer introduced delta encoding for timestamps, while LogShrink utilized entropy-based analysis to identify variable patterns. Additionally, systems like CLP and LogGrep have extended this paradigm to enable direct

search on compressed data. To avoid the overhead of template extraction, alternative approaches have emerged. Early works, such as LogArchive [8] and MLC [13], applied similarity grouping or block-level deduplication to compress logs. More recently, the number-centric paradigm, exemplified by Denum [39], performs a global binary classification to split tokens into numeric and non-numeric streams. Concurrent work DeLog [40] empirically studies the relation between log parsing accuracy and compression ratio, and also observes that conventional parsing objectives are not fully aligned with compression effectiveness. DeLog addresses this issue by synthesizing pattern signatures and grouping same-signature tokens into compressible streams, while LogNexus uses a single URT where structural contexts and frequent correlated variable values share paths. On the same 16 public datasets, LogNexus obtains a marginally higher average CR than the DeLog results reported in its paper, with higher CR on 8 of 16 datasets.

However, prior parser-based and number-centric methods share a fundamental limitation: they rely on early, rigid token categorization. Parser-based methods separate templates from variables, while number-centric methods separate numerics from strings. This irreversible decoupling destroys contextual correlations that span these boundaries. Instead, LogNexus unifies structural extraction and pattern encoding to mine deep "structure+variable" redundancies that existing methods ignore.

6 Threats to Validity

External Validity. The external threat is whether the results generalize beyond the evaluated datasets. LogNexus is evaluated on 16 public benchmark datasets that cover multiple types of log-producing systems. However, these datasets might not fully capture the diversity of logs in different operational environments, especially proprietary large-scale industrial systems. Therefore, the effectiveness of LogNexus may differ on logs from domains not represented in our evaluation.

Internal Validity. LogNexus uses predefined regular expressions and a numeric-token heuristic in preprocessing and residual-value handling, following the common practice of number-aware log compression. These rules identify common fields such as dates, timestamps, PIDs, and numeric-bearing tokens, and they affect how tokens are routed to different stages of the pipeline. When a log contains uncommon numeric formats, these rules may extract fewer compact numeric patterns and leave more values to residual storage, which can influence compression ratio and speed.

7 Conclusion

This paper reevaluates the prevailing "parse-then-compress" paradigm in log storage, identifying the rigid decoupling of structure extraction and variable encoding as a fundamental bottleneck. Our empirical analysis reveals that high parsing accuracy does not guarantee compression efficiency, as it often obscures critical "template-variable" and inter-variable correlations. To address this, we introduce LogNexus, a framework implementing Unified Redundancy Encoding through a Unified Redundancy Tree (URT) that dynamically models both log structure and variable patterns. Leveraging hierarchical redundancy mining and fine-grained parallelism, LogNexus simultaneously optimizes compression ratio and throughput. Evaluations on 16 benchmarks show that LogNexus establishes a new state-of-the-art. It achieves the highest compression ratio on 14 datasets (outperforming baselines by 9.48%~89.13%) and the fastest speed (1.51×~40.06× faster than competitors). Moreover, in single-archive mode, LogNexus improves compression by 285.13%, 27.08% higher than Denum with a 2.43× speed advantage. These findings demonstrate that co-designing parsing and compression is critical for maximizing log data reduction in large-scale systems.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62402536).

Data-Availability Statement

The artifact for LogNexus is available on Zenodo [25]. The source code of LogNexus is also available at https://github.com/OpsPAI/LogNexus.

References

[1] Zhuangbin Chen, Zhihan Jiang, Yuxin Su, Michael R. Lyu, and Zibin Zheng. 2024. Tracemesh: Scalable and Streaming Sampling for Distributed Traces. In *17th IEEE International Conference on Cloud Computing, CLOUD 2024, Shenzhen, China, July 7-13, 2024*, Rong N. Chang, Carl K. Chang, Jingwei Yang, Nimanthi L. Atukorala, Zhi Jin, Michael Sheng, Jing Fan, Kenneth Fletcher, Qiang He, Tevfik Kosar, Santonu Sarkar, Sreekrishnan Venkateswaran, Shangguang Wang, Xuanzhe Liu, Seetharami Seelam, Chandra Narayanaswami, and Ziliang Zong (Eds.). IEEE, 54–65. doi:10.1109/CLOUD62652.2024.00016

[2] Zhuangbin Chen, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, Yingnong Dang, Feng Gao, Pu Zhao, Bo Qiao, Qingwei Lin, Dongmei Zhang, and Michael R. Lyu. 2020. 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*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1487–1497. doi:10.1145/3368089.3417055

[3] Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R. Lyu. 2021. Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection. *CoRR* abs/2107.05908 (2021). arXiv:2107.05908 https://arxiv.org/abs/2107.05908

[4] Zhuangbin Chen, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xiao Ling, and Michael R. Lyu. 2022. Adaptive Performance Anomaly Detection for Online Service Systems via Pattern Sketching. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 61–72. doi:10.1145/3510003.3510085

[5] Zhuangbin Chen, Junsong Pu, and Zibin Zheng. 2025. Tracezip: Efficient Distributed Tracing via Trace Compression. *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 411–433. doi:10.1145/3728888

[6] Zhuangbin Chen, Juzheng Zheng, and Zibin Zheng. 2025. Metronome: Differentiated Delay Scheduling for Serverless Functions. *CoRR* abs/2512.05703 (2025). arXiv:2512.05703 doi:10.48550/ARXIV.2512.05703

[7] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. 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*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 217–231. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow

[8] Robert Christensen and Feifei Li. 2013. 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*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1283–1284. doi:10.1145/2463676.2465341

[9] Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. *RFC* 1951 (1996), 1–17. doi:10.17487/RFC1951

[10] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. 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*, Shan Lu and Erik Riedel (Eds.). USENIX Association, 139–150. https://www.usenix.org/conference/atc15/technical-session/presentation/ding

[11] Min Du and Feifei Li. 2016. Spell: Streaming Parsing of System Event Logs. In *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, Francesco Bonchi, Josep Domingo-Ferrer, Ricardo Baeza-Yates, Zhi-Hua Zhou, and Xindong Wu (Eds.). IEEE Computer Society, 859–864. doi:10.1109/ICDM.2016.0103

[12] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1285–1298. doi:10.1145/3133956.3134015

[13] Bo Feng, Chentao Wu, and Jie Li. 2016. MLC: An Efficient Multi-level Log Compression Method for Cloud Backup Systems. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*. IEEE, 1358–1365. doi:10.1109/TRUSTCOM.2016.0215

[14] Ying Fu, Meng Yan, Jian Xu, Jianguo Li, Zhongxin Liu, Xiaohong Zhang, and Dan Yang. 2022. 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*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1566–1577. doi:10.1145/3540250.3558947

[15] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. 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*, Ilkay Altintas and Shiping Chen (Eds.). IEEE, 33–40. doi:10.1109/ICWS.2017.13

[16] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. 2022. A Survey on Automated Log Analysis for Reliability Engineering. *ACM Comput. Surv.* 54, 6 (2022), 130:1–130:37. doi:10.1145/3460345

[17] Zhihan Jiang, Junjie Huang, Guangba Yu, Zhuangbin Chen, Yichen Li, Renyi Zhong, Cong Feng, Yongqiang Yang, Zengyin Yang, and Michael R. Lyu. 2025. L4: Diagnosing Large-scale LLM Training Failures via Automated Log Analysis. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*, Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang (Eds.). ACM, 51–63. doi:10.1145/3696630.3728531

[18] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R. Lyu. 2024. 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*, Maria Christakis and Michael Pradel (Eds.). ACM, 223–234. doi:10.1145/3650212.3652123

[19] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. 2008. 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*, Hong Zhu (Ed.). IEEE Computer Society, 181–186. doi:10.1109/QSIC.2008.50

[20] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel C. Briand. 2022. 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*. ACM, 1095–1106. doi:10.1145/3510003.3510101

[21] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel C. Briand. 2024. Impact of log parsing on deep learning-based anomaly detection. *Empir. Softw. Eng.* 29, 6 (2024), 139. doi:10.1007/S10664-024-10533-W

[22] Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. 2024. LogShrink: Effective Log Compression by Leveraging Commonality and Variability of Log Data. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 23:1–23:12. doi:10.1145/3597503.3608129

[23] Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. 2023. 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*. IEEE, 830–842. doi:10.1109/ICSE48619.2023.00078

[24] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R. Lyu. 2019. Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 863–873. doi:10.1109/ASE.2019.00085

[25] Yang Liu, Kaiming Zhang, Zhuangbin Chen, and Zibin Zheng. 2026. Artifact for LogNexus: Effective Log Compression via Unified Redundancy Encoding. doi:10.5281/zenodo.21281836

[26] Adetokunbo Makanju, Nur Zincir-Heywood, and Evangelos E. Milios. 2009. 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*, John F. Elder IV, Françoise Fogelman-Soulié, Peter A. Flach, and Mohammed Javeed Zaki (Eds.). ACM, 1255–1264. doi:10.1145/1557019.1557154

[27] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel C. Briand, and Raimondas Sasnauskas. 2018. 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*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 167–177. doi:10.1145/3196321.3196340

[28] Masayoshi Mizutani. 2013. Incremental Mining of System Log Format. In *2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*. IEEE Computer Society, 595–602. doi:10.1109/SCC.2013.73

[29] Meiyappan Nagappan and Mladen A. Vouk. 2010. 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*, Jim Whitehead and Thomas Zimmermann (Eds.). IEEE Computer Society, 114–117. doi:10.1109/MSR.2010.5463281

[30] Junsong Pu, Yichen Li, Zhuangbin Chen, Jinyang Liu, Zhihan Jiang, Jianjun Chen, Rui Shi, Zibin Zheng, and Tieying Zhang. 2025. ErrorPrism: Reconstructing Error Propagation Paths in Cloud Service Systems. *CoRR* abs/2509.26463 (2025). arXiv:2509.26463 doi:10.48550/ARXIV.2509.26463

[31] Kirk Rodrigues, Yu Luo, and Ding Yuan. 2021. CLP: Efficient and Scalable Search on Compressed Text Logs. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 183–198. https://www.usenix.org/conference/osdi21/presentation/rodrigues

[32] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. 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*, Craig Macdonald, Iadh Ounis, and Ian Ruthven (Eds.). ACM, 785–794. doi:10.1145/2063576.2063690

[33] Rui Wang, Devin Gibson, Kirk Rodrigues, Yu Luo, Yun Zhang, Kaibo Wang, Yupeng Fu, Ting Chen, and Ding Yuan. 2024. μSlope: High Compression and Fast Search on Semi-Structured Logs. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 529–544. https://www.usenix.org/conference/osdi24/presentation/wang-rui

[34] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. 2023. LogGrep: Fast and Cheap Cloud Log Storage by Exploiting both Static and Runtime Patterns. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 452–468. doi:10.1145/3552326.3567484

[35] Junyu Wei, Guangyan Zhang, Yang Wang, Zhiwei Liu, Zhanyang Zhu, Junchao Chen, Tingtao Sun, and Qi Zhou. 2021. On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, Marcos K. Aguilera and Gala Yadgar (Eds.). USENIX Association, 249–262. https://www.usenix.org/conference/fast21/presentation/wei

[36] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. 2010. 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*, Johannes Fürnkranz and Thorsten Joachims (Eds.). Omnipress, 37–46. https://icml.cc/Conferences/2010/papers/902.pdf

[37] Stephen Yang, Seo Jin Park, and John K. Ousterhout. 2018. 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*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 335–350. https://www.usenix.org/conference/atc18/presentation/yang-stephen

[38] Kundi Yao, Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2020. A study of the performance of general compressors on log files. *Empir. Softw. Eng.* 25, 5 (2020), 3043–3085. doi:10.1007/S10664-020-09822-X

[39] Siyu Yu, Yifan Wu, Ying Li, and Pinjia He. 2024. 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*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 919–930. doi:10.1145/3691620.3695474

[40] Siyu Yu, Yifan Wu, Junjielong Xu, Ying Fu, Ning Wang, Maoyin Liu, Pancheng Jiang, Xiang Zhang, Tong Jia, Pinjia He, and Ying Li. 2026. DeLog: An Efficient Log Compression Framework with Pattern Signature Synthesis. *CoRR* abs/2601.15084 (2026). arXiv:2601.15084 doi:10.48550/ARXIV.2601.15084

[41] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. 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*, James C. Hoe and Vikram S. Adve (Eds.). ACM, 143–154. doi:10.1145/1736020.1736038

[42] Wei Yuan, Shan Lu, Hailong Sun, and Xudong Liu. 2020. How are distributed bugs diagnosed and fixed through system logs? *Inf. Softw. Technol.* 119 (2020). doi:10.1016/J.INFSOF.2019.106234

[43] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. 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*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 603–618. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhao

[44] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 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*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 683–694. doi:10.1145/3338906.3338961

[45] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2019. 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*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 121–130. doi:10.1109/ICSE-SEIP.2019.00021

[46] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343. doi:10.1109/TIT.1977.1055714

[47] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* 24, 5 (1978), 530–536. doi:10.1109/TIT.1978.1055934

Received 2026-01-30; accepted 2026-06-25