Exploiting Data-pattern-aware Vertical Partitioning to Achieve Fast and Low-cost Cloud Log Storage

JUNYU WEI, GUANGYAN ZHANG, and JUNCHAO CHEN, Tsinghua University, Beijing, China
YANG WANG, The Ohio State University, Columbus, United States
WEIMIN ZHENG, Tsinghua University, Beijing, China
TINGTAO SUN, JIESHENG WU, and JIANGWEI JIANG, Alibaba Group, Hangzhou, China

Cloud logs can be categorized into on-line, off-line, and near-line logs based on the access frequency. Among them, near-line logs are mainly used for debugging, which means they prefer a low query latency for better user experience. Besides, the storage system for near-line logs prefers a low overall cost including the storage cost to store compressed logs, and the computation cost to compress logs and execute queries. These requirements pose challenges to achieving fast and cheap cloud log storage.

This article proposes LogGrep, the first log compression and query tool that exploits both static and runtime patterns to properly structurize and organize log data in fine-grained units. The key idea of LogGrep is "vertical partitioning": it stores each log entry into multiple partitions by first parsing logs into variable vectors according to static patterns and then extracting runtime pattern(s) automatically within each variable vector. Based on such runtime patterns, LogGrep further decomposes the variable vectors into fine-grained units called "Capsules" and stamps each Capsule with a summary of its values. During the query process, LogGrep can avoid decompressing and scanning Capsules that cannot match the keywords, with the help of the extracted runtime patterns and the Capsule stamps. We further show that the interactive debugging can well utilize the advantages of the vertical-partitioning-based method and mitigate its weaknesses as well. To this end, LogGrep integrates incremental locating and partial reconstruction to mitigate the read amplification incurred by vertical-partitioning-based method.

We evaluate LogGrep on 37 cloud logs from the production environment of Alibaba Cloud and the public datasets. The results show that LogGrep can reduce the query latency and the overall cost by an order of magnitude compared with state-of-the-art works. Such results have confirmed that it is worthwhile applying a more sophisticated vertical-partitioning-based method to accelerate queries on compressed cloud logs.

CCS Concepts: • **Information systems → Information lifecycle management**; **Storage management;**

This article is an extended version of our article appeared in Eurosys '23 [77]. In this extended version, we present how to mitigate the read amplification incurred by vertical-partitioning-based method by incremental locating and partial reconstruction. We describe our observation of interactive debugging process and summarize its key features. We also show how we use indexed bitmap and matching matrix to achieve an effective incremental locating. As for evaluation, we present many new evaluation results of LogGrep, including the detailed incremental results, results on public logs, the benefits of indexed bitmap and partial reconstruction techniques, and the evaluation results of zstd-based LogGrep.

This work was supported by the National Natural Science Foundation of China under Grant 62025203.

Authors' addresses: J. Wei, G. Zhang (Corresponding author), J. Chen, and W. Zheng, Tsinghua University, Beijing, China; e-mails: wei-jy19@mails.tsinghua.edu.cn, gyzh@tsinghua.edu.cn; Y. Wang, The Ohio State University, Columbus, United States; T. Sun, J. Wu, and J. Jiang, Alibaba Group, Hangzhou, China.

ACM 1553-3077/2024/02-ART12
https://doi.org/10.1145/3643641

Additional Key Words and Phrases: Cloud log, data compression, full-text query, runtime pattern, static pattern

ACM Reference Format:
Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. 2024. Exploiting Data-pattern-aware Vertical Partitioning to Achieve Fast and Low-cost Cloud Log Storage. *ACM Trans. Storage* 20, 2, Article 12 (February 2024), 35 pages. https://doi.org/10.1145/3643641

1 INTRODUCTION

1.1 Motivation

Large-scale cloud systems log system events for several purposes, such as system modeling [11, 32], error diagnosing [19, 61, 82], user behavior profiling [21, 23, 50], and security attack detecting [24, 63]. Large cloud providers can easily generate up to PBs of logs per day [54, 70, 78], and thus often choose to compress these logs to reduce storage cost; furthermore, they sometimes need to query these compressed logs for the purposes discussed above.

We studied the log access pattern in Alibaba Cloud, a major cloud provider and our collaborator. We observed that these logs can be categorized into three types: *online logs* are mainly used for monitoring system states and are queried frequently; *near-line logs* are mainly used for debugging and thus are queried only when a problem occurs; after a certain period (typically 6–12 months [67, 78]), logs will be archived as *offline logs*.

The difference in their query patterns motivates different tradeoffs between storage costs and query latency. Online logs are queried frequently but do not need to be stored for a long time. They thus prefer methods that compromise storage cost for a lower query latency [8, 42]. Offline logs need to be stored for a long time but are rarely queried; they thus prefer methods that tradeoff query latency for a high compression ratio [13, 16, 39, 54, 55, 57, 71, 75, 78]. Near-line logs require (1) a high compression ratio like offline logs to reduce the storage cost, since it takes up a large part of cloud log storage and needs to be stored for a relatively long time, (2) a low query latency even though they are not queried as frequently as online logs. An engineer expects a query to finish as quickly as possible since a delayed query time will reduce productivity when debugging. During interviews, engineers from Alibaba Cloud conveyed their views that a query completion time of a few seconds is preferred and that a completion time of less than one minute is deemed acceptable. However, a level of dissatisfaction is reported for queries that consume several minutes or more.

We have tested several existing works, including **ElasticSearch** (**ES**) [8], CLP [67], and so on, and found none that achieve both goals. For example, it takes on average 14 minutes to execute a query using CLP, the state-of-the-art approach to query on compressed logs.

1.2 Contributions

This article aims at designing a compression and query method for near-line cloud logs with two objectives. First, it should minimize the overall cost including computation cost to compress logs, storage cost to store compressed logs, and computation cost to query logs. Second, it should limit query latency to an acceptable level.

To achieve both objectives, we adopt a classic idea called "vertical partitioning" in data processing systems. This idea is to break a single data entry into multiple fragments, compress similar fragments from different entries into a partition, and generate a summary for each partition to avoid decompressing irrelevant partitions when executing a query [15, 26, 51, 52, 73].

There are two key challenges in realizing such an idea in the log storage field. The first is *how to partition data entries during the compression process, such that the content in each partition shares*

common features, which will allow us to generate strict summaries to filter as many irrelevant partitions as possible. The second is *how to mitigate the read amplification incurred by the vertical-partitioning-based method*. When using such a method, fragments of each log entry are stored in multiple partitions and each partition contains fragments from different log entries. Once a log entry is to be reconstructed, we need to read and decompress all contents in the partitions containing the fragments of such log entry, which incurs read amplification.

To solve the first challenge, we follow the three steps below.

Structurizing logs by exploiting static patterns. We first leverage existing log parsing methods to structurize log entries into templates and variables [30, 38, 39, 54, 60, 74, 75, 78, 87], because values of the same variable are more likely to share common features [54, 78]. For example, if an application has a log output statement "printf("write to file:%s", filepath)", log parsing methods can parse a corresponding log entry into the template "write to file:" and a variable "filepath". Since the string template "'write to file:" is specified by the developer, we call the template a *static pattern* in the rest of this article. After parsing log entries, we organize values of the same variable (called a *variable vector*) into a partition. Compared to storing variable values following their original order in the logs [3, 67, 84], our method tends to store values that may share common features in the same partition, which is beneficial for both compression and generating strict summaries.

In our experiments, with a state-of-the-art log parser [78], this approach can reduce query latency by about 5.72× and improve the compression ratio by about 2.01×, compared with CLP. However, many queries still take more than one minute, which is still unacceptable. We find the main reason is that summaries generated for the whole variable vectors are often still too general to enable efficient filtering.

Further improving the filtering efficiency by exploiting runtime patterns. To further improve the filtering efficiency, our key idea is to exploit *runtime patterns* within each variable vector. Unlike a static pattern specified by a programmer, a runtime pattern is generated by the application at run time. In our previous example, all values of "filepath" may follow a pattern "/tmp/1FF8<*>.log", which is a runtime pattern.

We find runtime patterns are ubiquitous after exploring a wide range of production logs. These runtime patterns can help to filter keywords and we find they have a key feature: *a variable part of the same runtime pattern (referred to as a sub-variable) often includes limited types of characters and has a similar length.* For example, in our filepath runtime pattern "/tmp/1FF8<C_1>.log", values of the sub-variable vector C_1 all include 4 hexadecimal characters. By exploiting this key feature, we propose two optimizations. First, in addition to partitioning data into variable vectors, we further partition data into fine-grained sub-variable vectors and generate summaries on them. These summaries are stricter and allow more effective filtering. Second, we pad the values of each sub-variable vector to a fixed length to enable efficient keyword search and locating methods with minimal impact on compression ratio.

Extracting runtime patterns automatically. Extracting runtime patterns automatically, however, is challenging. General-purpose pattern extraction algorithms [4, 59, 62, 86] are too slow given the scale of our logs. As a result, prior works extract log patterns by (1) analyzing the source/binary code [9, 12, 80, 85], which only works for static patterns, (2) first splitting logs into tokens with user-defined delimiters and then regarding constant tokens among logs as patterns [30, 39, 45, 56], which works well for static patterns but poorly for runtime patterns, since runtime patterns are more versatile and cannot be tokenized with user-defined delimiters, or (3) setting default patterns or asking the developer to manually provide patterns [67], which is certainly not ideal.

To address this challenge, we design a novel runtime pattern extraction method based on the following observation: *variable vectors which do not include many duplicated values are usually*

dominated by a single runtime pattern. Following this observation, we first categorize variable vectors based on how many of their values are duplicated. We call variable vectors with a small percentage of duplicated values as *real variable vectors* and variable vectors with many duplicated values as *nominal variable vectors*. For real variable vectors, under the assumption that they only include one pattern, we design a *tree expanding approach* [44] to extract their patterns, which has $O(n)$ time complexity (n is the number of unique values in the sub-variable vector) and can extract finer runtime patterns. For nominal variable vectors, considering the fact that their values have many duplicates, and we only need to extract patterns on deduplicated values. The complexity of the pattern extraction algorithm presents less of a problem. Therefore, we design a *pattern merging approach* [39, 56], which has a time complexity of $O(n \log n)$ but can extract multiple patterns.

To solve the second challenge, we follow the two steps below.

Observations on interactive debugging session. We observe that when an error occurs, the engineers usually query the logs interactively and incrementally: they first query the logs with coarse-grained keywords, like "ERROR"; then they will browse the temporal distribution of hit entries and only check a small number of them to come up with more fine-grained queries, like "ERROR not 404", to narrow down the search space; they will repeat this procedure till they can locate the root cause (see a detailed example in Section 2.4). We call such a procedure *interactive debugging* and call multiple successive incremental queries from one engineer a *session*. Based on an analysis of 70,406,619 real-world debugging sessions in Alibaba Cloud, we find that 87.79% of them are interactive debugging sessions. Such interactive debugging can well mitigate the read amplification of the vertical-partitioning-based method: a later query only refines the locating result of the prior one and an engineer can only read a limited number of rows, and thus it is not necessary to reconstruct all hit entries.

Incremental locating and partial reconstruction. Based on such observation, we introduce *incremental locating* techniques to locate the positions of hit entries among different partitions incrementally. Such a technique is based on a well-designed structure called *Indexed Bitmap*. This structure can record and maintain the previous locating results, based on which the following locating process can only check and prune previous hit entries. Indexed Bitmap enables checking and pruning with a complexity of $O(1)$ by utilizing an index array and fixed-length padding. We also introduce *partial reconstruction* to only return the locating results and limited reconstructed entries. This is enough for interactive debugging and will significantly mitigate the read amplification incurred by the vertical-partitioning-based method.

Based on these ideas, we have designed and implemented LogGrep, a tool that can compress logs with a high compression ratio and support Linux grep-like commands on these compressed logs. On 21 Alibaba Cloud production logs and 16 public logs, we compare LogGrep with CLP [67], the state-of-the-art method to compress and execute text query on logs, ES [8], a method focusing more on query latency, and gzip+grep, the current method used by Alibaba Cloud. Our evaluation shows that first, LogGrep can usually complete a query within a minute: this is an order of magnitude faster than CLP and gzip+grep, and is comparable to ES. Second, by considering the storage and computation cost in Alibaba Cloud, the overall cost of LogGrep is 36% of CLP, 7% of ES, and 34% as much as that of gzip+grep.

Our contributions can be briefly summarized as the five points below.

— We propose LogGrep, the first vertical-partitioning-based log compression and query tool that structurizes log data in fine-grained units. We demonstrate that the proper structurization method enables simple but effective summaries to accelerate queries on compressed log data.

— To the best of our knowledge, LogGrep is the first one to extract runtime patterns automatically to improve the data filtering efficiency. To achieve that, we propose a novel runtime pattern extraction method by separating real and nominal variable vectors.
— We observe that the pattern of interactive debugging can well mitigate the weakness of vertical-partitioning-based log storage. We introduce incremental locating and partial reconstruction techniques to mitigate the read amplification during locating and reconstruction procedures, respectively.
— We evaluated LogGrep on 21 real-world production logs [5] and found LogGrep achieves a significant query latency reduction and a considerable cost saving over the state-of-the-art system.
— We make LogGrep open-sourced [6].

1.3 Organization

The rest of the article is organized as follows. Section 2 shows how we came up with the key idea of LogGrep. Section 3 gives an overview of LogGrep. Section 4 presents the process of exploiting both static and runtime patterns to structurize logs, and puts particular emphasis on the runtime pattern extracting approaches. Section 5 depicts the matching process on cloud logs structurized by the static and runtime patterns as well as the design of the Indexed Bitmap to achieve efficient incremental locating. Section 6 presents experimental results and Section 7 discusses related work. Finally, Section 8 concludes the article.

2 ROAD MAP OF KEY IDEA

In Alibaba Cloud, applications first write raw logs, usually in text format, into 64 MB blocks. We call these blocks *log blocks*. Then Alibaba Cloud compresses these log blocks in the background. During a debugging procedure, an engineer first starts a session and then sends query commands. A query command can contain multiple search strings concatenated by classic logical operators. Alibaba Cloud engineers execute queries on either the raw log blocks, if they have not been compressed yet, or on compressed log blocks. Since a log block will usually be compressed soon after it is generated, queries on compressed log blocks are much more common.

This section will first present our observations on the data pattern and our attempts to partition logs. Then we present our observation on the query pattern of the interactive debugging process and show how we utilize such a pattern to mitigate the weakness of the vertical-partitioning-based method.

2.1 Prior Work: Structurizing with Static Patterns

A naive way to query on compressed logs is to decompress the logs first and then run standard tools like grep [17]. However, this approach incurs a long query latency.

We first present CLP [67], the state-of-the-art work to query directly on compressed logs, since our work borrows several ideas from CLP. We discuss other related works in Section 7.

By using both generic and user-specified rules, CLP first identifies several templates, which correspond to the format string the application uses to generate logs (e.g., "filename=%s" in a printf statement). During the compression phase, CLP splits each log entry into tokens using specific delimiters. By comparing these tokens with templates, CLP can determine whether a token is from the static part of a template or a variable in the template. To encode each log entry, CLP replaces the static part of its template with an identifier and stores its variable tokens. Finally, CLP compresses all encoded log entries in a log block into a log segment.

During the query phase, CLP can search a string in the log. It first tokenizes the search string into several keywords using the same delimiters. Then it starts from the first keyword: for a log

entry, it will search both the static part of its template and the variables to see whether any can match the keyword. Once it finds a match, it will continue to match the following keywords from the matched position.

CLP adopts the idea of data partitioning and filtering to optimize the query process. It builds an inverted index [14] for the static part of each template to record which log segments contain the corresponding static part; based on both generic and user-specified rules, it stores certain variables in a dictionary and builds an inverted index for distinct values of these variables as well. When matching keywords on log segments, CLP will use the inverted index to decide which log segments may contain a target keyword and filter unrelated segments. Although this approach significantly improves query speed on some datasets compared with the naive approach, its query latency is still not satisfactory.

2.2 First Attempt: Fine-grained Partitioning and Summaries Based on Static Patterns

To avoid the time-consuming decompression process, we adopt the idea of "vertical partitioning" in data processing systems [15, 26, 51, 52, 73]. This idea is to (1) break each log entry into multiple fragments, (2) compress similar fragments into a partition, and (3) build summaries on each data partition to filter as many partitions as possible when executing a query.

Such an idea poses two questions for compression, namely, how to partition data and how to create summaries. Ideally, data from different partitions should have different characteristics and their summaries should be able to capture such differences. Besides, our method should not impair the overall compression ratio. For example, making partitions smaller is usually good for the purpose of filtering, but may hurt the compression ratio [39, 54].

Motivated by the observation that values of the same variable often share similar characteristics, we propose the following design. First, for a log block, we store values of the same variable into a partition, instead of storing values from different variables consecutively (i.e., CLP approach). We call these data partitions as *variable vectors* and all variable vectors of the same static pattern form a *group*. For example, in Figure 1, the log block is parsed into two groups and four variable vectors. Second, we generate a summary to capture the type and the maximal length of the values in each variable vector. We compute the type number based on whether values of a variable vector include a decimal integer, a hexadecimal integer, an alphanumeric string, or something else. Following this idea, we represent the type number using six bits, each of which represents whether the values include characters from the following groups: 0-9, a-f, A-F, g-z, G-Z, and "other".

Such a combination is effective in creating strict summaries in our experiments: a variable vector has 3.1 types of characters on average and its length variance is 66.1 on average; if we generate the same summary on the whole log block instead of on each variable vector, each log block has almost all six types (5.8 types) of characters on average and its length variance is 198.5 on average. Furthermore, compressing each variable vector individually improves the compression ratio by 2.01× compared with CLP, because values of the same variable vector have more similarity.

Our query procedure is similar to that of CLP (Section 2.1), except that when searching a keyword in variable vectors, we skip variable vectors whose summaries do not match (part of) the keyword. Such optimizations bring a significant improvement: compared with CLP, this attempt can reduce query latency by 5.72×. However, queries on larger log datasets still take more than one minute, which is unacceptable.

2.3 Opportunity: Runtime Patterns

To further partition logs into finer-grained partitions, we propose the key idea of our work: *structurizing and filtering logs by exploiting both static and runtime patterns*, which can achieve fast

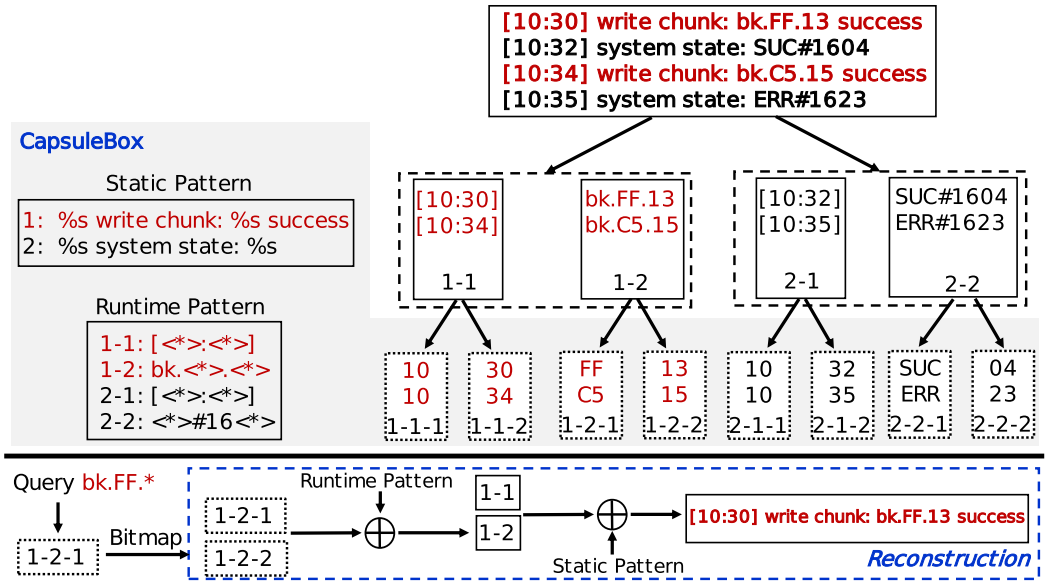


Fig. 1. Fine-grained storage format and reconstruction process. The log block shown above is structurized as four variable vectors in two groups, and these vectors are further structurized as eight Capsules. A user query (e.g., "bk.FF.*") first locates the hit positions and then reconstructs the original log entry.

and cheap cloud log storage. This idea is motivated by our following observations about runtime patterns.

Runtime pattern exists widely in variable vectors. We observe values in the same variable vector tend to have runtime patterns. For example:

- Variables like block numbers may have a fixed prefix. For example, block numbers in HDFS follow a fixed pattern: "blk_<*>".
- Values of a numerical variable in the same log block may all fall into a specific range. For example, time stamps in January of 2021 follow a fixed pattern: "[2021-01-<*> <*>:<*>:<*>.<*>]".
- Values of a variable vector like file paths and IP addresses in the same log block may all come from a common root path or the same sub-network. For example, file paths of the same log block in Log A from Alibaba Cloud follow a pattern: "/root/usr/admin/<*>" and IP addresses of the same log block in Log G from Alibaba Cloud follow a pattern: "11.187.<*>.<*>".

Intuitively, these runtime patterns can help filtering as well, in ways similar to static patterns. We call each "<*>" in a runtime pattern as a *sub-variable*. All values of the same sub-variable in a variable vector form a *sub-variable vector*. For example, in Figure 1, variable vector "1-2" is decomposed as two sub-variable vectors "1-2-1" and "1-2-2".

Values of the same sub-variable vector include limited types of characters and have similar lengths. On our production logs, a sub-variable vector has 1.5 types of characters on average and its length variance is 32.5 on average. Compared to 3.1 types of characters and a length variance of 66.1 on whole variable vectors, this is a significant reduction. As a result, we can build stricter summaries and filter keywords more effectively.

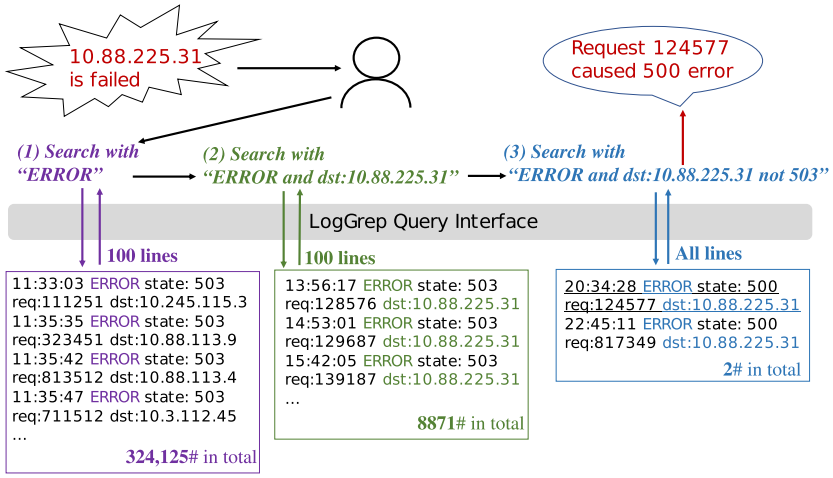


Fig. 2. A usage example of LogGrep to trace root cause by debugging interactively.

2.4 Opportunity: Interactive Debugging on Near-line Logs

In addition to the challenges for compression, the vertical-partitioning-based method also poses a challenge of read amplification for query. Each log entry is now stored into several partitions while each partition contains sub-variables from different log entries. Once a log entry is to be reconstructed, sub-variables from many other entries need to be decompressed. Fortunately, we find the query pattern of near-line logs can well mitigate such weakness.

In Alibaba Cloud, near-line logs are mainly used for interactive debugging. We show an example of such a process in Figure 2. The process is launched as a session between a human engineer and the query engine. Assuming an error occurring on server 10.88.225.31 is detected, the engineer needs to query the log to find relevant log entries. Due to the variety of cloud log formats, the engineer may not know the exact query commands to trace the bug at the beginning. Instead, a broad query, such as "ERROR", is first performed. A broad query of this nature typically results in an excessively high number of log entries (324,125 in this example). However, when the engineer scans a subset of the results, the detailed format of the error messages is provided, which facilitates a further refined query. In our example, the engineer finds the "dst:xxx" can be used to designate the target server, and then sends the second query "ERROR and dst:10.88.225.31". The second query narrows down the search scope but still returns too many results (8,871 in this example). By scanning a subset, the engineer finds "state:503" dominates. 503 is a common error state caused by temporary network congestion and has little relevance with the error. To exclude these cases, the engineer further refines the query to "ERROR and dst:10.88.225.31 not 503", which returns only 2 log entries. From the results, the engineer knows the error was first caused by Request 124577 with state 500.

Based on such process, we make following key observations:

— During the locating procedure, a session usually includes a series of queries, in which a later one refines the result of the prior one, usually by adding new conditions. As a result, we only need to check previous hit results for the subsequent queries.

— During the reconstruction procedure, although a query may return many entries, the engineer only needs to know the locating results and read a limited subset of the results to further narrow down the searching scope. As a result, we can partially reconstruct the hit entries to mitigate the read amplification incurred by the vertical-partitioning-based method.

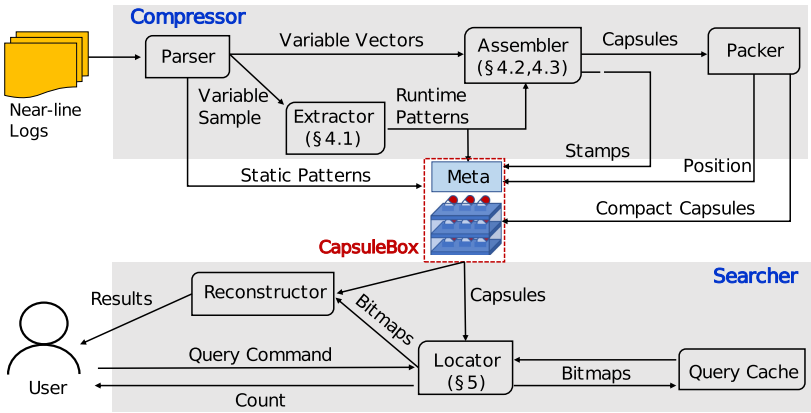


Fig. 3. LogGrep architecture.

3 SYSTEM OVERVIEW

Based on our key idea, we designed a system called LogGrep. Figure 3 shows the detailed architecture of LogGrep. We first present the query interface of LogGrep. Then we introduce the compression, query, and reconstruction workflow to give an overview of the system.

3.1 Query Interface

In general, log analysis is executed in two phases: in the first phase, the engineer may send grep-like query commands to find log entries that may be relevant to the error. In the second phase, the query result will be passed to another system, which performs more sophisticated analysis like anomaly detection, structure-based aggregation with SQL, and so on. We focus on the query of the first phase since it may need to scan all logs and thus is the bottleneck.

Targeting this phase, LogGrep provides a grep-like query interface on text log: as we have mentioned that a query can contain multiple search strings concatenated by classic logical operators; a search string can contain wildcard characters, but LogGrep assumes a wildcard character will not include token delimiters (like space and comma) or line breaks—in other words, an engineer can only perform a wildcard search within a single token. To give a concrete example, a query in LogGrep may look like "error AND dst:11.8.* NOT state:503". LogGrep does not support more complicated operations like join, aggregation, and so on. These operations are performed in the second phase.

When encountering a new error, an engineer may start a session and build the query command incrementally in an interactive manner. As for a known error, the engineer may directly launch a complete query command from a prior session. LogGrep thus has two query modes. We use *refining mode* to denote the former query experience and *direct mode* to denote the latter.

3.2 Workflow

Compression. For each log block, LogGrep first samples a subset (5% in our experiments) of its log entries, and identifies static patterns on this sample using the *Parser* adopted by LogReducer [78]. Then, with the help of those static patterns, it parses all the log entries into variable vectors. For each variable vector, likewise, LogGrep first samples a subset of its values, and extracts runtime patterns automatically on this sample (Section 4.1) with the *Extractor*. Then the *Assembler* decomposes the whole variable vector into several fine-grained units called "Capsules" accordingly (Section 4.2) and generates a Capsule stamp for each Capsule (Section 4.3). Finally,

the *Packer* compresses and packs all Capsules into a compressed file called "CapsuleBox" using LZMA [88], which is reported to have a high compression ratio [67, 78]. During the procedure, the Packer pads all values of the same Capsule to the same length, which can vastly improve query speed with minimal impact on compression ratio. As shown in Figure 1, a CapsuleBox includes all compressed Capsules belonging to this log block, as well as their metadata including static and runtime patterns.

Query. LogGrep first parses a query command into several search strings and tokenizes each search string as several keywords. Each keyword can be a part of static/runtime patterns or within a Capsule. The *Locator* executes a keyword matching process with the help of runtime patterns to filter unrelated Capsules (Sections 5.1 and 5.3). It executes fixed-length matching within decompressed Capsules (Section 5.4) and incrementally merges results from different Capsules (Section 5.2).

To achieve incremental locating, we (1) use an *Indexed Bitmap* to prune and units results effectively and (2) use a *Query Cache* to store the results of previous queries in a hashmap. The key of the hashmap is a query, and the value is the Indexed Bitmap recording the result of the corresponding query. Before querying a search string, the Locator will first check the Query Cache to see if LogGrep can reuse a previous result. If the current query refines the condition of a previous query, LogGrep only needs to check previous hit entries. When a debugging session terminates, LogGrep clears the corresponding Query Cache. When a Query Cache becomes full, LogGrep evicts its items in a first-in-first-out manner.

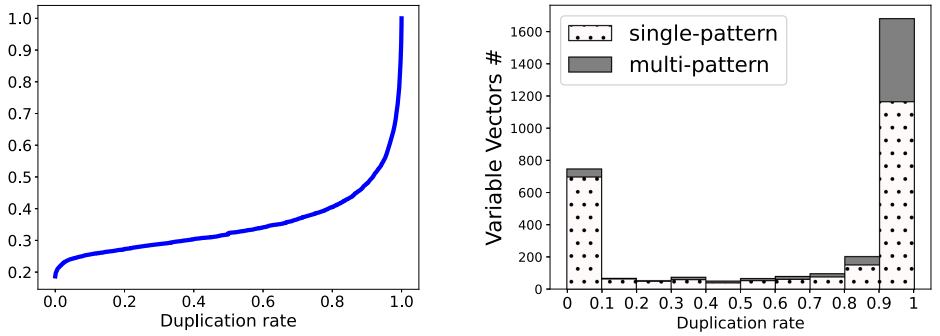
Reconstruction. The locating process returns all entries matching the query, LogGrep reconstructs the original log entries with a *Reconstructor*. To achieve that, the Reconstructor first decompresses all relevant Capsules. For example, as we show in Figure 1, supposing i^{th} entry within Capsule "1-1-1" is hit, the Reconstructor needs to decompress "1-2-1" and "1-2-2". It then fetches the i^{th} value of each Capsule. Since the Packer pads values to the same length in each Capsule, fetching the i^{th} value is an operation with a complexity of $O(1)$. Then the Reconstructor finds the corresponding static pattern and the runtime pattern according to the metadata in CapsuleBox. Finally, the Reconstructor fills values into the patterns to rebuild the original log entries.

If the locating process returns multiple entries, the Reconstructor needs to order them after reconstructing them. The entries of the same static pattern are naturally ordered since LogGrep stores values in the same variable vector according to their original order appearing in the log block. For entries of different static patterns, the Reconstructor merges them based on their timestamps to restore the global order. If there is no timestamp in logs, LogGrep needs to add logical timestamps to log entries before compressing them, but so far, we have not implemented this mechanism since all logs in Alibaba Cloud have timestamps.

We apply partial reconstruction by only reconstructing hit entries from the latest log blocks to avoid read amplification on other blocks. By default, the Reconstructor will only reconstruct the latest 100 hit entries. The engineer can ask for more entries or send another query.

4 LOG STRUCTURIZATION PROCESS

After structurizing logs into variable vectors based on their static patterns, we achieve further improvement by exploiting runtime patterns. Log structurization with runtime patterns has three steps: (1) Extract runtime pattern(s) within each variable vector; (2) Decompose each variable vector into fine-grained Capsules; (3) Generate a stamp for each Capsule, which can help to filter irrelevant Capsules during query execution. We discuss these three steps in this section.



(a) CDF of duplication rate.

(b) Distribution of single-pattern and multi-pattern variable.

Fig. 4. Distribution of single-pattern and multi-pattern variable vectors in respect of duplication rate.

4.1 Runtime Pattern Extraction

Since runtime patterns are more versatile, we can not split the keyword and the variable into tokens by pre-defined delimiters as we do for static patterns. We test many general-purpose pattern extraction methods [4, 59, 62, 86], and find they are slow given the scale of production logs. We thus need to design an efficient extraction method targeting runtime patterns specifically.

Our design is motivated by our observation that most of the variable vectors are dominated by one pattern. If we know that a vector only has one pattern, it is possible to design a more efficient algorithm. In the rest of this section, we first present the heuristic rule we use to determine whether a vector is dominated by one pattern. Then we present the pattern extraction algorithms for different categories of vectors respectively.

Heuristic rule for variable vector categorization. We extract patterns with a general-purpose extraction method [62] on samples of 13,238 variable vectors from 37 cloud logs (21 from Alibaba Cloud, 16 from a public dataset [40]). We find these variable vectors can be generally divided into two categories. Values of the first category, such as block number, time stamp, and request number, do not repeat but tend to have one common pattern. Values of the second category, such as file path, user name, and error code, may repeat many times but their unique values may have multiple patterns. We use the duplication rate to measure the repetition of values in a variable vector, this rate is defined as follows: assuming we have a variable vector $\vec{V}$, we denote the count of all values within the vector as $|\vec{V}|$ and denote the count of unique values within the vector as $|Set(\vec{V})|$. The duplication rate of this variable vector is $(|\vec{V}| - |Set(\vec{V})|)/|\vec{V}|$. Based on our observation, the duplication rate can serve as a heuristic metric to determine whether a variable vector is dominated by one pattern.

We present the distribution of duplication rate of all these variable vectors in Figure 4(a), we find it has a bathtub distribution with most vectors falling into range [0, 0.2] and range [0.8, 1.0]. We call a variable vector a "single-pattern vector" if one pattern can cover at least 90% of its values. And if not, the variable vector is called a "multi-pattern vector". Figure 4(b) presents the correlation between the likelihood that a variable vector is a single-pattern vector and its duplication rate. As shown in the figure: (1) most of the vectors with a low duplication rate are single-pattern vectors, and (2) vectors with a high duplication rate can be single-pattern vectors or multi-pattern vectors. Based on this observation, we propose our heuristic rule: we apply a single-pattern extraction method for low-duplication-rate vectors and a multi-pattern extraction method for high-duplication-rate vectors. Note this heuristic rule may apply an expensive multi-pattern extraction method for single-pattern vectors with a high duplication rate. However, this is not too bad:

considering pattern extraction only needs to be applied to unique values, the high duplication rate means there are fewer values to analyze, which reduces the overhead.

We need a threshold to separate low and high duplicate rates. Considering the bathtub-like distribution in Figure 4(a), the efficiency of our approach is not very sensitive to this threshold, as long as it is somewhere in the middle. In this article, we choose 0.5 as the threshold. We use *real variable vectors* to denote those with a duplication rate under 0.5 and *nominal variable vectors* to denote those with a duplication rate greater than or equal to 0.5. We design different methods to extract runtime patterns for these two types of variable vectors, which are discussed next.

Note that the accuracy of pattern extraction does not affect the correctness of our system: if a value does not match any found patterns, our system will store it in an outlier partition; any query needs to scan the outlier partition. Therefore, if our heuristic rule fails, it will only affect the performance of compression and query, but will not cause data loss or wrong query results.

Tree expanding approach for real variable vectors. Under the assumption that a real variable vector is dominated by one pattern, we design a tree expanding approach [44] to extract its runtime pattern by building and fully expanding a pattern tree (i.e., expanding all leaf nodes in the tree recursively).

In the beginning, LogGrep constructs a sample set by choosing 5% values and puts all unique values in them in a root node (e.g., vector #1 in Figure 5(a)). Then LogGrep tries to expand the tree with multiple iterations. During an iteration, for each leaf node, if it is not marked as unsplittable, LogGrep chooses a delimiter, by either using a non-alphanumeric character from a randomly picked value, such as "_" in vector #1 in Figure 5(a), or the **longest common sub-string** (**LCS**) between two randomly picked values, such as "F8" in vector #3 in Figure 5(a). Then LogGrep tests if this delimiter can split the leaf node, namely, at least 95% of values containing the delimiters. LogGrep tries each delimiter three times by extracting the delimiter from different randomly picked values, and if all fail, LogGrep marks the corresponding leaf node as unsplittable. If all leaf nodes are unsplittable, the expanding process terminates.

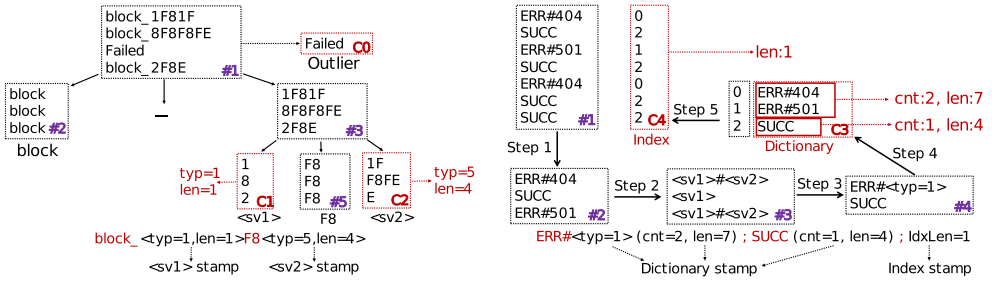
We choose non-alphanumeric characters as delimiters since we find non-alphanumeric characters tend to split a value into parts with different semantic information. We choose LCS since values in variable vectors of the same log block tend to share a common sub-string.

At the end of tree expanding, if all values in a leaf node are the same, LogGrep represents the leaf node as a constant sequence (e.g., node #2 "block" and #5 "F8" in Figure 5(a)). Otherwise, the leaf node constitutes a sub-variable. Finally, LogGrep concatenates all these constant sequences and sub-variables to build the runtime pattern of the corresponding variable vector.

The complexity of this algorithm is $O(n)$, where n is the number of values in the root node because each iteration needs to scan all values in the root node and the number of iterations is determined by the number of sub-variables in the runtime pattern, which do not correlate with the number of values in the root node and can be regarded as constant.

Pattern merging approach for nominal variable vectors. According to our observation, nominal variable vectors have fewer unique values, but these values may have multiple patterns. As a result, we design a pattern merging approach [39, 56] for nominal variable vectors.

LogGrep first builds a temporary vector (vector #2 in Figure 5(b)) by only retaining unique values in the original variable vector. Then LogGrep splits each value in the vector into several sub-variables by using non-alphanumeric characters as delimiters to generate a "pattern sketch" vector (vector #3 in Figure 5(b)). Then LogGrep merges those pattern sketches with the same form. If a sub-variable has a constant value in all values of the same pattern sketch, LogGrep will represent the sub-variable as a constant in the final pattern. For example, "<sv1>" in "<sv1>#<sv2>" of vector #4 in Figure 5(b) is a constant "ERR". Finally, LogGrep reorders the temporary vector by storing



(a) Real Variable Vectors. (b) Nominal Variable Vectors.

Fig. 5. Runtime pattern extraction process. Extracted runtime patterns and Capsule stamps are shown at the bottom.

all values with the same pattern sequentially to generate the dictionary vector. It assigns a unique index number to each value in the dictionary vector and generates an index vector by replacing original variable values with the corresponding index numbers.

The complexity of this algorithm is $O(nlogn)$, where n is the number of values in the dictionary vector: LogGrep generates a pattern sketch for each value, and to store all values of the same sketch sequentially, LogGrep sorts all these sketches.

4.2 Variable Vector Encapsulation

Based on the extracted runtime patterns, LogGrep can further decompose variable vectors into Capsules. Each Capsule will be compressed independently and compactly.

For real variable vectors, LogGrep stores all sub-variables of the same place in a runtime pattern as a sub-variable vector and compresses it as a Capsule (e.g., Capsules "C1" and "C2" in Figure 5(a)). If some values do not match this runtime pattern, LogGrep puts them in an outlier vector and stores it as another Capsule (e.g., Capsule "C0" in Figure 5(a)).

For nominal variable vectors, LogGrep does not further split the dictionary vector into smaller sub-variable vectors, like it does for real variable vectors, because the dictionary vector is usually small enough. In this case, the additional metadata overhead of further splitting may overcome the benefit of splitting. Therefore, LogGrep encapsulates each nominal variable vector into two Capsules, one for the dictionary vector and the other for the index vector (Capsule "C3" and "C4" in Figure 5(b)).

4.3 Stamping Capsules

LogGrep generates a Capsule stamp for each Capsule, which includes a type number and the maximal length of the values in the Capsule. During log query, LogGrep will use runtime patterns and stamps to determine whether a keyword *may* match a value in the Capsule. If not, LogGrep avoids decompressing and scanning the corresponding Capsule.

For a Capsule, LogGrep represents its type with a six-bit type number: the first bit (the most significant bit) represents whether the values include characters 0-9; the second bit represents whether the values include characters a-f; the third bit represents whether the values include characters A-F; the fourth bit represents whether the values include characters g-z; the fifth bit represents whether the values include characters G-Z; and the sixth bit represents whether the values include other characters. The only exception is the index vector in a nominal variable vector: since an index vector only contains 0-9, LogGrep does not compute and store its type number.

LogGrep computes the maximal length in different manners for different types of Capsules. For a sub-variable vector and an index vector, LogGrep computes the maximal length of its values. For

a dictionary vector of a nominal variable vector, LogGrep computes the maximal length of values belonging to each pattern.

Furthermore, for a dictionary vector, LogGrep calculates the count of values belonging to each runtime pattern, which is used to accelerate queries (Section 5.4).

Example of real variables vectors. For sub-variable vector "C1" in Figure 5(a), since it only contains 0-9, its type number is 000001b=1; for sub-variable vector "C2", since it contains 0-9 and A-F, its type number is 000101b=5. Then LogGrep attaches the type number and the maximal length of each sub-variable vector to the runtime pattern. Now the pattern becomes "block_<type=1,len=1>F8<type=5,len=4>".

Example of nominal variable vectors. For "ERR#<*>" in Figure 5(b), LogGrep finds the type number of the sub-variable "<*>" is 000001b=1, since it only contains digits. Furthermore, LogGrep finds runtime pattern "ERR#<type=1>" has two values and the maximal length is seven. Therefore, the patterns become "ERR#<type=1> (cnt=2, len=7); SUCC (cnt=1, len=4); IdxLen=1".

5 LOG QUERY PROCESS

As we have discussed in Section 3, given a query command, the log query process locates the hit entries on all compressed logs and represents their positions in a set of Indexed Bitmaps. In this section, we first present an overview of such query process in Section 5.1, and then show the structure of Indexed Bitmap, which can support incremental locating in Section 5.2. Then we elaborate on two specific procedures during the query. Namely, how does LogGrep decompress as few Capsules as possible by matching with the runtime pattern in Section 5.3 and how does it match within Capsules in Section 5.4.

5.1 Query Process Overview

Algorithm 1 presents an overview of the query process. We first discuss its main logic and proceed to discuss its functions.

Main logic of query process. The query algorithm takes a query command, the zipped "CapsuleBox", static and runtime patterns, a set of pre-defined delimiters, and the bitmap cache as its input. It first checks the query command in the cache. If the command can be found, it will reuse the previous query result (lines 1–2). Note if only parts of the command are found, LogGrep will reuse the query results of these parts and merge them into the final result.

This algorithm splits the query command into several search strings using logical operators (line 3). Then it queries each search string on each static pattern and records the query result in an Indexed Bitmap, which corresponds to all hits on this static pattern. The search string is split as a set of keywords ($\mathbb{K}$) by pre-defined delimiters (line 5). The static pattern to be queried is split into a token set $\mathbb{T}$ by the same delimiters (line 7). A token in $\mathbb{T}$ can be a constant token or a variable vector. The token set and the keyword set will be used to generate a static pattern matching matrix $\mathcal{M}$ (line 9), and each cell in $\mathcal{M}$ corresponds to a matching process of a keyword on a variable vector.

After obtaining the static pattern matching matrix, the algorithm will process each cell by matching keyword k_n on the runtime pattern $\mathcal{R}_m$ of the corresponding variable vector (line 13), which generates a runtime pattern matching matrix M_{mn}. Each cell in M_{mn} corresponds to a matching process of part of the keyword in a Capsule. Merging all results in matrix M_{mn} with the operations of incremental pruning and union using the Indexed Bitmap delivers the result of cell (t_m, k_n).

The result of a search string is generated by merging all results in $\mathcal{M}$ (line 15). The searching result of a query command on a static pattern is generated by merging the results of search strings according to the logical operators (line 16). Finally, the algorithm stores these results in the cache and returns $\mathbb{B}$, i.e., a set of Indexed Bitmaps.

ALGORITHM 1: Pseudo code of log query algorithm in LogGrep

Input: Query command Q, Capsule box $\mathbb{P}$, Static pattern set $\mathbb{S}$, Runtime pattern set $\mathbb{R}$, Bitmap cache $\mathbb{C}$, Pre-defined delimiter set $\mathbb{D}$
Output: Indexed bitmap set $\mathbb{B}$

```
if 𝔹 ← Cache_Query(Q, ℂ) ≠ ∅ then
    return 𝔹
ℚ ← Split Q by logic operators    // ℚ = [q0,q1,...,qn]
for q_j in ℚ do
    𝕂_j ← Split q_j by delimiters in 𝔻    // 𝕂_j = [k0,k1,...k_l]
    for S_i in 𝕊 do
        𝕋_i ← Split S_i by delimiters in 𝔻    // 𝕋_i = [t0,t1,...,t_p]
        // Generate static pattern matching matrix M_ij
        M_ij ← Match_on_StaticPattern(𝕋_i, 𝕂_j)
        for (t_m, k_n) in M_ij do
            ℛ_m ← Load the runtime pattern of t_m in ℝ
            // Generate runtime pattern matching matrix M_mn
            M_mn ← Match_on_RuntimePattern(ℛ_m, k_n)
            𝕄_ij ← 𝕄_ij ⋃ M_mn
        B_ij ← Merge_StaticPattern_Result(M_ij, 𝕄_ij, ℙ)
        B_i ← Merge_Logic_Result(B_i, B_ij)
Save_Bitmap(𝔹, Q, ℂ)
return 𝔹
Function Match_on_StaticPattern(𝕋, 𝕂)
    for t_i in 𝕋 do
        if Match_Keyword(t_i, k0) then
            for j in [0, l] do
                if t_{i+j} is constant then
                    if t_{i+j} ≠ k0 then
                        Delete M_i
                        break
                else
                    M_i ← M_i ⋃ (t_{i+j}, k_j)
    return M
Function Merge_StaticPattern_Result(M, 𝕄, ℙ)
    for M_i in M do
        for (t_i, k_j) in M_i do
            M_ij ← Load matching matrix from 𝕄
            ℬ_i ← Bitmap_Pruning(B_i, Merge_RuntimePattern_Result(M_ij, ℙ))
        ℬ ← Bitmap_Union(ℬ, ℬ_i)
    return ℬ
Function Merge_RuntimePattern_Result(M, ℙ)
    for M_i in M do
        for m_ij in M_i do
            ℬ_i ← Bitmap_Pruning(B_i, Match_within_Capsule(m_ij, ℙ))
        ℬ ← Bitmap_Union(ℬ, ℬ_i)
    return ℬ
```

Matching process on static and runtime patterns. The matching process on a static pattern takes a set of tokens and a set of keywords as its input (line 19). It first checks if a keyword k_0 can match a token in the pattern (line 21). If it matches, we regard this as a starting point and further try to match subsequent keywords with tokens. If a token is constant, it checks if the keyword matches this constant token. If it is a variable vector, this matching process generates a cell in the matching matrix. All cells of a starting point form a row in the matrix. Note that all keywords may be successfully matched with the constant tokens. In this case, the process returns a special flag indicating a full match on all checked entries, thereby eliminating the need to generate a matching matrix.

The matching process on runtime patterns also generates a matching matrix. This process is more complex than the one on static patterns since the delimiters cannot be pre-defined. This process will be discussed in Section 5.3.

Result merging on matching matrix. Given a matching matrix, we execute a matching process for each cell to generate an Indexed Bitmap and merge these bitmaps by pruning and union to get the final result. For runtime pattern matching matrix, LogGrep matches part of a keyword within a Capsule (line 40 and see Section 5.4 for details) to get the result of one cell, and then it prunes results of the same row and unites results from different rows (line 41) to get the query result of the matrix. Such result corresponds to a cell in static pattern matching matrix. LogGrep then executes a similar merging process to get the query result of a search string (lines 34–35).

5.2 Key Data Structure: Indexed Bitmap

We store the query result on all log entries of the same static pattern in an Indexed Bitmap. An Indexed Bitmap contains two parts: a bitmap and an index array. The bitmap contains the same amount of bits as the queried entries. If the corresponding log entry is hit by the query command, the bitmap records a "1", if not, it records a "0". The index array stores all positions recording "1" in the bitmap.

Such a structure is used to record temporary results. There are two basic operations to merge results: pruning and union. The pruning operation is used to incrementally locate hit entries by checking a condition within all positions recording "1" in current bitmap and further changing some bits to "0" to narrow down the hit scope. The union operation is to calculate the union result of two bitmaps and update the index array according to the result.

As shown in Figure 6, we reduce the overhead of pruning by directly writing the hit positions that remain after pruning (1, 7, 1, 5 in the figure) to the head of the index array. Although this may cause duplicate positions, we can tolerate such duplication by recording the hit counts. We reduce the overhead of union by directly appending new positions into the index array by getting aware that it is unnecessary to sort this index array. Such awareness is from the observation that there are few conditions during the log query to calculate the intersection of two temporary results. This is because after storing the original log entry into several fine-grained Capsules, we do not need to search each Capsule from scratch and calculate their intersection, instead, we can prune previous results and incrementally narrow down the hit scope.

5.3 Matching and Filtering with Runtime Patterns

By matching the keyword on runtime patterns, LogGrep locates all relevant Capsules and obtains a matching matrix as discussed above (line 13 in Algorithm 1). In this subsection, we first show such matching process for real variable vectors and proceed to discuss the differences for nominal variable vectors. Given a keyword and a runtime pattern, depending on the position of the keyword in an original variable value, LogGrep may need to check whether the keyword is a prefix, a suffix,

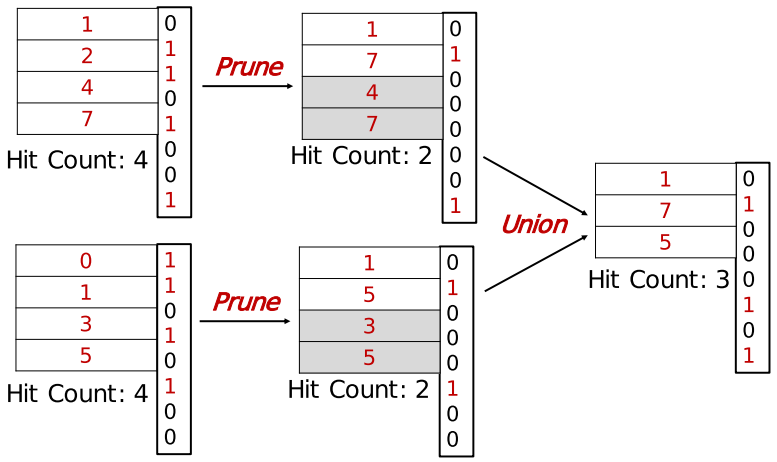


Fig. 6. Basic operations on the Indexed Bitmap structure. Two Indexed Bitmaps are first pruned to narrow down each hit scope and are finally united as one Indexed Bitmap.

or a sub-string of any of the values following this runtime pattern. The algorithm for sub-string matching is presented in isolation since it is the most general case.

The algorithm of the matching process is as follows: (1) LogGrep will try to match the keyword to each sub-variable vector in the runtime pattern since the keyword may be fully contained in one sub-variable vector (such as the matching case ① and ⑤ in Figure 7). (2) LogGrep will try to match the keyword to each constant string in the runtime pattern since the keyword may be fully contained in the runtime pattern: if the keyword is a sub-string of the constant, then all values of this variable vector will contain the keyword and thus they all match the keyword. Otherwise, we have the following three cases:

— Head case: The suffix of the constant is a prefix of the keyword (such as the matching case ④ in Figure 7). LogGrep will execute the prefix matching process recursively to check whether the remaining suffix of the keyword is a prefix of any of the values following the remaining part of the runtime pattern.
— Tail case: The prefix of the constant is a suffix of the keyword (such as the matching case ② in Figure 7). LogGrep will execute the suffix matching process recursively to check whether the remaining prefix of the keyword is a suffix of any of the values following the remaining part of the runtime pattern.
— Body case: The constant string is a sub-string of the keyword (such as the matching case ③ in Figure 7). LogGrep will recursively execute a prefix matching process and a suffix matching process. Finally, it will calculate the intersection of these two matching results.

In a possible match, we require a certain Capsule to have a certain value to match a sub-string of the keyword. LogGrep will search the corresponding Capsule to check if there is an actual match (such as match "F8F" in sub-variable vector "<sv2>" in matching case ④ in Figure 7), which generates a cell in the matching matrix.

Before the decompression of a candidate Capsule, LogGrep will first check its stamp and only decompress it if the sub-string of the keyword does not violate the recorded restrictions. The type number recorded in a Capsule stamp helps to check if all character types in the sub-string of the keyword can be found in the Capsule. Assume the type number of the Capsule is C, LogGrep calculates the six-bit type number $\mathcal{K}$ of the sub-string of the keyword and checks if "$\mathcal{K}\&C = \mathcal{K}$".

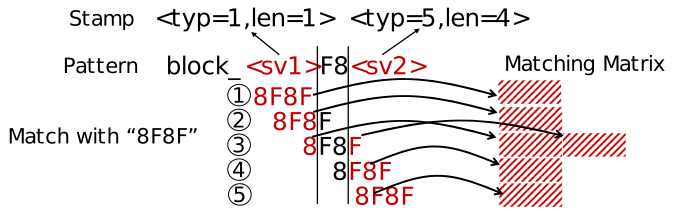


Fig. 7. Runtime pattern matching of a keyword to generate a matching matrix.

If "$\mathcal{K}\&\mathcal{C} \neq \mathcal{K}$", LogGrep will not decompress the Capsule. Otherwise, LogGrep will check whether the length of the sub-string of the keyword exceeds the max-length in the stamp.

For example, the matching case ② in Figure 7 requires "<sv1>" to have "8F8", which violates the max-length of "<sv1>" (len=1). Therefore, Capsule "C1" will not be decompressed for this matching. The matching case ⑤ requires "<sv2>" to have "8F8F", which passes all checks (type number check and max-length check), and thus Capsule "C2" will be decompressed.

Differences for nominal variable vectors. LogGrep first tries to match a keyword on each runtime pattern of the dictionary vector in the same way as it matches a keyword in a real variable vector. Once LogGrep finds the keyword that may match a pattern and this keyword passes the stamp checking, it decompresses the dictionary Capsule to see whether it can find an actual match. LogGrep can jump to the matched runtime pattern directly with the help of count and length in Capsule stamps (Section 5.4). If an actual match is found, LogGrep can know the index number corresponding to the keyword and will search for this index number in the index vector. Otherwise, LogGrep can avoid decompressing and scanning the Capsule holding the index vector.

For example, if LogGrep needs to search the keyword "ERR#404" in Figure 5(b), it first finds this keyword matches the pattern "ERR#<type=1>" and then finds an actual match with index number "0" in the dictionary vector. Finally, it will search "0" in the index vector.

5.4 Matching within Capsules

Once a Capsule is decompressed, LogGrep uses fixed-length matching to accelerate the matching process within it (line 40 in Algorithm 1). During the encapsulation process (Section 4.2), LogGrep pads values of the same Capsule for the sub-variable vector and index vector, and pads values of the same pattern for dictionary vector to the max-length. Furthermore, it records this max-length in the stamp. During the query process, LogGrep will execute fixed-length matching based on the max-length information recorded in stamps.

This fixed-length matching brings three benefits. First, when matching in a Capsule, we can use the fast **Boyer-Moore** (**BM**) algorithm [7] to replace the traditional **Knuth-Morris-Pratt** (**KMP**) algorithm [66]. The reason is that, if values can have a variant length, we have to add some delimiters to separate values. In this case, the BM algorithm may skip characters when it finds a matching string. It is thus unaware of the entry index of the matching string without knowing how many delimiters have been skipped. If each value has a fixed length, LogGrep can compute the entry index by dividing the matching position by the value length.

Second, as discussed in Section 5.2, to mitigate the read amplification during the locating process, LogGrep will check previous hit entries in the following Capsules to incrementally narrow down the hit scope. Instead of scanning all entries in the following Capsules, padding all values to the same size makes such direct checking possible.

Finally, when LogGrep checks a dictionary vector, it first needs to match the keyword to the patterns as discussed in Section 5.3. If a possible match is found, LogGrep can calculate the

Table 1. Description of Log Dataset from Alibaba Cloud

Log type	Log A	Log B	Log C	Log D	Log E	Log F	Log G	Log H	Log I	Log J	Log K
Total Size(GB)	18.67	16.05	0.84	45.82	65.74	34.98	443.3	0.19	0.24	25.81	3.48
Total Line(10^6)	74.74	72.60	5.03	231.43	406.98	77.56	1425.37	1.08	1.27	64.34	24.97
Log type	**Log L**	**Log M**	**Log N**	**Log O**	**Log P**	**Log Q**	**Log R**	**Log S**	**Log T**	**Log U**	
Total Size(GB)	2.03	0.11	7.82	2.21	9.83	1.06	1.04	0.5	964.15	123.22	
Total Line(10^6)	5.84	1.17	33.75	9.48	19.90	4.40	3.26	4.83	6997.09	444.95	

Table 2. Description of Public Log Dataset

Log type	Android	Apache	Bgl	Hadoop	Hdfs	Healthapp	Hpc	Linux
Total Size(GB)	0.179	0.005	0.692	16.035	1.47	0.022	0.031	0.002
Total Line(10^6)	1.555	0.056	4.748	71.512	11.176	0.253	0.433	0.026
Log type	**Mac**	**Openstack**	**Proxifier**	**Spark**	**Ssh**	**Thunderbird**	**Windows**	**Zookeeper**
Total Size(GB)	0.016	0.057	0.002	2.708	0.068	29.605	26.089	0.01
Total Line(10^6)	0.117	0.208	0.021	33.237	0.655	211.212	114.608	0.074

starting position of values of the corresponding pattern, by utilizing the count and length of all prior patterns (i.e., $\sum_i(count_i \times length_i)$), and jump to the starting position directly. Again, such direct locating cannot work if the value length is not fixed.

Our evaluation (Section 6.3) shows that padding can even improve compression ratio to a small degree, due to two reasons. First, after structurization with runtime patterns, according to our observation in Section 2.3, values of the same sub-variable or pattern tend to have a similar length. Therefore, padding does not bring much space overhead. Second, without padding, we still need to add delimiters to separate different values. Therefore, the cost of padding may be smaller than adding a delimiter.

6 EVALUATION

We implement LogGrep with 12,679 lines of C++ code. Our evaluation tries to answer four questions:

- What is the performance of LogGrep in terms of query latency, compression ratio, compression speed, and overall cost on Alibaba Cloud production logs? (Section 6.1)
- How does LogGrep perform on logs beyond Alibaba Cloud logs? (Section 6.2)
- What is the effect of each individual technique incorporated by LogGrep? (Section 6.3)
- What is the effect of different packing methods? (Section 6.4)

To answer these four questions, we measure the performance of LogGrep on 21 production logs (samples of them can be found in [5]) from Alibaba Cloud and 16 public logs [35]. We list the basic information including the total size and lines of production logs from Alibaba Cloud and public logs in Tables 1 and 2, respectively.

Due to privacy limitations, we cannot get the exact query commands used by the Alibaba Cloud engineers. Instead, we synthesize a query command for a typical error type on each type of production log under the guide of our collaborator from Alibaba Cloud. As for public logs, if a log is from an application that can also be found in Alibaba Cloud, we consult Alibaba Cloud developers to synthesize query commands. Otherwise, we use query commands from a previous work [67]. The detailed query commands can be found in Table 3.

As we mentioned in Section 3, LogGrep works in two modes: in *refining mode*, users build the query command gradually in a session; in *direct mode*, users directly launch a complete query command. In the rest of this section, evaluation is conducted in the direct mode unless explicitly stated otherwise.

Table 3. Query Commands Used in Evaluations

LogName	QueryStr
Log A	ERROR and state:REQ_ST_CLOSED and 20012 and reqId:5E9D21AD5E473938
Log B	ERROR and Project:2963 and RequestId:5EA6F82FDF142E2
Log C	ERROR
Log D	project_id:30935 and logstore:res_p and inflow:5
Log E	project:161 and logstore:????_ay87a and shard:99 and wcount:10
Log F	ERROR not UserId:-2
Log G	Operation:ReadChunk and SATADiskId:7 and From:tcp://10.???.??.??:????? and TraceId:3615b60b169820bf160d4acd7b8b8732
Log H	ERROR
Log I	WARNING and 2019-11-06 07
Log J	TraceType:PanguTraceSummary and SectionType:RPC_SealAndNew not CountFail:0
Log K	DELETE and /results/0 and 2019-11-04T02:26
Log L	WARNING and Errorcode:0 and Packet id:172397858
Log M	ERROR and exchange-client-24 and /results/10
Log N	ERROR and project_id:51274
Log O	error and ProjectId:2396 and 2020-04-14 04
Log P	ERROR and CLICK_SAVE_ERROR
Log Q	ERROR and PostLogStoreLogsHandler.cpp and Time:1622009998
Log R	ERROR and part_id:510 and request id REQ_11.???.???.???
Log S	TTY=unknown and /etc/init.d/ilogtaild and Aug 30 10
Log T	ERROR and 39244 and 2020-04-08 05:5
Log U	failed to read trie data and 1618152650857662364_3_149245463_199235229
Android	ERROR and socket read length failure -104
Apache	error and Invalid URI in request
Bgl	ERROR and R00-M1-ND
Hadoop	ERROR and RECEIVED SIGNAL 15: SIGTERM and 2015-09-23
Hdfs	error and blk_8846
Healthapp	Step_ExtSDM and totalAltitude=0
Hpc	unavailable state and HWID=3378
Linux	authentication failure and rhost=221.230.128.214
Mac	failed and Err:-1 Errno:1
Openstack	ERROR or WARNING and Unexpected error while running command
Proxifier	HTTPS and play.google.com:443
Spark	ERROR and Error sending result
Ssh	Received disconnect from and 202.100.179.208
Thunderbird	Doorbell ACK timeout
Windows	Error and Failed to process single phase execution
Zookeeper	ERROR and CommitProcessor

Some characters are changed to "?" due to privacy limitation.

We compute the overall cost of the whole system using the following equation:

$$C_{total} = C_{storage} \times Duration_{storage} \times \frac{Size}{CompressionRatio} + C_{CPU} \times \frac{Size}{CompressionSpeed} + C_{CPU} \times QueryLatency \times QueryFrequency. \quad (1)$$

Based on the data from Alibaba Cloud, we set $C_{storage}$ to be 0.017$ per month, $Duration_{storage}$ to be 6 months [67, 78], and C_{CPU} to be 0.016$ per hour given the type of the CPU we use. The query frequency highly varies: we use a default frequency of 100 in our computation and we discuss how changing the frequency may change the conclusion. Note that $C_{storage}$ already includes the cost of erasure coding to protect logs.

For comparison, we measure the performance of four other systems:

— gzip+grep (ggrep): This is the default choice for near-line logs in Alibaba Cloud. It uses gzip to compress logs. When querying the logs, it decompresses the logs first and then uses grep to search on the decompressed logs. grep supports logical operators with "-E" and "-v" options [17]. It executes the query commands with these operators and UNIX pipe.

— ES [8]: ES is a classical query engine for logs and is the default choice for on-line logs in Alibaba Cloud. We use ES 7.8.0 with the default settings and use python SDE to insert logs

to the index using "bulk" [34] method to accelerate the insertion process. ES supports logical operators and can execute query commands with the help of these operators.
— CLP [81]: CLP is a state-of-the-art method to search directly on compressed logs. We use the source code of CLP with the default settings. CLP cannot support logical operators. After discussing with CLP authors, to execute our query command, we use CLP to execute the first step of each debugging session, and then use grep to query with additional conditions with the help of UNIX pipe. By default, CLP uses zstd [22] as the second-stage compression tool. Compared with LZMA used in LogGrep, this may offer a lower compression ratio but a higher compression and decompression speed.
— LogGrep-SP: For comparison, we also evaluate the performance of LogGrep which only exploits static patterns (i.e., our first attempt in Section 2.2).

Testbed. We perform all experiments on the Linux server with 2× Intel Xeon E5-2682 2.50 GHz CPUs (with 16 cores), 188 GB RAM, and Red Hat 4.8.5 with Linux kernel 3.10.0. Since both compression and query execution can easily be parallelized, we normalize compression time and query latency to using one CPU.

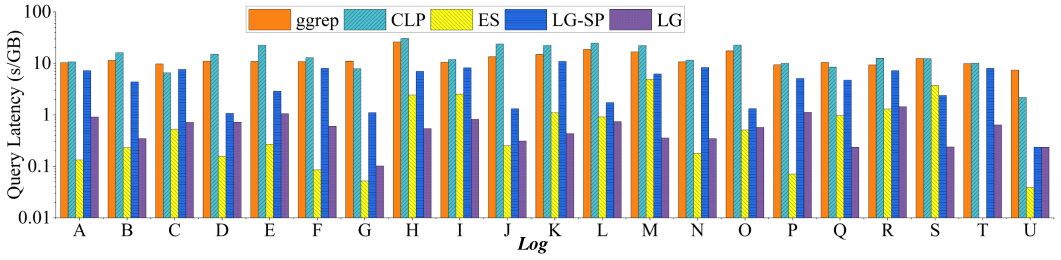
6.1 Performance on Production Logs

We evaluate the query latency, compression ratio, and compression speed on 21 production logs (1.73 TB in total) from Alibaba Cloud. These logs are from different cloud apps and have various characteristics. The total line number of Log T exceeds the max limitation (2,147,483,647) of a single index in ES. As a result, we do not have ES results on Log T.

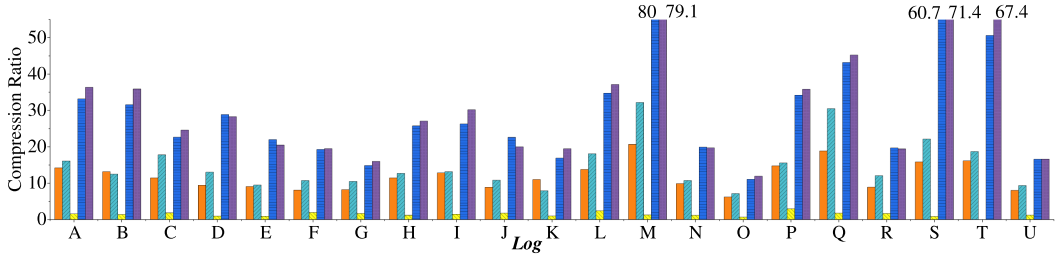
Query latency. As shown in Figure 8(a), the query latency of LogGrep is 6.47× to 108.91× (30.60× on average) lower than ggrep and 8.67× to 77.33× (35.74× on average) lower than CLP. The comparison with ES varies significantly: on 7 logs, the query latency of LogGrep is lower, by up to 15.75×; on 13 logs, the query latency of LogGrep is higher, by up to 15.71×. This is because, on the one hand, ES is highly optimized for query latency and thus performs better on more logs; on the other hand, LogGrep performs better if a query directly hits the template or the keyword is a sub-string of a constant part in the pattern, such that few Capsules need to be decompressed (e.g., Log S, Log M). On average, the query latency of LogGrep is about half of that of ES. The query latency of LogGrep is lower than LogGrep-SP on 20 logs by up to 25.33× (10.07× on average). The only exception is Log U, where the variable vectors that are related to the query have few runtime patterns: in this case, exploiting runtime patterns cannot reduce query latency.

The query latency of LogGrep is longer than one minute only on Log T, since this log is as large as 964 GB. Comparably, ggrep takes longer than one minute to execute a query on 11 logs; CLP takes longer than one minute on 12 logs; LogGrep-SP takes longer than one minute on 7 logs. ES finishes the queries within one minute on all logs it has been tested.

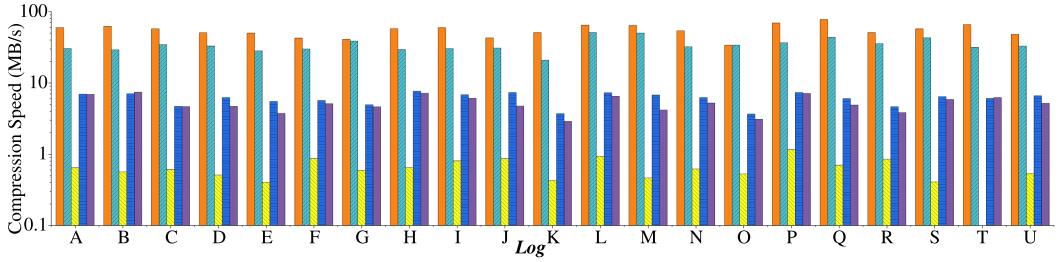
Compression ratio. As shown in Figure 8(b), LogGrep has the highest compression ratio among ggrep, ES, and CLP on all production logs. To be concrete, its compression ratio is 1.77× to 4.50× (2.57× on average) higher than gzip, 1.38× to 3.60 × (2.14× on average) higher than CLP, and 9.53× to 82.51× (23.14× on average) higher than ES. ES needs to build a large index to support low-latency queries, and as a result, its compression ratio is sometimes even smaller than one. The compression ratio of LogGrep-SP and LogGrep are comparable. After exploiting runtime patterns, the compression ratio increases on 15 logs by up to 1.33× and decreases on the other 6 logs by up to 1.13×. This is because, on the one hand, after exploiting runtime patterns, values in a data partition have more similarities; on the other hand, exploiting runtime patterns will introduce more metadata, which incurs more storage overhead.



(a) Query latency on Alibaba Cloud logs (log scale)



(b) Compression ratio on Alibaba Cloud logs



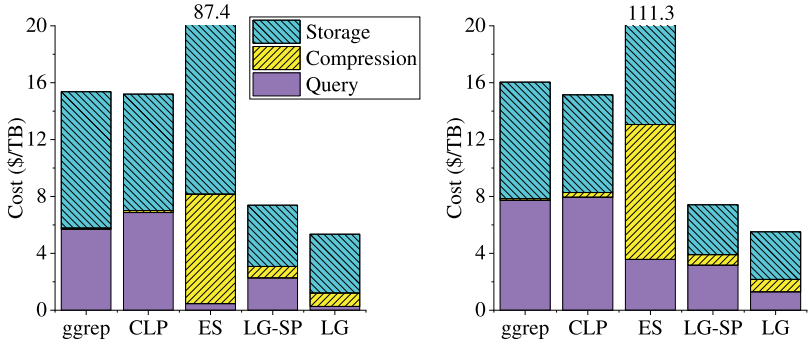
(c) Compression speed on Alibaba Cloud logs (log scale)

Fig. 8. Query latency, compression ratio, and compression speed on 21 Alibaba Cloud logs.

Compression speed. As shown in Figure 8(c), the compression speed of LogGrep is lower than gzip and CLP and higher than ES. Specifically, the compression speed of LogGrep is 0.06× to 0.12× (0.10× on average) as much as that of gzip, 0.08× to 0.26× (0.16× on average) as much as that of CLP, and 4.51× to 14.38× (8.32× on average) higher than that of ES. Here we include the index building time of ES into its compression time. After exploiting runtime patterns, the compression speed of LogGrep is 0.61× to 0.99× on 19 logs (0.86× on average) as much as that of LogGrep-SP. The compression speed of LogGrep is slightly higher than LogGrep-SP by up to 1.05× on 2 logs since LogGrep-SP takes more time to compress coarse storage units.

Such a slowdown is expected since fine-grained structurization certainly needs extra CPU cycles. However, since LogGrep has a lower query latency and high compression ratio, LogGrep can still achieve a lower overall cost.

Overall cost. We calculate the overall cost based on Equation (1). As shown in Figure 9(a), we find LogGrep has the lowest overall cost. Its cost is 34% as much as that of ggrep, 36% as much as that of CLP, and 7% as much as that of ES. By exploiting runtime patterns, the cost of LogGrep is 73% as much as that of LogGrep-SP. As for costs for individual logs, LogGrep has a lower overall cost



(a) Alibaba Cloud logs (b) Public logs

Fig. 9. Overall cost including compression cost, storage cost, and query cost on both Alibaba Cloud and public logs.

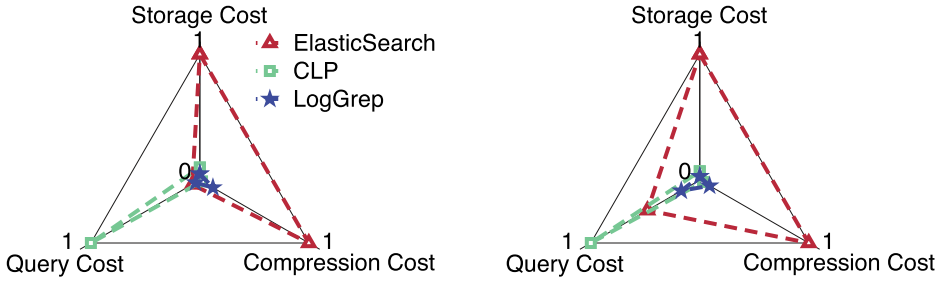
compared with ggrep, CLP, and ES on all tested logs. Its cost is slightly higher than LogGrep-SP on Log D, Log J, and Log U by up to 8% since extracting runtime patterns has a higher compression cost but occasionally does not improve query latency and compression ratio significantly.

If we look at the detailed breakdown, compared with ggrep and CLP, LogGrep pays more cost for compressing logs to reduce storage and query costs; compared with ES, all types of costs of LogGrep are lower. As a result, we conclude that, for near-line cloud logs, trading compression speed for compression ratio and query latency is worthwhile. Besides, although LogGrep pays more cost for compressing logs and more storage cost to store metadata on some logs by exploiting runtime patterns, its overall cost is lower than LogGrep-SP.

Figure 10(a) and (b) compares the individual cost component of LogGrep, ES, and CLP. We calculate the average result of each component on Alibaba Cloud logs and public logs and normalize each cost component to the highest value of such component. We find on both Alibaba Cloud logs and public logs, ES has the highest compression cost and storage cost while CLP has the highest query cost. On Alibaba Cloud, the storage cost and query cost of LogGrep are only 5% and 4% of the highest cost, respectively, while on public logs these results are 3% and 17%, respectively. CLP has the lowest compression cost, its compression cost is only 2% and 3% of the highest cost on Alibaba Cloud logs and public logs, respectively.

In respect to the effect of different parameter settings, we find the CPU costs per hour (C_{CPU}) will affect the sum of query cost and compression cost (we call this sum as the computation cost). The storage duration ($Duration_{storage}$) and the storage price ($C_{storage}$) will affect the storage cost. Compared with other methods, LogGrep has the lowest computation cost and storage cost on all tested logs. As a result, no matter how we change these three parameter settings, LogGrep will still outperform other systems in respect of the overall cost. On the other hand, the query cost of LogGrep is higher than ES on some logs and the parameter of query frequency ($QueryFrequency$) only affects this value. As a result, if we have a higher query frequency, the overall cost of ES may be lower than LogGrep. For the 13 logs on which LogGrep's query latency is higher than ES, our computation shows, if the query frequency is over 7,447 to 542,194 times (130,169 times on average), ES will have a lower overall cost: such frequency is much higher than the common use cases for near-line cloud logs.

Response time during interactive debugging. When an error occurs, engineers usually query in refining mode, namely, build the query command gradually like the usage example in Section 2.4. The total and maximum response time of such mode are all critical to user experience. To measure



(a) Alibaba Cloud logs (b) Public logs

Fig. 10. Individual cost components on Alibaba Cloud and public logs.

them, we split the query commands by logical operators (and, or, not) into parts and test each query command in multiple steps, in each step we add a new part to the command. We show the total response time of the whole session in Figure 11(a) and the response time of the slowest step within the session in Figure 11(b).

According to the query, the total response time of LogGrep in each session is lower than that of ggrep and CLP on all 21 Alibaba Cloud production logs. To be concrete, the total response time of LogGrep is 8.33× to 177.01× (27.66× on average) lower than ggrep, 8.29× to 216.41× (60.05×) lower than CLP. The comparison with ES varies significantly: on 7 logs, the total response time of LogGrep is lower, by up to 11.81 ×; on 13 logs, the total response time of LogGrep is higher, by up to 18.57×. On average, the total response time of LogGrep is about half of that of ES.

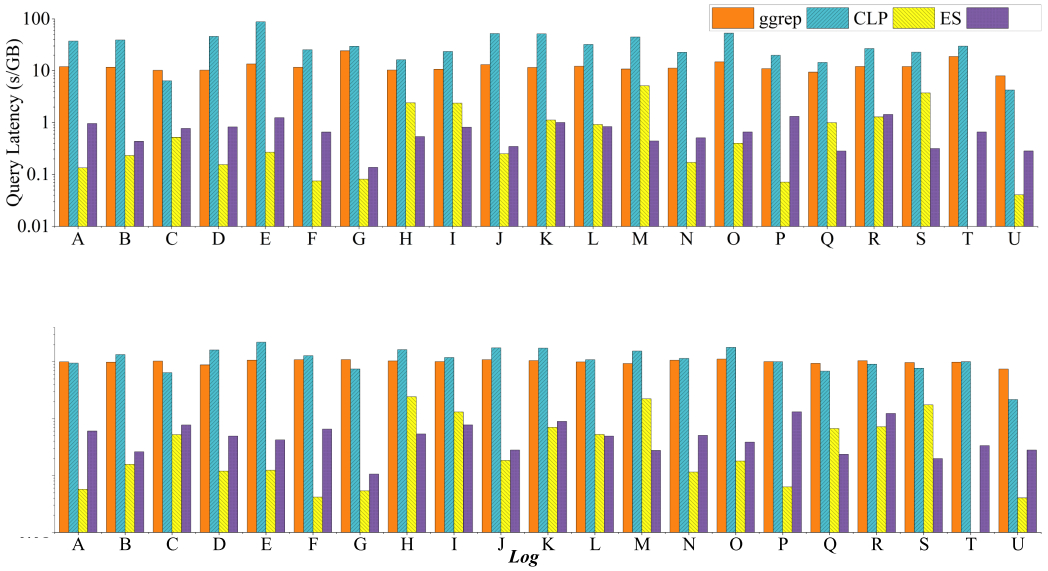
As for the maximum response time, LogGrep has a lower response time than ggrep and CLP on all production logs. To be concrete, the max response time of LogGrep is 7.61× to 102.89× (27.44× on average) lower than ggrep, 7.37× to 70.35× (30.65× on average) lower than CLP. As for ES, the maximum response time of LogGrep is lower than ES on 6 logs by up to 8.80× and higher on 14 logs by up to 20.90×. On average, the maximum response time of LogGrep is about 60% of that of ES. We find the maximum response time of LogGrep is within 1 minute on 20 production logs except Log T while the maximum response time of CLP is larger than 1 minute on 12 production logs, which is deemed as unacceptable.

Besides, we find the total response time of LogGrep in the refining mode is close to the query latency of executing the last command of the session in the direct mode (only slower by 3% on Log E and Log T). This is because LogGrep can incrementally search on previous hit position with the help of Query Cache and Indexed Bitmap.

6.2 Performance on Public Logs

We also evaluate query latency, compression ratio, and compression speed on 16 public logs benchmark [35, 40] (77 GB in total), which are from various scopes of applications including HPC, personal digital devices of different platforms (Linux, Windows, and Mac), and Web servers. Since CLP does not support all logical operations, it fails to work on "Openstack", and thus we do not list this result.

Query latency. The query latency of LogGrep is 2.27× to 51.25× (14.56× on average) lower than that of ggrep, and 1.94× to 42.00× (13.74× on average) lower than that of CLP. Again, the comparison with ES varies significantly: on 11 logs, the query latency of LogGrep is lower, by up to 12×; on 5 logs, the query latency of LogGrep is higher, by up to 12.23×. On average, the query latency of LogGrep is about 33% of that of ES. The query latency of LogGrep is lower than LogGrep-SP on public logs by up to 25.00 × (7.02 × on average).



(b) Maximum response time of each session (log scale)

Fig. 11. Response time of each session in the refining mode on Alibaba Cloud logs.

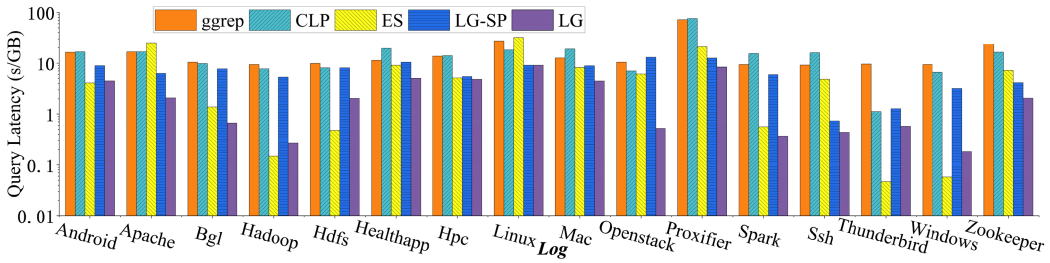
Compression ratio. LogGrep has the highest compression ratio among ggrep, ES, and CLP on all public logs. LogGrep's compression ratio is 1.34× to 26.55× (3.99× on average) higher than that of gzip, 1.03 to 3.12× (2.10× on average) higher than that of CLP and 10.23× to 182.65× (41.44× on average) higher than that of ES. After exploiting runtime patterns, the compression ratio of LogGrep increases on 12 logs by up to 1.33×, and decreases on the other 4 logs by up to 1.20×.

Compression speed. LogGrep's compression speed is slower than gzip (0.07× to 0.28×, 0.14× on average) and CLP (0.08× to 0.99×, 0.35× on average), but is still higher than ES (5.84× to 16.12×, 11.15× on average). After exploiting runtime patterns, the compression speed of LogGrep is 0.61× to 0.99× (0.86× on average) as much as that of LogGrep-SP on 14 logs and is slightly higher than that of LogGrep-SP on 2 logs by up to 1.07×.

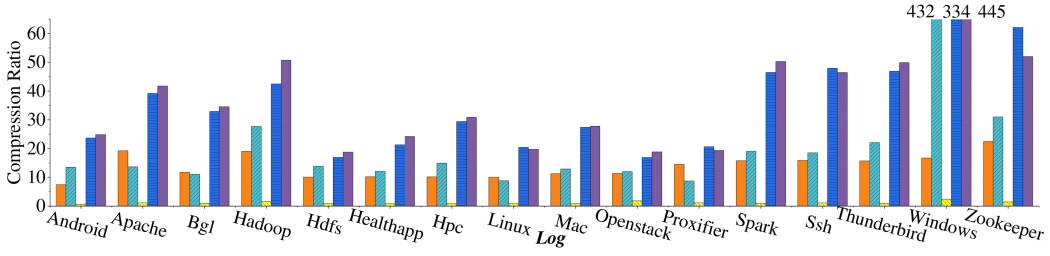
Overall cost. As shown in Figure 9(b), LogGrep's overall cost is 34% as much as that of ggrep, 41% as much as that of CLP, 5% as much as that of ES, and 74% as much as that of LogGrep-SP. The detailed breakdown shows a trend similar to that on Alibaba Cloud logs. For ES to have a lower overall cost, the system needs to query at least 17,718 to 125,466 times (73,019 on average), which is not likely to happen on near-line logs.

6.3 Effects of Individual Techniques

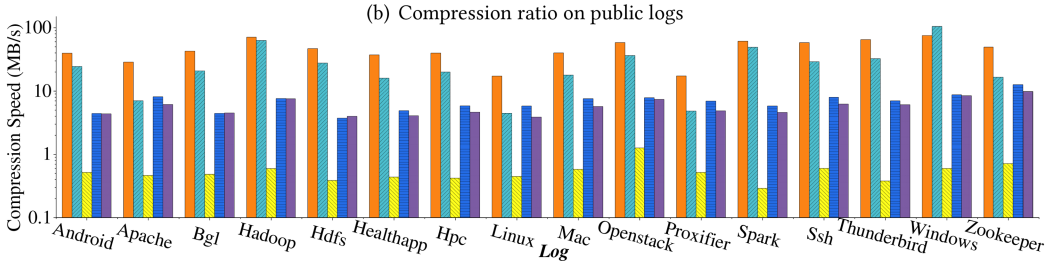
This section measures the effects of individual techniques proposed by this article. We first evaluate the effect of techniques on the query latency, which includes runtime pattern extraction (Sections 4.1 and 5.3), Capsule stamping (Section 4.3), fixed-length query within the Capsule (Section 5.4), query cache as well as the Indexed Bitmap (Section 5.2). Then we evaluate the effects of these techniques on the compression ratio. We mainly evaluate the effect of fixed-length padding since other techniques except fixed-length padding and runtime pattern extraction have no effect on the compression ratio; the effect of runtime pattern extraction can be shown by the compression ratio of LogGrep-SP in Figure 8(b). Besides, we also evaluate the effect of partial reconstruction on the read amplification per class of workloads.



(a) Query latency on public logs (log scale)



(b) Compression ratio on public logs



(c) Compression speed on public logs (log scale)

Fig. 12. Query latency, compression ratio and compression speed on 16 public logs.

Effect on the query latency. We implement six versions of LogGrep, each removes a technique: "w/o real" removes runtime pattern extraction and fine-grained structurization in real variable vectors; "w/o nomi" does similar for nominal variable vectors; "w/o stamp" removes the stamp of each Capsule and thus does not filter Capsules with their stamps during keyword matching; "w/o fixed" removes the padding process and queries on variant-length "Capsules" with the KMP algorithm ; "w/o cache" removes the Query Cache and re-executes prior queries; "w/o IndexedBitmap" removes the design of indexed bitmap and uses an ordinary bitmap structure to record the temporary results instead. Since Query Cache is especially useful for the refining mode, we compare the full version with "w/o cache" version in the refining mode. We conduct other performance comparisons in the direct mode.

Figure 13 shows the query latency of the first five versions on production logs after normalized to the full version. Generally speaking, runtime pattern extraction and structurization for real variable vectors and nominal variable vectors can achieve 1.51× and 4.03× reduction to query latency respectively; Capsule stamp can achieve 3.59× reduction to query latency; fixed-length matching can achieve 1.89× reduction to query latency; Query Cache can achieve 2.08× reduction to query latency.

Among them, runtime pattern extraction and encapsulation in real variable vectors accelerate queries by 1.13× to 3.61×, it has the most significant improvement on Log G, which has many

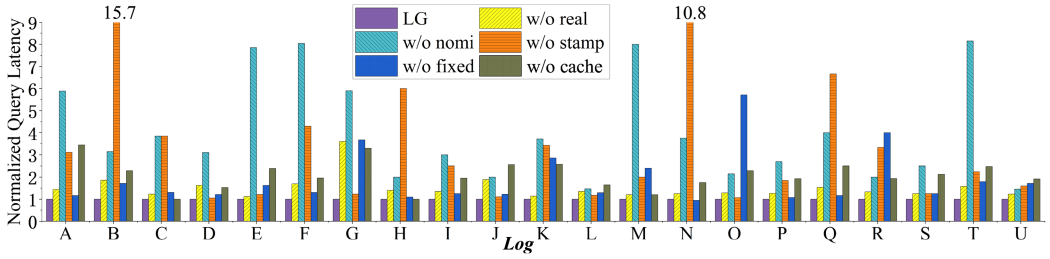


Fig. 13. The effects of individual techniques on production logs.

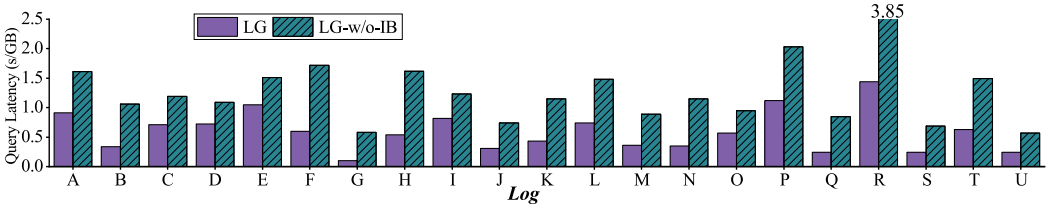


Fig. 14. The effect of Indexed Bitmap on production logs.

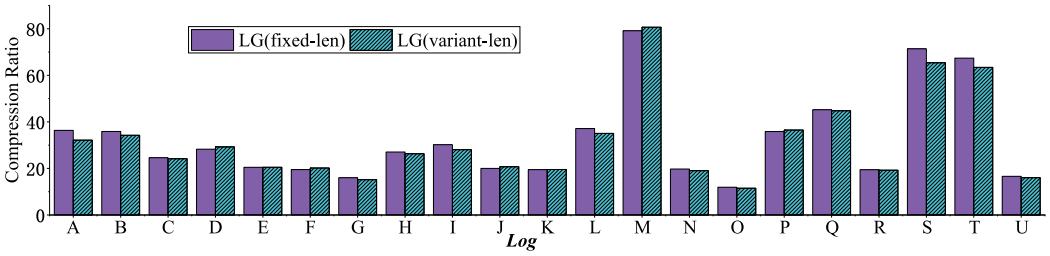


Fig. 15. The effect of fixed length encoding to the compression ratio on production logs.

real variable vectors with specific formats, such as block number and IP address. Runtime pattern extraction and encapsulation in nominal variable vectors bring the most significant reduction to query latency by 1.46× to 8.16×, it can reduce query latency by over 8× on Log F, Log M, and Log T, where nominal variables dominate the storage space. The effect of Capsule stamping varies significantly, it can accelerate queries by 1.05× to 15.71×. Its result depends on how strict the stamp is, values in Capsules of Log B tend to have less character type and a smaller length scope and thus have a stricter stamp. Fixed-length query accelerates queries on 20 logs by up to 5.71×, and has a longer latency by 5% on Log N, since values in Log N have a larger length scope, compared with query on varied-length Capsules, LogGrep takes more time to decompress the Capsules. The effect of the query cache technique depends on how many queries a session contains in refining mode, it can accelerate queries more significantly when the final query command has many search strings. The effect on the query latency of the indexed bitmap design can be found in Figure 14. Based on the figure, we find the design of Indexed Bitmap can mitigate the query latency by 1.43× to 5.73× (2.52× on average).

Effect on the compression ratio. We evaluate the effect of fixed-length padding on compression ratio. The results can be found in Figure 15. We find the compression ratio with padding is 0.99× to 1.10× (1.04× on average) as much as that with no padding. This means on most logs, the overhead of padding is smaller than the overhead to add a delimiter to separate different values.

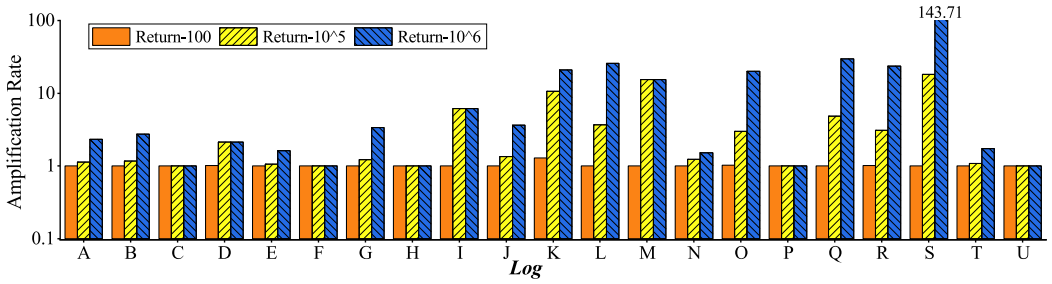


Fig. 16. The effect of partial reconstruction to the amplification rate on production logs (log scale).

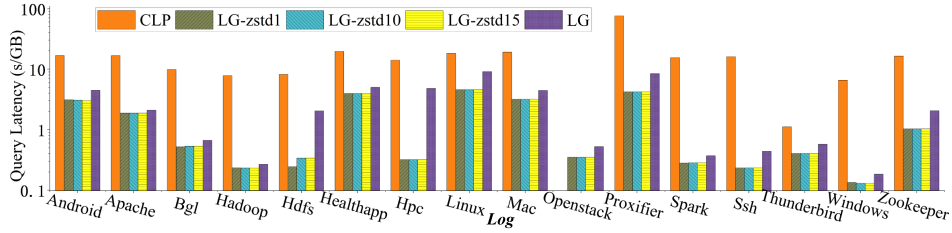
Effect on the read amplification. We evaluate the effect of partial reconstruction on the read amplification during the reconstruction process. The read amplification rate in the reconstruction process is measured by dividing the read size of the reconstruction process by the necessary read size to locate all hit entries. We measure this rate when reconstructing different numbers of entries on each dataset. The result can be found in Figure 16. Given different hit numbers on each dataset, the amplification rate can vary from 1×, namely there is almost no read amplification since all hit entries are within one block, to 18.14× when reconstructing 10^5 entries. Based on our observation of the interactive debugging process, we only need to locate all hit entries and reconstruct a small number of hit entries in practice. As a result, we can reconstruct a small part of hit entries (such as 100), whose amplification rate is only up to 1.29× according to the evaluation.

6.4 Effects of Different Packing Methods

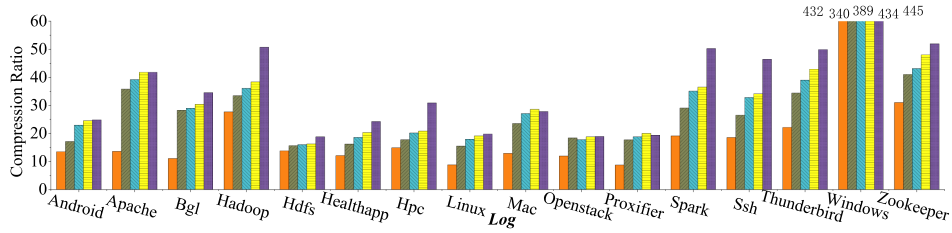
One concern is that it is unfair to compare the performance between LogGrep and CLP with different packing methods, i.e., using LZMA and zstd [22], respectively, for LogGrep and CLP. To address this concern, we change the packing method of LogGrep and evaluate the query latency, compression ratio, compression speed, and overall cost, respectively. We use LZMA [88] as the default packing method of LogGrep and test zstd [22] of three different levels (i.e., LogGrep-zstd1, LogGrep-zstd10, and LogGrep-zstd15) for comparison. A higher level means the packing method has a higher compression ratio but a lower compression and decompression speed.

Query latency. Figure 17(a) shows a comparison on the query latency. We find changing the packing method to zstd leads to a lower query latency. For example, the query latency of LogGrep-zstd10 is lower than LogGrep by 1.11× to 15× (2.63× on average). The query latency drops significantly on Hpc since it costs a large amount of time to decompress during the query; compared with LogGrep, LogGrep-zstd10 has a much lower decompression latency. The query latency of LogGrep-zstd1, LogGrep-zstd10, LogGrep-zstd15 are close to each other except for Hdfs, where LogGrep-zstd1 has a lower query latency by 1.39×.

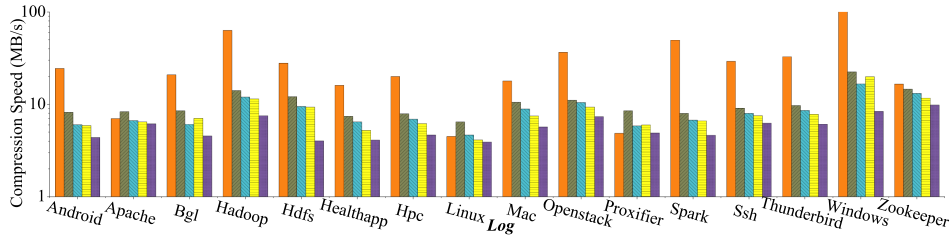
Compression ratio. Figure 17(b) shows a comparison on the compression ratio. One can see that when changing the packing method to zstd, the compression ratio drops on most logs but is still higher than CLP. The compression ratio of LogGrep is higher than LogGrep-zstd10 by 1.02× to 1.53× (1.21× on average). The compression ratio of LogGrep-zstd10 is still higher than CLP on 15 logs by 1.16× to 2.88× and is only lower than CLP on Windows logs by 10%. The compression ratio of LogGrep-zstd1, LogGrep-zstd10, LogGrep-zstd15 is getting higher with the growth of the compression level, where the compression ratio of LogGrep-zstd15 is higher than that of LogGrep-zstd1 by 1.02× to 1.43× (1.19 × on average).



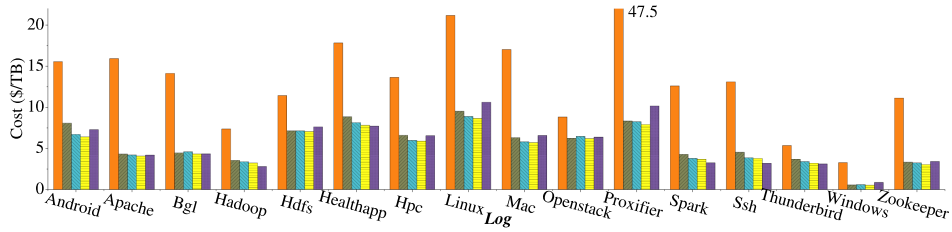
(a) Query latency (log scale)



(b) Compression ratio



(c) Compression speed (log scale)



(d) Overall cost

Fig. 17. Query latency, compression ratio, compression speed, and overall cost of different packing methods.

Compression speed. Figure 17(c) shows a comparison on the compression speed. We find changing the packing method to zstd leads to a higher compression speed than LogGrep, but such a speed is still lower than CLP on most logs. For example, the compression speed of LogGrep-zstd10 is higher than LogGrep by 1.08× to 2.37× (1.48× on average). Compared with CLP, the compression speed of LogGrep-zstd10 is lower by 1.04× to 7.34× on 14 logs, but is higher on Linux and Proxifier by up to 20%. As the compression level grows from LogGrep-zstd1, LogGrep-zstd10, to LogGrep-zstd15, the compression speed becomes increasingly lower. The compression speed of LogGrep-zstd15 is higher than that of LogGrep-zstd1 by 1.13× to 1.57× (1.30× on average).

Overall cost. Figure 17(d) shows a comparison on the overall cost. We calculate the overall costs of different methods with the assumption the storage duration is 6 months and the query count within this duration is 100 times. We find LogGrep-zstd15 has the lowest overall cost on 11 logs, and LogGrep has the lowest overall cost on another 5 logs (i.e., Hadoop, Healthapp, Spark, Ssh, and Thunderbird). We find the storage cost of these 5 logs takes a larger part in the overall cost than other logs; since LogGrep has a higher compression ratio, its overall cost stands lower. On other logs, LogGrep-zstd15 has a lower compression and query cost and thus a lower overall cost.

7 RELATED WORK

Previous log management tools mainly target either a high compression ratio or a low query latency. Some newly proposed methods are trying to achieve both by adopting the idea of filtering and only decompressing relevant logs. LogGrep follows the idea of "vertical partitioning" and proposes to exploit both static and runtime patterns to partition data such that the content in each partition shares common features. To achieve this, LogGrep extracts runtime patterns automatically and structurizes log data in fine-grained units, which can reduce query latency significantly while maintaining a low storage cost.

Log compression. Log compression can be categorized into general-purpose compression methods and log-specific compression methods. General-purpose methods include light-weight encoding methods and LZ77-based compression methods. Light-weight encoding methods include dictionary encoding [64], RLE encoding [69], and delta encoding. Such methods can achieve a high compression speed but a low compression ratio. LZ77-based compression methods [89] first remove the duplicated parts and then encode the left parts with a heavy method such as Huffman encoding [41], arithmetic encoding [79], as well as asymmetric arithmetic encoding [20]. These methods can achieve a high compression ratio but their compression speed is unsatisfied. Linux gzip [18], LZMA [88] and zstd [22] can all be categorized into this type.

The log-specific compression methods can be categorized into two groups: bucket-based and parser-based. Bucket-based methods (e.g., LogArchive [36], Cowic [53], and MLC [25]) first group logs into different buckets according to their similarity, and compress each bucket individually. Parser-based methods (e.g., LogReducer [78], Logzip [37], Drain [39], LogMine [38], SHISO [58], LogSig [74], SLCT [75], and LFA [60]) parse log into constant format (a.k.a, templates) and variables and then compress them separately using general-purpose compression methods.

These methods usually have a high compression ratio, but to execute a query, one needs to decompress data first, which can cause a high latency. LogGrep inherits the parser from the parser-based methods and further implements query on compressed logs.

Log query. Some general-purpose text query tools, like ES [8] and Splunk [42], can be used to query log files. They build an inverted index for tokens in the original text and can achieve low query latency. However, compared with LogGrep, their compression ratio and speed are unsatisfactory for near-line logs. Besides, the standard query tool in Linux, namely grep, is often used to query uncompressed text.

Several log management companies have designed log-specific query tools, such as Scalyr [43] and Loki [48], which build indexes for different log fields. However, they don't take compression into consideration.

Query on compressed data. These works can be categorized into three types based on their generality. First, some approaches [46] can query encoded data directly without any decompression such as queries on LZW-encoded [90], LZ78-encoded [71], and light-weight-encoded data. These approaches are highly dependent on specific compression methods, and cannot utilize methods with a higher compression ratio (e.g., LZMA).

Second, some approaches can query compressed text data directly. They can be roughly divided into two types: index-based and DAG-based. Index-based methods can query on compressed suffix trees [33, 47, 68], tries [27], and inverted indices [64, 65], but they only reduce the size of index instead of original data. DAG-based methods, e.g., TADOC [84], Succict [3], and Comopressed-Data [83] can query directly on compressed text, but since these methods need to keep the interpretability of encoded data, their overall storage cost is unsatisfactory [67].

Third, the idea of partitioning data into independent units and filtering irrelevant units has been proposed by database community [1, 2, 15, 49] and has been applied by CLP, the state-of-the-art log compression and query tool. However, according to our experiments (Section 6.1), this filtering granularity is still too coarse and as a result, both the query latency and the overall cost of CLP are unsatisfactory.

Pattern extraction. General pattern extraction methods on text data are well-studied [4, 31, 59, 62, 76, 86], but their time complexity is relatively high and thus they are slow on large-scale log data [87]. Previous log pattern extraction on logs mainly focuses on extracting static patterns automatically by source/binary code analysis [9, 12, 80, 85] or applying heuristic methods [38, 45, 56, 58, 60, 72, 74, 75].

There are also some sub-pattern extraction methods commonly used in column-family databases [2, 10, 29]. This method can be categorized into two types: pruning techniques and heuristics-based techniques. PADS [28] and Datamaran [30] are two classical pruning techniques, which score all possible splitting paths for string sets and prune those states with low scores. PIDS [44] is a typical heuristic-based technique, which splits nodes based on specific characters, character types, and longest common strings. Our runtime pattern extraction process can be categorized into a heuristic-based technique inspired by PIDS and to the best of our knowledge, it is the first work to extract runtime patterns automatically to partition data into fine-grained units with common features.

8 CONCLUSION AND FUTURE WORK

This article proposes a new near-line log compression and query method called LogGrep. LogGrep exploits both static and runtime patterns to partition logs vertically into fine-grained units called Capsules during the compression process and executes pattern matching and data filtering to avoid decompressing irrelevant Capsules during the query process. Our evaluation shows this design enables effective summaries and can accelerate queries, mitigate read-amplification during interactive debugging, and save overall cost significantly. Our measurement also indicates that using different packing methods in LogGrep has an obvious impact on compression ratio, compression speed, and query latency.

Our profiling shows that there is room to improve the compression speed, which is important to ingesting raw logs at a high speed. In the future, we will continue optimizing our implementation and scale LogGrep to a distributed cluster.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. 671–682.

[3] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *NSDI'15 Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. 337–350.

[4] Deepak Agnihotri, Kesari Verma, and Priyanka Tripathi. 2014. Pattern and cluster mining on text data. In *Proceedings of the 2014 4th International Conference on Communication Systems and Network Technologies*. IEEE, 428–432.

[5] LogGrep authors. 2022. Producition logs sample. Retrieved from https://github.com/THUBear-wjy/openSample

[6] LogGrep authors. 2023. Source code of LogGrep. Retrieved from https://github.com/THUBear-wjy/LogGrep
[7] R. Boyer and J. S. Moore. 1977. A fast string searching algorithm. *Communications of the ACM* 20 (1977), 762–772.
[8] Elasticsearch B. V. 2020. Elasticsearch 7.8.0. Retrieved from https://www.elastic.co/downloads/past-releases/elasticsearch-7-8-0
[9] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. 2021. LogStore: A cloud-native and multi-tenant log database. In *Proceedings of the 2021 International Conference on Management of Data*. 2464–2476.
[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2 (2008), 1–26.
[11] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 217–231.
[12] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. 2005. String analysis for x86 binaries. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 88–95.
[13] John Cleary and Ian Witten. 1984. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications* 32, 4 (1984), 396–402.
[14] Doug Cutting and Jan Pedersen. 1989. Optimization for dynamic inverted index maintenance. In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 405–411.
[15] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.
[16] Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. Retrieved from https://tools.ietf.org/html/rfc1951
[17] Linux developer community. 2020. Grep Manuel. Retrieved from https://linux.die.net/man/1/grep
[18] GNU developer group. 2019. Homepage and documentation of Tar. Retrieved from https://www.gnu.org/software/tar/
[19] 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*. ACM, 1285–1298.
[20] Jarek Duda. 2013. Asymmetric numeral systems as close to capacity low state entropy coders. arXiv:1311.2540. Retrieved from https://arxiv.org/abs/1311.2540
[21] Susan Dumais, Robin Jeffries, Daniel M. Russell, Diane Tang, and Jaime Teevan. 2014. Understanding user behavior through log data and analysis. In *Proceedings of the Ways of Knowing in HCI*. Springer, 349–372.
[22] Facebook. 2021. z-standard compression tool. Retrieved from https://github.com/facebook/zstd
[23] Yaochung Fan, Yuchi Chen, Kuanchieh Tung, Kuochen Wu, and Arbee L. P. Chen. 2016. A framework for enabling user preference profiling through Wi-Fi logs. *IEEE Transactions on Knowledge and Data Engineering* 28, 3 (2016), 592–603.
[24] Bettina Fazzinga, Sergio Flesca, Filippo Furfaro, and Luigi Pontieri. 2018. Online and offline classification of traces of event logs on the basis of security risks. *Journal of Intelligent Information Systems* 50, 1 (2018), 195–230.
[25] Bo Feng, Chentao Wu, and Jie Li. 2016. MLC: An efficient multi-level log compression method for cloud backup systems. In *Proceedings of the 2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 1358–1365.
[26] Sérgio Fernandes and Jorge Bernardino. 2015. What is bigquery?. In *Proceedings of the 19th International Database Engineering and Applications Symposium*. 202–203.
[27] Paolo Ferragina and Giovanni Manzini. 2001. An experimental study of a compressed index. *Information Sciences* 135, 1 (2001), 13–28.
[28] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. 2008. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 421–434.
[29] Yannis Foufoulas, Lefteris Sidirourgos, Eleftherios Stamatogiannakis, and Yannis Ioannidis. 2021. Adaptive compression for fast scans on string columns. In *Proceedings of the 2021 International Conference on Management of Data*. 554–562.
[30] Yihan Gao, Silu Huang, and Aditya Parameswaran. 2018. Navigating the data lake with DATAMARAN: Automatically extracting structure from log datasets. In *Proceedings of the 2018 International Conference on Management of Data*. 943–958.
[31] Yang Gao, Yue Xu, and Yuefeng Li. 2014. Pattern-based topics for document modelling in information filtering. *IEEE Transactions on Knowledge and Data Engineering* 27, 6 (2014), 1629–1642.
[32] Mona Ghassemian, Philipp Hofmann, Christian Prehofer, Vasilis Friderikos, and Hamid Aghvami. 2004. Performance analysis of internet gateway discovery protocols in ad hoc networks. In *Proceedings of the 2004 IEEE Wireless Communications and Networking Conference*. IEEE, 120–125.

[33] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing* 35, 2 (2005), 378–407.
[34] ElasticSearch group. 2019. Elasticsearch: Bulk Inserting Examples. Retrieved from https://queirozf.com/entries/elasticsearch-bulk-inserting-examples
[35] Loghub group. 2019. Download link of public log dataset. Retrieved from https://zenodo.org/record/7056802#.Yxm2VexBwq2
[36] LogArchive group. 2019. Open source code of LogArchive. Retrieved from https://github.com/robertchristensen/log_archive_v0
[37] Logzip group. 2019. Open source code of Logzip. Retrieved from https://github.com/logpai/logzip
[38] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 1573–1582.
[39] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *Proceedings of the 2017 IEEE International Conference on Web Services*. IEEE, 33–40.
[40] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2020. Loghub: A large collection of system log datasets towards automated log analytics. arXiv:2008.06448. Retrieved from https://arxiv.org/abs/2008.06448
[41] David A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. DOI: https://doi.org/10.1109/JRPROC.1952.273898
[42] Splunk Inc. 2020. Spunk Enterprise 8.0.3. Retrieved from https://www.splunk.com/en_us/download/previous-releases.html
[43] Scalyr Inc. 2021. Scalyr home page. Retrieved from https://www.scalyr.com/
[44] Hao Jiang, Chunwei Liu, Qi Jin, John Paparrizos, and Aaron J. Elmore. 2020. PIDS: Attribute decomposition for improved compression and query performance in columnar storage. *Proceedings of the VLDB Endowment* 13, 6 (2020), 925–938.
[45] 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 8th International Conference on Quality Software*. IEEE, 181–186.
[46] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. 1998. Multiple pattern matching in LZW compressed text. In *Proceedings of the DCC '98 Data Compression Conference (Cat. No.98TB100225)*. 103–112.
[47] Stefan Kurtz. 1999. Reducing the space requirement of suffix trees. *Software - Practice and Experience* 29, 13 (1999), 1149–1171.
[48] Grafana Labs. 2021. Loki Documentation. Retrieved from https:grafana.com/docs/loki/latest/
[49] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*. 311–326.
[50] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. 2012. The unified logging infrastructure for data analytics at Twitter. arXiv:1208.4171. Retrieved from https://arxiv.org/abs/1208.4171
[51] Rundong Li, Pinghui Wang, Jiongli Zhu, Junzhou Zhao, Jia Di, Xiaofei Yang, and Kai Ye. 2021. Building fast and compact sketches for approximately multi-set multi-membership querying. In *Proceedings of the 2021 International Conference on Management of Data*. 1077–1089.
[52] Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. 2021. Combining aggregation and sampling (nearly) optimally for approximate query processing. In *Proceedings of the 2021 International Conference on Management of Data*. 1129–1141.
[53] Hao Lin, Jingyu Zhou, Bin Yao, Minyi Guo, and Jie Li. 2015. Cowic: A column-wise independent compression for log stream analysis. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 21–30.
[54] 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 *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 863–873.
[55] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. 2011. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering* 24, 11 (2011), 1921–1936.
[56] Adetokunbo A. O. Makanju, A. 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*. 1255–1264.
[57] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel 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*. ACM, 167–177.

[58] Masayoshi Mizutani. 2013. Incremental mining of system log format. In *Proceedings of the 2013 IEEE International Conference on Services Computing*. IEEE, 595–602.
[59] Fionn Murtagh and Pedro Contreras. 2012. Algorithms for hierarchical clustering: An overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2, 1 (2012), 86–97.
[60] Meiyappan Nagappan and Mladen A. Vouk. 2010. Abstracting log lines to log event types for mining software system logs. In *Proceedings of the 7th Working Conference on Mining Software Repositories*. IEEE, 114–117.
[61] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 26–26.
[62] Frank Nielsen. 2016. Hierarchical clustering. In *Proceedings of the Introduction to HPC with MPI for Data Science*. Springer, 195–211.
[63] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H. Chin, and Sumayah Alrwais. 2015. Detection of early-stage enterprise infection by mining large-scale log data. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 45–56.
[64] Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. 2019. Fast dictionary-based compression for inverted indexes. In *Proceedings of the 12th ACM International Conference on Web Search and Data Mining*. 6–14.
[65] Giulio Ermanno Pibiri and Rossano Venturini. 2020. Techniques for inverted index compression. *Comput. Surveys* 53, 6 (2020), 1–36.
[66] Mireille Régnier. 1989. Knuth-morris-pratt algorithm: An analysis. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*. Springer, 431–444.
[67] Kirk Rodrigues, Yu Luo, and Ding Yuan. 2021. CLP: Efficient and scalable search on compressed text logs. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. 183–198.
[68] Kunihiko Sadakane. 2002. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. 225–232.
[69] D. Salomon. 2004. *Data Compression: The Complete Reference*. Springer, New York.
[70] Vijay Samuel. 2018. Monitoring anything and everything with beats at eBay. (2018).
[71] Khalid Sayood. 2017. *Introduction to Data Compression*. Morgan Kaufmann.
[72] Keiichi Shima. 2016. Length matters: Clustering system log messages using length of words. arXiv:1611.03213. Retrieved from https://arxiv.org/abs/1611.03213
[73] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented partitioning for columnar layouts. *Proceedings of the VLDB Endowment* 10, 4 (2016), 421–432.
[74] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. ACM, 785–794.
[75] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations and Management*. IEEE, 119–126.
[76] Paola Velardi, Giovanni Stilo, Alberto E. Tozzi, and Francesco Gesualdo. 2014. Twitter mining for fine-grained syndromic surveillance. *Artificial Intelligence in Medicine* 61, 3 (2014), 153–163.
[77] 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 18th European Conference on Computer Systems (Eurosys'23)*.
[78] 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 *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*. 249–262.
[79] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic coding for data compression. *Communications of the ACM* 30, 6 (1987), 520–540.
[80] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 117–132.
[81] YScope. 2021. clp-core. Retrieved from https://github.com/y-scope/clp-core
[82] 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*. ACM, 143–154.
[83] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. 2022. CompressDB: Enabling efficient compressed data direct processing for various databases. In *Proceedings of the 2022 International Conference on Management of Data*. 1655–1669.
[84] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.

[85] Maosheng Zhang, Ying Zhao, and Zengmingyu He. 2017. Genlog: Accurate log template discovery for stripped x86 binaries. In *Proceedings of the 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 337–346.

[86] Ning Zhong, Yuefeng Li, and Sheng-Tang Wu. 2010. Effective pattern discovery for text mining. *IEEE Transactions on Knowledge and Data Engineering* 24, 1 (2010), 30–44.

[87] 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*. IEEE, 121–130.

[88] 7 zip developer group. 2019. 7-zip file achiever home page. Retrieved from https://www.7-zip.org/

[89] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. DOI: https://doi.org/10.1109/TIT.1977.1055714

[90] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.

Received 3 January 2023; revised 5 November 2023; accepted 26 December 2023