LogGrep: Fast and Cheap Cloud Log Storage by Exploiting both Static and Runtime Patterns

Junyu Wei
Tsinghua University

Guangyan Zhang*
Tsinghua University

Junchao Chen
Tsinghua University

Yang Wang
The Ohio State University

Weimin Zheng
Tsinghua University

Tingtao Sun
Alibaba Group

Jiesheng Wu
Alibaba Group

Jiangwei Jiang
Alibaba Group

Abstract

In cloud systems, near-line logs are mainly used for debugging, which means they prefer a low query latency for a better user experience, and like any other logs, they also prefer a low overall cost including storage cost to store compressed logs and computation cost to compress logs and execute queries.

This paper proposes LogGrep, the first log compression and query tool that structurizes and organizes log data properly in fine-grained units by exploiting both static and runtime patterns. It first parses logs into variable vectors by exploiting static patterns and then extracts runtime pattern(s) automatically within each variable vector with a novel extraction method. Based on these 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 possibly match the keywords, with the help of the extracted runtime patterns and the Capsule stamps.

We evaluate LogGrep on 21 types of logs from the production environment of Alibaba Cloud, and 16 types of logs from the public datasets. The results show that LogGrep can reduce query latency and overall cost by an order of magnitude compared to state-of-the-art works. Such results have confirmed that exploiting both static and runtime patterns to structurize logs can achieve fast and cheap cloud log storage.

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

Keywords: 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. 2023. LogGrep: Fast and Cheap Cloud Log Storage by Exploiting both Static and Runtime Patterns. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, ROME, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3567484

1 Introduction

Large-scale cloud systems log system events for several purposes, such as system modeling [8, 25], error diagnosing [15, 52, 72], user behavior profiling [16, 18, 41], and security attack detecting [19, 54]. Large cloud providers can easily generate up to PBs of logs per day [45, 60, 69], 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 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 of time (typically 6-12 months [58, 69]), logs will be archived into *offline logs*.

The difference of their query patterns motivates different trade-offs among compression ratio, compression speed, and query latency. Online logs are queried frequently and thus prefer methods with a low query latency [6, 33]. Offline logs are almost never queried but need to be stored for a long time, and thus prefer methods with a high compression ratio [10, 13, 31, 45, 46, 48, 61, 67, 69]. Near-line logs require a careful thought about the trade-off. First, our experiments

*Corresponding Author: Guangyan Zhang (gyzh@tsinghua.edu.cn).
Junyu Wei, Guangyan Zhang, Junchao Chen, and Weimin Zheng are with the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China.

EuroSys '23, May 8–12, 2023, ROME, Italy

ACM ISBN 978-1-4503-9487-1/23/05.
https://doi.org/10.1145/3552326.3567484

show that the dominating cost varies depending on the compression method and parameters like how long the logs will be stored and query frequency. Second, though they are not queried frequently, an engineer expects a query to finish as quickly as possible, since a long query time will hurt her/his productivity when debugging a problem. During the conversation with Alibaba Cloud engineers, we find they prefer finishing a query within a few seconds, can accept a query taking less than a minute, but feel frustrated if a query takes several minutes or longer.

This paper targets designing a compression and query method for near-line logs with two goals. First, it should try to 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 try to limit query latency to an acceptable level. We have tested a number of existing works including ElasticSearch [6], CLP [58], etc, and found none can satisfy both goals. For example, it takes 14 minutes on average to execute a query using CLP, a state-of-the-art approach to query on compressed logs.

Structurizing logs by exploiting static patterns. To achieve both goals, we adopt a classic idea in data processing systems—split data into multiple partitions, compress each partition independently, and generate a summary for each partition to avoid decompressing irrelevant partitions when executing a query [12, 21, 42, 43, 63].

The key challenge in realizing this idea is *how to partition data, so 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. To achieve this goal, we leverage existing log parsing methods to structurize log entries into templates and variables [23, 30, 31, 45, 51, 66, 67, 69, 76], because values of the same variable are more likely to share common features [45, 69]. 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 paper. 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, 58, 73], 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 [69], 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 found this is mainly because summaries generated for the whole variable vectors are often still too general, which results in inefficient filtering.

Further improvement 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.

After exploring runtime patterns on a wide range of production logs, we found they widely exist. 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 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, comparing to partition data into whole 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, however, is challenging. General-purpose pattern extraction algorithms [4, 50, 53, 75] are too slow given the scale of our logs. As a result, prior works extract log patterns by 1) analyzing the source/binary code [7, 9, 70, 74], which only works for static patterns, by 2) applying heuristics [23, 31, 36, 47], which works well for static patterns but poorly for runtime patterns, since runtime patterns are more versatile, or by 3) setting default patterns or asking the developer to manually provide patterns [58], 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* [35] 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 their values have many duplicates and we only need to extract patterns on deduplicated values, the complexity

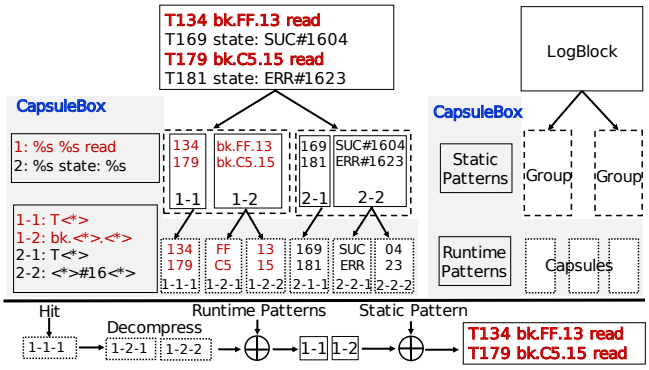


Figure 1. Fine-grained storage format and reconstruction process.

of the pattern extraction algorithm is less of an issue. Therefore, we design a *pattern merging approach* [31, 47], which has a time complexity of $O(n \log n)$ but can extract multiple patterns.

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 types of Alibaba Cloud production logs and 16 types of public logs, we compare LogGrep with CLP [58], a state-of-the-art method to compress and execute text query on logs, ElasticSearch [6], 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 comparable to ElasticSearch. Second, by considering the storage and computation cost in Alibaba Cloud, the overall cost of LogGrep is 36% of CLP, 7% of ElasticSearch, and 34% as much as that of gzip+grep.

Our contributions can be briefly summarized as follows:

- We propose LogGrep, the first 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 evaluated LogGrep on 21 types of real-world production logs [64] and found LogGrep achieves a significant query latency reduction and a considerable cost saving over the state-of-the-art system.
- We have open-sourced LogGrep [65].

2 Key Idea Roadmap

In Alibaba Cloud, applications first write raw logs, usually in text format, into 64MB blocks. We call these blocks *log blocks*. Then Alibaba Cloud compresses these log blocks in the background.

In general, the log-based debugging procedure is executed in two phases. In the first phase, the engineer may send full-text 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, etc. This work focuses on the query of the first phase since it may need to scan all logs and thus is the bottleneck.

During the first phase, 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, full-text queries on compressed log blocks are much more common.

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 [14]. However, this approach incurs a long query latency. We first present CLP [58], 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 a number of templates (called "log types" in CLP paper [58]), 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 [11] 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 an idea in data processing systems [12, 21, 42, 43, 63]: building summaries on each data partition to filter as many partitions as possible when executing a query. This approach creates two challenging and correlated questions: 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 difference. 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 compression ratio [31, 45].

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 left 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 combination is effective to create 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 6 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 dataset still take more than one minute, which are 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 and cheap cloud log storage. This idea is motivated by our following observations about runtime pattern.

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 length. 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. Comparing to 3.1 types of characters and 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.

3 System Overview

Based on our key idea, we design a system called LogGrep. Figure 2 shows the detailed architecture of LogGrep. We present the compression, query and reconstruction workflow to give an overview of the system.

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 [69]. 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 (§4.1) with the *Extractor*. Then the *Assembler* decomposes the whole variable vector into several fine-grained units called "Capsules" accordingly (§4.2) and generates a Capsule stamp for each Capsule (§4.3). Finally, the *Packer* compresses and packs all Capsules into a compressed file called "CapsuleBox" using LZMA [77], which is reported to have a high compression ratio [58, 69]. 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 provides a grep-like full-text query interface on compressed logs. 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 first parses a query command into several search strings and tokenizes each search string as keywords. Each keyword can be a part of static/runtime patterns or within a Capsule. The *Locator* executes keyword matching process with the help of runtime patterns to filter unrelated Capsules (§5.1) and finally executes fixed-length matching within decompressed Capsules (§5.2). LogGrep also has a *Query Cache* to store the results of past queries in a hashmap. The key of the hashmap is a query command, and the value is the corresponding location result.

Reconstruction. The previous step may return that the i^{th} entry in a group matches a query, and LogGrep needs to reconstruct the original log entry with a *Reconstructor*. To achieve that, Reconstructor first decompresses all Capsules of the corresponding group. For example, in Figure 1, if there is a hit on Capsule "1-1-1", Reconstructor needs to decompress "1-2-1" and "1-2-2". Reconstructor then fetches the i^{th} value of each Capsule. Since the Packer pads values to the same length in each Capsule, locating the i^{th} value is an O(1) operation. Then, from the CapsuleBox metadata, Reconstructor finds the corresponding static pattern of the group and the runtime pattern of each Capsule. Finally, Reconstructor fills values into the patterns to rebuild the original log entry/entries.

If the previous step returns multiple entries, Reconstructor needs to order them after reconstructing them. Entries from the same group are naturally ordered since LogGrep stores values in the same variable vector according to their original order appearing in the log block. For entries from different groups, 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.

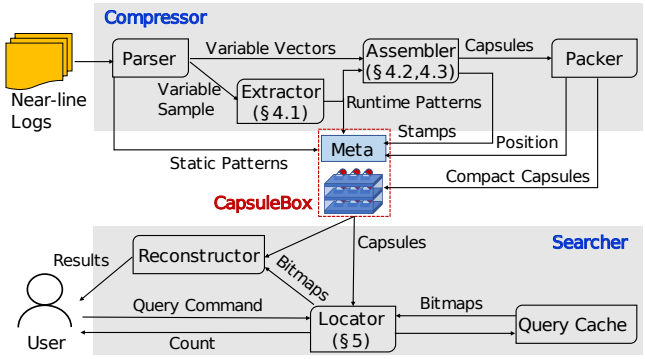


Figure 2. LogGrep architecture.

4 Log Structurization with Runtime Patterns

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.

4.1 Runtime Pattern Extraction

Pattern extraction is a well-studied field, but when testing with general-purpose pattern extraction methods [4, 50, 53, 75], we find they are slow given the scale of production logs.

To address this problem, 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, then 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.

Variable vector categorization. We first try to extract patterns with a general-purpose pattern extraction method [53] on samples of 13,238 variable vectors from 37 types of logs (21 from Alibaba Cloud, 16 from public logs). 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

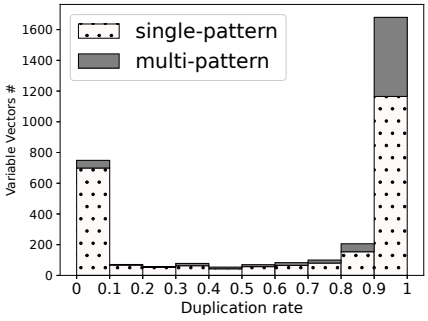


Figure 3. Distribution of single-pattern and multi-pattern variable vectors in respect of duplication rate.

but their unique values may have multiple patterns. This result motivates us to use the duplication rate of values, i.e., $(total\_count - unique\_count)/total\_count$, as a heuristic metric to determine whether a variable vector is dominated by one pattern.

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 3 presents the correlation between the likelihood 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, 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 apply 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 on unique values, the high duplication rate means there are fewer values to analyze, which reduces the overhead.

We need to choose a threshold to separate low and high duplicate rates. Considering the bathtub-like distribution in Figure 3, the efficiency of our approach is not very sensitive to this threshold, as long as it is somewhere in the middle. In this paper, we choose 0.5 as the threshold. We use *real variable vectors* to denote those with duplication rate under 0.5 and *nominal variable vectors* to denote those with 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 will need 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 [35] 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 4). Then LogGrep tries to expand the tree with multiple iterations. During an iteration, for each leaf node, if it is not marked as unsplitable, LogGrep chooses a delimiter, by either using a non-alphanumeric character from a randomly picked value, such as "_" in vector #1 in Figure 4, or the longest common sub-string (LCS) between two randomly picked values, such as "F8" in vector #3 in Figure 4. Then LogGrep tests if this delimiter can split the leaf node, namely at least 95% values containing the delimiters. LogGrep tries each delimiter for three times by extracting the delimiter from different randomly picked values, and if all fail, LogGrep marks the corresponding leaf node as unsplitable. If all leaf nodes are unsplitable, the expanding process terminates.

We choose non-alphanumeric character as delimiter 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 4). 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 have no correlation 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 [31, 47] for nominal variable vectors.

LogGrep first builds a temporary vector (vector #2 in Figure 5) 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

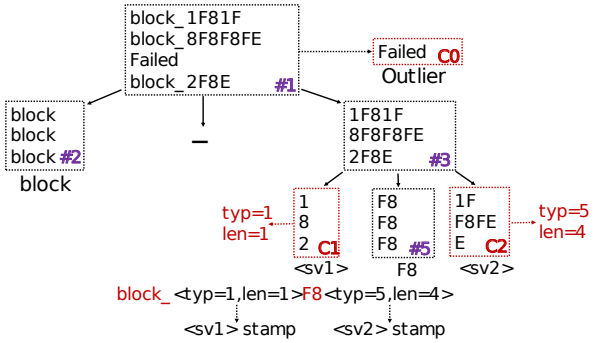


Figure 4. Runtime pattern extraction for real variable vectors. The original variable vector is stored into 3 Capsules: C0, C1 and C2. Extracted runtime pattern and Capsule stamps are shown at the bottom.

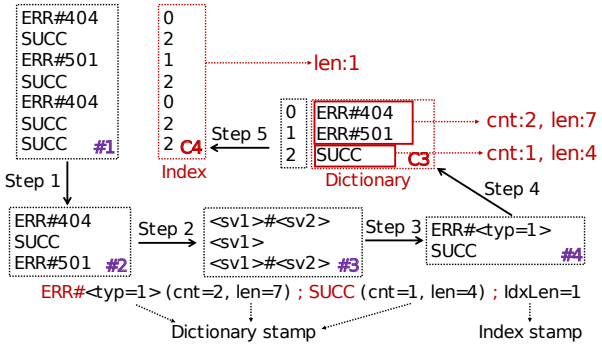


Figure 5. Runtime pattern extraction for nominal variable vectors. The original variable vector is stored into 2 Capsules: C3 and C4. Extracted patterns and Capsule stamps are shown at the bottom.

in Figure 5). 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 is a constant "ERR". Finally, LogGrep reorders the temporary vector by storing 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 in order 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 4). 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 4).

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).

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 computes its type in the same way as described in Section 2.2 (i.e., six bits to categorize decimal, hexadecimal, alphanumeric, or other data). 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 (§5.2).

Example of real variables vectors. For sub-variable vector "C1" in Figure 4, 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, LogGrep finds the type number of the sub-variable "<*>" is 000001b=1, since it only contains digits. Further, 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".

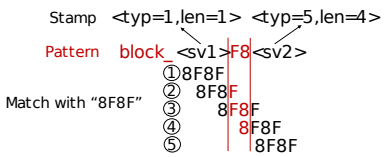


Figure 6. Possible matches for a keyword on a runtime pattern.

5 Keyword Matching with Runtime Patterns

When executing a query command, LogGrep splits the query command into several search strings using the logical operators. Each search string will be further tokenized as several keywords. LogGrep matches these keywords on static patterns like CLP (§2.1). We redesign the keyword matching process within a variable vector, since we further store variable vectors as fine-grained Capsules.

The keyword matching process within a variable vector includes two steps: 1) deciding which Capsules to decompress, and 2) matching given keyword(s) within decompressed Capsules. During the first step, LogGrep locates and filters Capsules based on the runtime patterns and Capsule stamps. During the second step, LogGrep matches the keyword(s) within a decompressed Capsule with fixed-length matching to further reduce the query latency.

5.1 Capsule Locating and Filtering

In order to decompress as few Capsules as possible for log queries, LogGrep first queries on the runtime pattern to locate all possible Capsules that may contain the keyword and filters these Capsules based on the Capsule stamps. The processes are similar on real variable vectors and nominal variable vectors. We first illustrate the process on real variable vectors in detail and then discuss the differences on nominal variable vectors.

Locating Capsules with runtime patterns. LogGrep can locate all relevant Capsules by matching the keyword on runtime patterns. 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, or a sub-string of any of the values following this runtime pattern. We only present our algorithm for sub-string matching, 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 6). 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 6). 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 6). 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 6). 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, if we require a certain sub-variable vector 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 6). The matching process on a Capsule may return a set of hit entries represented by their row numbers. When one keyword matching requires searching multiple Capsules, LogGrep takes two steps to produce the final result. First, one possible match may require multiple Capsules to have certain values and thus the final result of this possible match is the intersection of the matching result in each Capsule. Second, matching a keyword on a runtime pattern may lead to multiple possible matches, and the result of the matching should be the union of the results of all the possible matches.

Filtering Capsules with Capsule stamps. During the matching process of a keyword on a runtime pattern, we locate all candidate Capsules which are required to have a certain value. Before decompression, LogGrep will first check the stamp of each candidate Capsule and only decompress it if the sub-string of the keyword does not violate the restrictions recorded in the stamp.

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 "$\mathcal{K}\&C = \mathcal{K}$". If the check fails, 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 6 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 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 (§5.2). If an actual match is found, LogGrep can know the index number corresponding to the keyword and will search 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, 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.

5.2 Fixed-length Matching

Once a Capsule is decompressed, LogGrep will match a substring of keyword on all of its values using fixed-length matching. During the encapsulation process §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 and as mentioned above, 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 algorithm (BM algorithm) [5] to replace the traditional Knuth-Morris-Pratt algorithm [57] (KMP algorithm). The reason is that, if values can have a variant length, we have to add some delimiters to separate values. In this case, since the BM algorithm may skip characters, when it finds a matching string, it does not know the row number of the matching string, since it does not know how many delimiters have been skipped. If each value has a fixed length, LogGrep can compute the row number by dividing the matching position by the value length.

Second, as discussed in §5.1, if matching a keyword needs to scan multiple Capsules, LogGrep scans the first Capsule first, and if the matching returns some rows, LogGrep will check these rows in the second Capsule directly, instead of scanning all rows in the second Capsule. Padding all values to the same size makes such direct checking possible.

Finally, when LogGrep matches in the dictionary vector, it first needs to match the keyword to the patterns as discussed in §5.1. If a possible match is found, LogGrep can calculate the 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 is impossible if the value length is not fixed.

Our evaluation (§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 §2.3, values of the same sub-variable or pattern tend to have a similar length, so 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 three questions:

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

To answer these three questions, we measure the performance of LogGrep on 21 types of production logs (samples of them can be found in [64]) from Alibaba Cloud and 16 types of public logs [28]. 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 [58]. The detailed query commands can be found in the appendix.

LogGrep can work 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.

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 [58, 69], 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 [14]. It executes the query commands with these operators and UNIX pipe.
- ElasticSearch (ES) [6]: ElasticSearch is a classical query engine for logs and is the default choice for on-line logs in Alibaba Cloud. We use ElasticSearch 7.8.0 with the default settings, and use python SDE to insert logs to the index using "bulk" [27] method to accelerate the insertion process. ES supports logical operators and can execute query commands with the help of these operators.
- CLP [71]: 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 obscurest query, and then use grep to query with additional conditions with the help of UNIX pipe. By default, CLP uses zstd [17] as the second-stage compression tool. Comprared 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 §2.2).

Testbed. We perform all experiments on the Linux server with 2× Intel Xeon E5-2682 2.50GHz CPUs (with 16 cores), 188GB 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 be using one CPU.

6.1 Performance on Production Logs

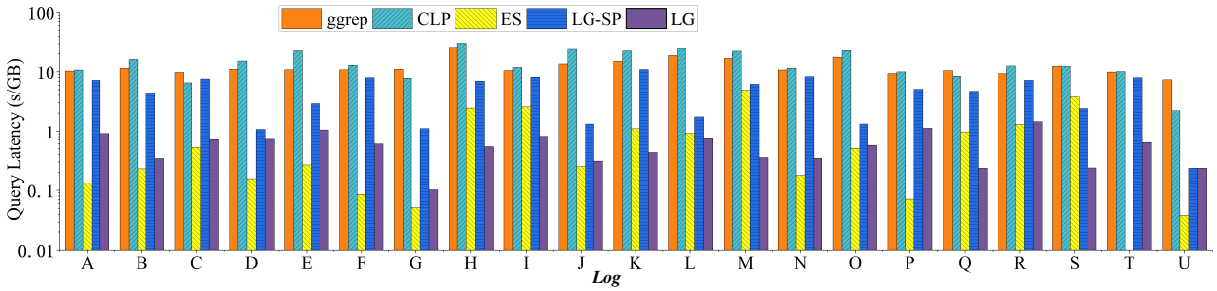
We evaluate the query latency, compression ratio and compression speed on 21 types of production logs (1.73TB in total) from Alibaba Cloud. These logs are from different cloud apps and have various characteristics. We anonymize log names due to privacy reasons. The total line number of Log T exceeds 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 7(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 types of logs, the query latency of LogGrep is lower, by up to 15.75×; on 13 types of 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 types of logs; on the other hand, LogGrep performs better if a query directly hits the template or the keyword is a sub-string of 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.

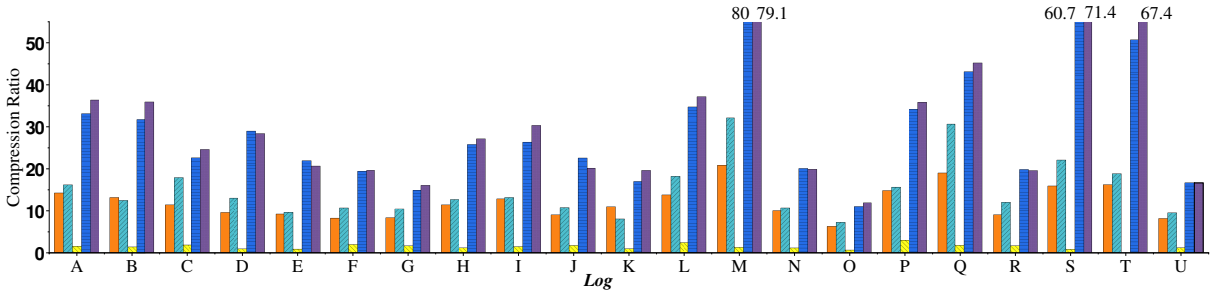
The query latency of LogGrep is longer than one minute only on Log T, since this log is as large as 964GB. Comparably, ggrep takes longer than one minute to execute a query on 11 logs; CLP takes longer than one minute on 12 logs; LogGrep-SP takes longer than one minute on 7 logs. ES finishes the queries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

Compression ratio. As shown in Figure 7(b), LogGrep has the highest compression ratio among ggrep, ES and CLP on all types of 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 types of logs by up to 1.33× and decreases on the other 6 types of 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.

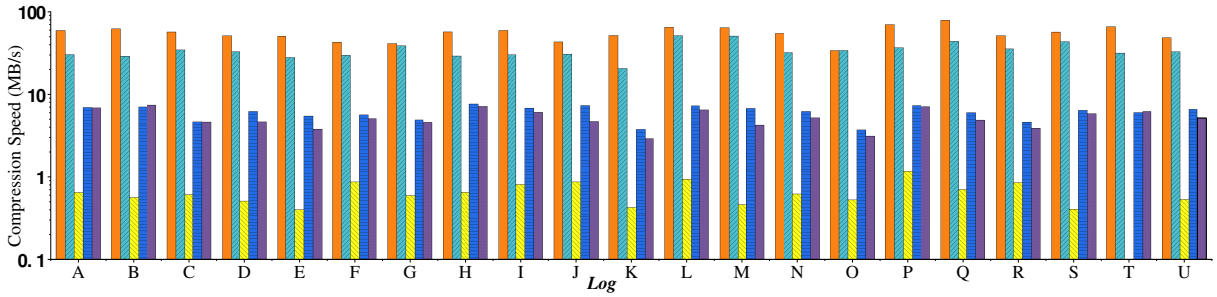
Compression speed. As shown in Figure 7(c), the compression speed of LogGrep is lower than gzip and CLP and higher than ES. Specifically, the compression speed of LogGrep is 0.06× to 0.12× (0.10× on average) as much as that of gzip, 0.08× to 0.26× (0.16× on average) as much as that of CLP, and 4.51× 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 types of 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 two types of logs since



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

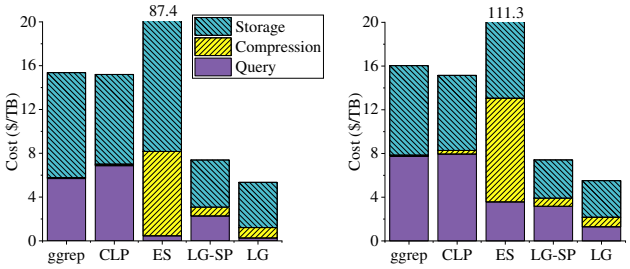


(b) Compression ratio on Alibaba Cloud logs



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

Figure 7. Query latency, compression ratio and compression speed on 21 Alibaba Cloud logs.



(a) Alibaba Cloud logs (b) Public logs

Figure 8. Overall cost.

LogGrep-SP takes more time to compress coarse storage units.

Such a slow-down 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 8(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 compared with ggrep, CLP and ES on all tested logs. Its cost is a little bit 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 sometimes 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.

If we consider higher querying frequency, LogGrep's benefit over ggrep and CLP will grow since LogGrep's query latency is significantly lower than these two. For the 13 types of 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.

6.2 Performance on Public Logs

We also evaluate query latency, compression ratio and compression speed on 16 public logs benchmark [28, 32] (77GB in total), which are from various scope 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 types of logs, the query latency of LogGrep is lower, by up to 12×; on 5 types of 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).

Compression ratio. LogGrep has the highest compression ratio among ggrep, ES and CLP on all types of 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 types of logs by up to 1.33×, and decreases on other 4 types of 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 types of logs and is slightly higher than that of LogGrep-SP on two types of logs by up to 1.07×.

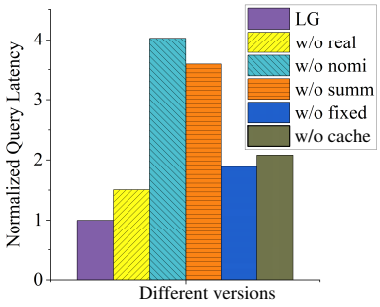


Figure 9. Effects of individual techniques on Alibaba Cloud logs (normalized to full-versioned LogGrep).

Overall cost. As shown in Figure 8(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. In order 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 impact of individual techniques proposed by this paper. We implement five versions of LogGrep, each removing an individual technique: "w/o real" removes runtime pattern extraction and fine-grained structurization in real variable vectors (§4.1); "w/o nomi" does similar for nominal variable vectors (§4.1); "w/o stamp" removes the stamp of each Capsule and thus does not filter Capsules with their stamps during keyword matching (§4.3); "w/o fixed" removes the padding process and queries on variant-length "Capsules" with the KMP algorithm (§5.2); "w/o cache" removes the Query Cache and re-executes prior queries. 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 9 shows the average query latency of each version after normalized to the full-version. Specifically, 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.

Runtime pattern extraction and encapsulation in nominal variable vectors bring the most significant reduction to query latency, because nominal variable vectors take a larger space compared with real variable vectors. Capsule stamp can bring up to 15× reduction to the query latency (on Log B), due to its stamp is strict, such that LogGrep can filter out more Capsules with the help of it. Fixed-length padding

and matching can bring improvement to 20 logs. On some logs (e.g., Log G, Log O, Log R), it can even bring up to 4× reduction to query latency. The only exception is Log N, on which this technique causes a slightly higher (5%) latency since the length of its values within a Capsule varies a lot.

We also evaluate the effect of fixed-length padding on compression ratio. We find the compression ratio with padding is 0.99× to 1.10× (1.04× on average) as much as that with no padding. Padding only reduces the compression ratio on one type of log (Log F) by 1%. This means on most logs, the overhead of padding is smaller than the overhead to add a delimiter to separate different values.

7 Related Work

Previous log management tools mainly target either high compression ratio or 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 this idea, and proposes to exploit both static and runtime patterns to partition data so 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 still maintaining a low storage cost.

Log compression and query. The methods of compressing large logs can be categorized into two groups: bucket-based methods [20, 29, 44] and parser-based methods [45, 69]. Bucket-based methods first group logs into different buckets according to their similarity and compress each bucket individually. Parser-based methods parse log into templates and variables and then compress variables from the same template individually. These methods usually have a high compression ratio, but to execute a query, one needs to decompress data first. LogGrep inherits the parser from parser-based methods and further accelerates queries on compressed logs.

Both general-purpose text query tools, such as ElasticSearch [6] and Splunk [33], and log-specific query tools, such as Scalyr [34] and Loki [39], can achieve low query latency on cloud logs. However, since they do not take compression into consideration, compared with LogGrep, their overall cost are unsatisfactory for cloud logs.

Query on compressed data. These works can be categorized into two types: First, some approaches [37] can query on encoded data directly without any decompression [3, 22, 26, 38, 55, 56, 59, 61, 73, 78] However, since these methods need to keep the interpretability of encoded data, they cannot compress data with higher compression ratio methods (e.g., LZMA). As a result, their overall storage cost is unsatisfactory [58].

Second, the idea of partitioning data into independent units and filtering irrelevant units has been proposed by database community [1, 2, 12, 40] and been applied by CLP, the state-of-the-art log compression and query tool. However, according to our experiments (§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.

Log pattern extraction. General pattern extraction methods on text data are well-studied [4, 24, 50, 53, 68, 75], but their time complexity are relatively high and thus they are slow on large-scale log data [76]. Previous log pattern extraction on logs mainly focus on extracting static patterns automatically by source/binary code analysis [7, 9, 70, 74] or applying heuristic methods [30, 36, 47, 49, 51, 62, 66, 67]. We propose a novel method to extract runtime patterns automatically to partition data in fine-grained units with common features.

8 Conclusion and Future Work

This work proposes a novel log compression and query tool called LogGrep, which exploits both static and runtime patterns to structurizes logs in fine-grained Capsules. Our evaluation shows this design enables effective summaries and can accelerate queries and reduce overall cost significantly.

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

Acknowledgment

We thank all reviewers for their insightful comments, and especially our shepherd, Ding Yuan, for his guidance during our camera-ready preparation. We also thank Kirk Rodrigues with YScope Inc. for helping us running the CLP system. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

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 *2014 fourth international conference on communication systems and network technologies*. IEEE, 428–432.

[5] R. Boyer and J. S. Moore. 1977. A fast string searching algorithm. *Commun. ACM* 20 (1977), 762–772.

[6] Elasticsearch B.V. 2020. Elasticsearch 7.8.0. https://www.elastic.co/downloads/past-releases/elasticsearch-7-8-0.

[7] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. 2021. LogStore: A Cloud-Native and Multi-Tenant Log Database. In *Proceedings of the 2021 International Conference on Management of Data*. 2464–2476.

[8] 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.

[9] 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.

[10] John Cleary and Ian Witten. 1984. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications* 32, 4 (1984), 396–402.

[11] 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.

[12] 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.

[13] Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. https://tools.ietf.org/html/rfc1951.

[14] Linux developer community. 2020. Grep Manuel. https://linux.die.net/man/1/grep.

[15] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1285–1298.

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

[17] Facebook. 2021. z-standard compression tool. https://github.com/facebook/zstd.

[18] 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.

[19] 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.

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

[21] Sérgio Fernandes and Jorge Bernardino. 2015. What is bigquery?. In *Proceedings of the 19th International Database Engineering & Applications Symposium*. 202–203.

[22] Paolo Ferragina and Giovanni Manzini. 2001. An experimental study of a compressed index. *Information Sciences* 135, 1 (2001), 13–28.

[23] 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.

[24] 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.

[25] 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 2004 IEEE Wireless Communications and Networking Conference*, Vol. 1. IEEE, 120–125.

[26] Roberto Grossi and Jeffrey Scott Vitter. 2005. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.* 35, 2 (2005), 378–407.

[27] ElasticSearch group. 2019. Elasticsearch: Bulk Inserting Examples. https://queirozf.com/entries/elasticsearch-bulk-inserting-examples.

[28] Loghub group. 2019. Download link of public log dataset. https://zenodo.org/record/7056802#.Yxm2VexBwq2.

[29] LogArchive group. 2019. Open source code of LogArchive. https://github.com/robertchristensen/log_archive_v0.

[30] 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.

[31] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *Proceedings of 2017 IEEE International Conference on Web Services*. IEEE, 33–40.

[32] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2020. Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. *arXiv preprint arXiv:2008.06448* (2020).

[33] Splunk Inc. 2020. Spunk Enterprise 8.0.3. https://www.splunk.com/en_us/download/previous-releases.html.

[34] Scalyr Inc. 2021. Scalyr home page. https://www.scalyr.com/.

[35] 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.

[36] 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.

[37] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. 1998. Multiple pattern matching in LZW compressed text. In *Proceedings DCC '98 Data Compression Conference (Cat. No.98TB100225)*. 103–112.

[38] Stefan Kurtz. 1999. Reducing the space requirement of suffix trees. *Software - Practice and Experience* 29, 13 (1999), 1149–1171.

[39] Grafana Labs. 2021. Loki Documentation. https:grafana.com/docs/loki/latest/.

[40] 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.

[41] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. 2012. The unified logging infrastructure for data analytics at Twitter. *arXiv preprint arXiv:1208.4171* (2012).

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] Adetokunbo AO 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.

[48] 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.

[49] Masayoshi Mizutani. 2013. Incremental mining of system log format. In *Proceedings of 2013 IEEE International Conference on Services Computing*. IEEE, 595–602.

[50] 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.

[51] 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.

[52] 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.

[53] Frank Nielsen. 2016. Hierarchical clustering. In *Introduction to HPC with MPI for Data Science*. Springer, 195–211.

[54] 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.

[55] Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. 2019. Fast Dictionary-Based Compression for Inverted Indexes. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. 6–14.

[56] Giulio Ermanno Pibiri and Rossano Venturini. 2020. Techniques for Inverted Index Compression. *Comput. Surveys* 53, 6 (2020), 1–36.

[57] Mireille Régnier. 1989. Knuth-Morris-Pratt algorithm: an analysis. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 431–444.

[58] Kirk Rodrigues, Yu Luo, and Ding Yuan. 2021. CLP: Efficient and Scalable Search on Compressed Text Logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. 183–198.

[59] Kunihiko Sadakane. 2002. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. 225–232.

[60] Vijay Samuel. 2018. Monitoring Anything and Everything with Beats at eBay. (2018).

[61] Khalid Sayood. 2017. *Introduction to data compression*. Morgan Kaufmann.

[62] Keiichi Shima. 2016. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213* (2016).

[63] 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.

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

[65] LogGrep authors. 2022. Source code of LogGrep. https://github.com/THUBear-wjy/LogGrep.

[66] 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.

[67] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management*. IEEE, 119–126.

[68] 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.

[69] Junyu Wei, Guangyan Zhang, Yang Wang, Zhiwei Liu, Zhanyang Zhu, Junchao Chen, Tingtao Sun, and Qi Zhou. 2021. On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*. 249–262.

[70] 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.

[71] YScope. 2021. clp-core. https://github.com/y-scope/clp-core.

[72] 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.

[73] 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.

[74] Maosheng Zhang, Ying Zhao, and Zengmingyu He. 2017. Genlog: Accurate log template discovery for stripped x86 binaries. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 337–346.

[75] 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.

[76] 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.

[77] 7 zip developer group. 2019. 7-zip file achiever home page. https://www.7-zip.org/.

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

A Artifact Appendix

A.1 Abstract

The artifact of this paper includes the source code of LogGrep and also the queries we use to query open logs. The reproduce procedure is mainly done on open logs, but we also open samples of our production logs after removing some sensitive messages.

A.2 Description & Requirements

A.2.1 How to access.

Our Source code is at:
https://github.com/THUBear-wjy/LogGrep/tree/master

A.2.2 Hardware dependencies.

- CPU: 2× Intel Xeon E5-2682 2.50GHz CPUs (with 16 cores)
- RAM: 188GB

A.2.3 Software dependencies.

- OS: Red Hat 4.8.5 with Linux kernel 3.10.0
- Compiler: gcc version 4.8.5 20150623

A.2.4 Benchmarks.

- The open dataset can be found at:
  https://zenodo.org/record/7056802#.Yxm2VexBwq2
  Open dataset inlcudes 16 types of logs with 76GB in total.
- The queries we tested on production logs and public logs can be found in Table 1
- We also open samples of our production logs at:
  https://github.com/THUBear-wjy/openSample.

A.3 Set-up

The compilation and quick test procedure can be found at the README.md file in our code repository.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): LogGrep achieves an order of magnitude query latency saving compared with gzip+grep when executing our listed queries and its overall cost is 34% as much as that of gzip+grep. This is proven by the experiment (E1) described in §6 whose results are illustrated/reported in §6.2.
- (C2): LogGrep achieves comparable latency compared with ES when executing our listed queries in code repository and its overall cost is 5% as much as that of ES. This is proven by the experiment (E2) described in §6 whose results are illustrated/reported in §6.2.
- (C3): LogGrep achieves an order of magnitude query latency saving compared with CLP and its overall cost is 41% as much as that of CLP. This is proven by the experiment (E3) described in §6 whose results are illustrated/reported in §6.2.

A.4.2 Experiments.

Experiment (E1): [Compare with gzip+grep]: To compare LogGrep with gzip+grep system.
[Preparation] Use linux "grep" to query logs and use linux gzip of version 1.5 to compress logs.
[Execution] Execute LogGrep in the way we described in our code repository. Execute "gzip" to compress logs and record the compression time and compressed size. Before query, first unzip compressed logs and record the decompression time and use "grep" to query on logs and record the query time. Final result
[Results] The query latency of LogGrep is 2.27x to 51.25x (14.56x on average) lower than that of gzip+grep. Overall cost is 34% as much as that of gzip+grep on average.

Experiment (E2): [Compared with ES]: To compare LogGrep with Elasticsearch system.
[Preparation] Install and configure ElasticSearch of version 7.8.0 at
https://www.elastic.co/downloads/past-releases/elasticsearch-7-8-0
[Execution] Execute LogGrep in the way we described in our code repository. Test ES with python API, use "bulk" insertion to insert logs into ES index, use "curl -X GET localhost:9200/_cat/indices?v" to see the storage consumption of index. Use python API to query with ES index and record the query latency
[Results] The query latency of LogGrep is comparable compared with ES (On Android, Hdfs, Hadoop, Thunderbird, Winodws, LogGrep has a higher latency by up to 12.23x. On other types of logs, LogGrep has a lower latency by up to 12x.) Overall cost is 5% as much as that of ES on average.

Experiment (E3): [Compare with CLP]: To compare LogGrep with CLP.
[Preparation] Install and configure CLP with instructions listed in https://github.com/y-scope/clp-core
[Execution] Execute LogGrep in the way we described in our code repository. Compress Logs with CLP and record the storage cost and compressino speed. Query with "clg" command in CLP. Since CLP can not process logic operators like "and" and "not", we use CLP to execute the first part connected by logic operators and use grep to execute the following part.
[Result] The query latency of LogGrep is 1.94x to 42.00x (13.74x on average) lower than that of CLP. Overall cost is 41% as much as that of CLP on average.

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

Table 1. Query commands used in evaluations. Some characters are changed to "?" due to privacy limitation.