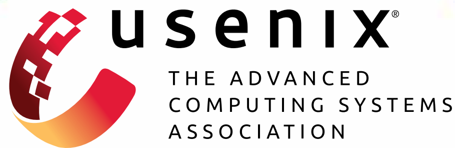

CLP: Efficient and Scalable Search on Compressed Text Logs

Kirk Rodrigues, Yu Luo, and Ding Yuan, *University of Toronto and YScope Inc.*

https://www.usenix.org/conference/osdi21/presentation/rodrigues

This paper is included in the Proceedings of the
15th USENIX Symposium on Operating Systems
Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Open access to the Proceedings of the
15th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by USENIX.

CLP: Efficient and Scalable Search on Compressed Text Logs

Kirk Rodrigues, Yu Luo, Ding Yuan
University of Toronto & YScope Inc.

Abstract

This paper presents the design and implementation of CLP, a tool capable of losslessly compressing unstructured text logs while enabling fast searches directly on the compressed data. Log search and log archiving, despite being critical problems, are mutually exclusive. Widely used log-search tools like Elasticsearch and Splunk Enterprise index the logs to provide fast search performance, yet the size of the index is within the same order of magnitude as the raw log size. Commonly used log archival and compression tools like Gzip provide high compression ratio, yet searching archived logs is a slow and painful process as it first requires decompressing the logs. In contrast, CLP achieves significantly higher compression ratio than all commonly used compressors, yet delivers fast search performance that is comparable or even better than Elasticsearch and Splunk Enterprise. In addition, CLP outperforms Elasticsearch and Splunk Enterprise's log ingestion performance by over 13x, and we show CLP scales to petabytes of logs. CLP's gains come from using a tuned, domain-specific compression and search algorithm that exploits the significant amount of repetition in text logs. Hence, CLP enables efficient search and analytics on archived logs, something that was impossible without it.

1 Introduction

Today, technology companies easily generate petabytes of log data per day. For example, eBay reported generating 1.2 PB of logs per day in 2018 [46]. This data can be used for a variety of important use cases including security forensics, business insights, trend analysis, resource optimization, and so on. Since many of these cases benefit from large amounts of data, companies strive to retain their logs for as long as possible. Moreover, some industries (e.g., health services) are required by law to store their logs for up to six years [5].

However, storing and analyzing a large amount of log data impose significant costs. Although it is difficult to obtain transparent, publicly available information about companies' storage costs, studies have estimated that a lower bound for capital depreciation and operational costs could be on the order of two cents per gigabyte, per month [7]. For a company like eBay, this translates to over $50 million to store the logs generated in a year, and nearly $500 million to store the logs generated over three years. As a result, the log management industry has grown incredibly large.

Currently, Elastic [2] and Splunk [4] are two of the largest companies in the industry. In just their last fiscal year, Elastic reported revenue of $428 million with a total of 11,300 customers [14] while Splunk reported revenue of $2.359 billion with 19,400 customers [36]. Moreover, their offerings, Elasticsearch [15] and Splunk Enterprise [37], are used by several large companies like eBay, Verizon, and Netflix.

Tools like Splunk Enterprise and Elasticsearch operate by generating external indexes on the log messages during ingestion. Then in response to a query, these tools can quickly search the indexes corresponding to the logs, decompressing only the chunks of data that may contain logs matching the search phrase. Elasticsearch, for example, is built around a general-purpose search engine Lucene [42]. However, this approach comes at the cost of a large amount of storage space and memory usage. Although these tools apply light compression to the logs, the indexes often consume an amount of space that is the same order of magnitude as the raw logs' size; furthermore, these indexes must be kept mostly in memory or on fast random access storage in order to be fully effective. Thus, Splunk Enterprise and Elasticsearch users with large amounts of data can only afford to retain their indexed logs for a short period, typically a few weeks [8].

To avoid discarding logs at the end of their retention period, companies can use industry-standard compression tools like Gzip [21] to archive them, potentially gaining a 95% reduction in storage costs. In addition, recent advancements in compression algorithms like Zstandard [16] bring significantly improved compression and decompression speeds. However, these general-purpose compressors are not designed with search (on compressed data) in mind. They typically encode duplicates in length-distance pairs [40, 49], i.e., starting from the current position, if the next L (length) characters are the same as the ones starting at D (distance) behind, we can encode the next L characters with (D,L), and directly embed this pair at the current position, an approach known as an internal macro scheme [40]. Performing searches on this archived data, however, is painful and slow—the tool needs to sequentially scan the entire data set, essentially decompressing the data. This leads to the unfortunate reality that *log analysis and log archiving are generally mutually exclusive*.

To bridge this gap, we have created a method for lossless log compression that still allows efficient searches on the compressed data, without the need to decompress every log message. It works by combining a domain-specific compression and search algorithm with a lightweight general-purpose compression algorithm. The former allows us to achieve a modest amount of compression without sacrificing search performance. The latter increases the overall compression ratio significantly with a minor impact on compression and decompression performance.

The domain-specific compression algorithm uses an external macro scheme, i.e., it extracts the duplicated patterns into a dictionary that is stored separately from the encoded log messages [40]. A search query will be processed by first searching in the dictionary, and then searching those encoded messages for which the dictionary search suggests possible matches. This method relies on the simple observation that today's software logs contain a large amount of repetitive static text. By systematically separating the static text from the variable values, the algorithm can deduplicate the static text into a dictionary. Applying a similar process to the variable values, the algorithm converts an entire log message into a series of integers and dictionary entries that are easily compressible using a general-purpose compressor.

The search process similarly encodes the query string as a compressed message and searches for a match; but supporting queries with wildcards makes this process significantly more involved. For example, a wildcard can make it ambiguous whether a token is part of the message's static text or whether it is part of a variable. As a result, the algorithm must consider the effect of wildcards at every stage of the encoding and search process.

Using this method of compression and search, we have built an end-to-end log management tool, CLP[1], that enables real-time data ingestion, search, analytics, and alerting on top of an entire history of logs. CLP is agnostic to the format of logs and can ingest heterogeneous and unstructured logs. As a result, CLP is capable of reducing the size of currently archived logs while simultaneously enabling search and analytics on the compressed data.

Our evaluation shows that CLP's compression ratio is significantly higher compared to all tested compressors (e.g., 2x of Gzip), while enabling efficient search on compressed data. This comparison even includes industry-standard tools like Zstandard at their highest (and slowest) compression level. Furthermore, CLP's search speed outperforms commonly used sequential search tools on compressed data by 8x in a wide range of queries. Even compared with index-based log-search tools Splunk Enterprise and Elasticsearch, CLP outperforms them by 4.2x and 1.3x respectively. CLP's distributed architecture further allows it to scale to petabytes of logs. CLP is open-sourced and can be found at `https://yscope.com`. It is also hosted in the cloud so users can use it as a service.

[1] CLP stands for Compressed Log Processor

CLP's main limitation is that its algorithm is designed primarily for text logs. This is not a problem in the vast majority of software logs that we have seen, but we acknowledge that there are projects that log primarily binary or structured data. However, if converted to text with a verbose schema, these logs can be compressed and searched using CLP without additional overhead.

The rest of this paper is organized as follows. §2 describes the core elements of CLP's design for compression and search. §3 details how CLP handles the various intricacies of handling wildcards and patterns of variables. §4 describes our syntax for variable patterns. §5 explains how CLP can cache queries in reusable manner for performance. §6 describes a characteristic of CLP's compression format that can be used for privacy control. §7 discusses the evaluation results of CLP compared with other tools. Finally, §8 discusses related work, before we conclude in §9.

2 Design Overview

CLP is a complete end-to-end system for ingesting, archiving, searching, and analyzing log messages. Figure 1 shows an overview of CLP's compression and search architecture. Within the compression architecture, logs can be ingested either through CLP's real-time ingestion engine (e.g., from rsyslog, Fluentd, Logstash, etc.) or by reading them directly from local or cloud storage (e.g., Amazon S3 [29]). The compression nodes compress the ingested logs into a set of archives. Users can access the compressed logs transparently using a Unix terminal through the Filesystem in Userspace (FUSE) layer or by querying them through CLP's search interface.

CLP allows users to query their logs using a wildcard search followed by a series of operators. An example query is shown in Figure 2, containing four commands pipelined with a Unix-style pipe ('`|`'). The first command returns all log messages matching the search phrase ('`*`' is a wildcard character that matches zero or more characters). Results are piped to the `regex` operator which uses a regular expression to extract the container ID and operation runtime, storing them in user defined variables. Next, the `filter` operator filters for runtimes that are above "`0.1`". Finally, the `unique` operator generates a list of unique container IDs that satisfy the filter. Overall, this query returns the unique containers where the assignment operation took over `0.1` seconds in the `172.128.*.*` subnet. We refer to this type of query as a *pipelined query*.

CLP's search architecture supports pipelined queries by combining search nodes with a MapReduce-style [11] framework. CLP receives queries through its web UI or Python APIs. Queries are first serviced by the search nodes which perform a wildcard search on the archives. Results are then forwarded to the operator nodes, after which the final results are sent back to the user. Users can also create alerts that trigger when newly added log messages satisfy a saved query.

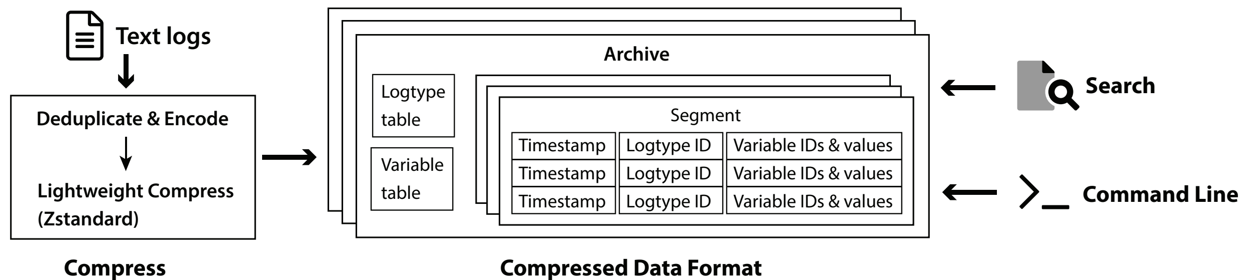

Figure 1: Overall architecture of CLP.

```
"Task * assigned to container*:172.128" |
regex "(?<container>container_\d+).* took (?<runtime>\d+)"
| filter float(runtime) > 0.1 | unique container
```

Figure 2: A query example. CLP operator and keywords are in blue, and user-defined variables are in red.

Note that because search is the first stage of every query, it is also the most important for performance since it operates on compressed data and all other stages operate on the decompressed data it outputs.

We aim to satisfy three objectives with this design: First, logs should be compressed losslessly so that users can delete their original logs without worrying that CLP would destructively transform them (e.g., by changing the precision of floating-point values). Second, users should be able to search their logs for any value, in contrast to index-based search tools which typically only allow searches for indexed values. For example, unlike grep-like tools that respect all characters in a search phrase, indexed-based search tools typically ignore punctuation and stop words (e.g., "and"). Finally, CLP should be performant and scalable so that users can use it to ingest and search a large amount of log data while saving on storage costs. By satisfying these objectives, we aim to bridge the gap between conventional log archival and search, e.g., using gzip and grep, and large-scale log analysis, e.g., using Splunk Enterprise or Elasticsearch.

The core of CLP is implemented in C++ for performance while higher-level functionality is built in a variety of languages from Java to JavaScript.

2.1 Compression

CLP's compression consists of two steps: first it deduplicates highly repetitive parts of each log message and encodes them in a special format, then it applies a lightweight compressor to the encoded data, further reducing its size. This section focuses on explaining the first step.

CLP splits each message into three pieces: 1) the log type, which is typically the static text that is highly repetitive, 2) variable values, and 3) the timestamp (if the message contains one). CLP further separates variable values into two categories: those that are repetitive, such as various identifiers (e.g., a username), and those that are not (e.g., a job's completion time). We refer to the former as *dictionary variables* since CLP extracts and stores them in a dictionary; the latter are called *non-dictionary* variables. Figure 3 shows a log message and how CLP converts it into a compressed form. Overall, this requires parsing the message to extract the aforementioned components and then compressing it using CLP's domain-specific compression algorithm.

2.1.1 Parsing Messages

CLP parses logs using a set of user-specified rules for matching variables. For example, Figure 4 lists a set of rules that can be used to parse the example log message. Lines 3–5 contain three dictionary variable schemas and line 8 contains a non-dictionary variable schema. This is similar to tools like Elasticsearch and Splunk Enterprise that either provide application-specific parsing rules or ask users to write their own. CLP provides a default set of schemas that can be applied universally to all log formats, or users can optimize them to achieve better compression and faster searches on their workloads.

One challenge with using variable schemas is that they can match pieces of a log message in multiple ways. For instance, "`172.128.0.41`" could match the schema for an IP address or it could match two instances of the floating point number schema, joined by a period. However, we have observed that developers typically separate different variable values with one or more *delimiter* characters. Furthermore, they also use delimiters to separate variable values from static tokens in the log type. We call this the *tokenization rule*, which states that *a token is inseparable*. That is, an entire token is either a variable value or part of the log type. In this case, "`172.128.0.41`" will be treated as a single token, so it can only match an IP address instead of two floating point numbers joined by a period. Accordingly, CLP allows users to specify a set of delimiters that ensures their schemas only match variables in a way that respects the tokenization rule.

To parse a log message, CLP first parses and encodes the message's timestamp as milliseconds from the Unix epoch. CLP then tokenizes the log message using the user-specified

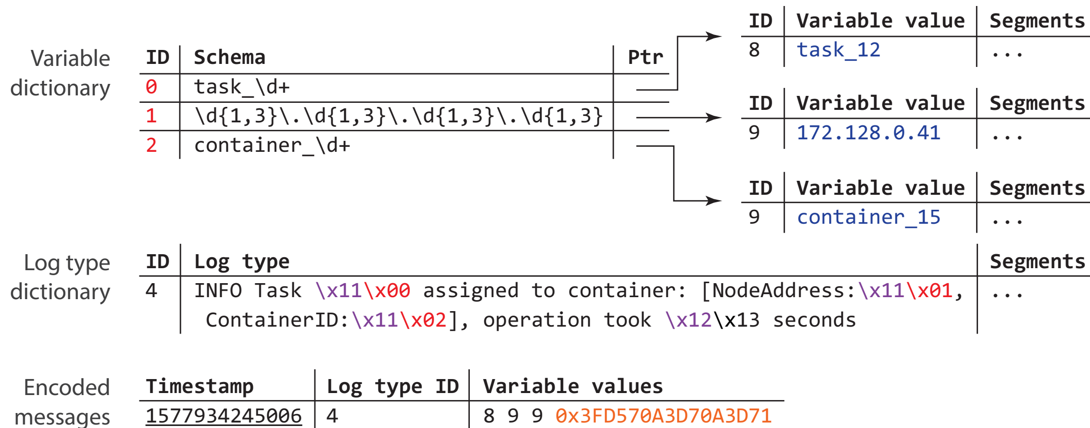

Figure 3: A log message and its encoding. Dictionary variables are in blue; Non-dictionary variables are in orange.

```
1  delimiters: "[],: "
2  dictionary_variables:
3    "task_\d+"                              # Task ID
4    "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}" # IP
5    "container_\d+"                     # Container ID
6
7  non_dictionary_variables:
8    "\d+\.\d+" as floating_point_number
```

Figure 4: Schemas used to parse the example in Figure 3.

delimiters. For each token, CLP compares it with each variable schema to determine whether it is a variable value. In Figure 3, CLP identifies three dictionary variables in the log message—"task_12", "172.128.0.41", and "container_15"—and a non-dictionary variable value, "0.335".

2.1.2 Compressing Messages

Once parsed, the dictionary variables are stored in a two-level variable dictionary, referred to as a *vDict*. The first level maps each dictionary variable schema to a unique ID. Each schema is also mapped to a pointer that points to the second level of the vDict, where the actual variable value is stored. In Figure 3, the schemas for the task ID, IP address, and container ID are mapped to IDs 0, 1, and 2 in the first level, and the actual variable values are stored in the second level.

Non-dictionary variable values are stored directly in the encoded log message if possible. For example, "0.335" is encoded using the IEEE-754 standard [1] and stored as a 64-bit value in the encoded message. CLP currently supports encoding floating point numbers and integers as non-dictionary variables. If a non-dictionary variable cannot be encoded precisely within 64-bits (e.g., its value overflows), it is stored as a dictionary variable instead. Non-dictionary variables tend to be unique values like counters, so they do not benefit from being stored in a dictionary. We use a fixed-width 64-bit encoding instead of a variable-width encoding because it is simple to implement, and the space inefficiency is diminished by the lightweight compressor applied to the encoded data.

The remaining portion of the log message is treated as being part of the log type, where variable values are replaced with special placeholder characters. Each unique log type is stored in the log type dictionary, or *ltDict*, and is indexed by an ID. CLP uses byte '\x11' to represent a dictionary variable value. The next one or more bytes after '\x11' are an index into the vDict's first level, i.e., an index to the variable schema. In Figure 3, '\x00', '\x01', and '\x02' in the log type are indices to the three schemas for the task ID, IP address, and container ID in the vDict. CLP uses '\x12' as the placeholder for a floating point non-dictionary value. The next byte, '\x13', in the log type indicates that there is one digit before and three digits after the '.' character in the raw log message, ensuring the floating point value can be losslessly decompressed.

Note that we could choose any bytes for the placeholder characters, but since '\x11' and '\x12' are not printable ASCII characters, they are unlikely to appear in text logs. If they do, CLP will escape them before insertion into the log type.

CLP outputs the encoded message as a tuple with three elements as shown in Figure 3: a 64-bit timestamp, a 32-bit log type ID, and a sequence of 64-bit variable IDs and encoded variable values.

We have experimented with additional encoding schemes that can further reduce the size of the encoded data, but decided not to adopt them due to their undesirable trade-off. For example, we could store variable IDs and non-dictionary variable values using a variable-length encoding, instead of a fixed-length 64-bit encoding. We have also experimented with delta encodings, run-length encodings, and so on. However, these would come at the cost of search performance since

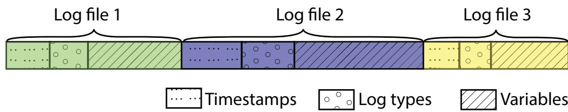

Figure 5: Storing encoded messages in column-oriented manner. It shows a segment that contains three encoded log files.

it is faster to scan fixed-length values than variable-length values. Moreover, the space savings are negligible after the lightweight compressor is applied on the encoded data.

2.1.3 Decompressing Messages

CLP's decompression process is generally a reversal of the compression process. Given an encoded message, CLP uses the message's log type ID to find the corresponding log type in the ltDict. CLP then reconstructs the variable values and replaces the placeholders in the log type. For example, CLP reconstructs the variable value "`task_12`" in Figure 3 as follows: the first '`\x11`' in the log type indicates that it is a dictionary variable, so CLP uses the next byte, '`\x00`' as an index into the first level of the vDict. CLP then uses the variable ID stored in the encoded message (8 in this case) to index the corresponding second level of the vDict, and restores the variable value "`task_12`". Finally, CLP converts the timestamp back to text and inserts it into the message.

2.1.4 On-disk Format

Figure 1 also shows the on-disk format of CLP's compressed logs. CLP encodes each message and stores them in the same temporal order as in the original log file. This ensures the file can be losslessly decompressed. The encoded messages are initially buffered in memory, and once the buffer reaches a certain size, they are compressed using Zstandard before being written to disk, creating what we call a *segment*.

Encoded messages are stored in a column-oriented manner [39], as shown in Figure 5—CLP stores the timestamp column of the messages from log file 1, then its log type IDs, and finally the variable IDs and values column, before storing the three columns of the next log file. Storing columnar data-series reduces data entropy within Zstandard's compression window, significantly improving compression ratio. In addition, columnar data-series can improve search performance: for instance, if users search for a message in a specific time range, CLP can skip messages outside that time range by only scanning the timestamp column rather than all columns.

Multiple segments further belong to an *archive*, where all segments in an archive use the same log type and variable dictionaries. CLP automatically closes and creates a new archive when the dictionaries reach a size threshold. This ensures that the dictionaries do not grow too large such that they have non-negligible loading times for decompression and search. CLP also compresses the dictionaries using the same lightweight compressor applied to the segments.

Each entry in the ltDict and the vDict's second level also has a list of pointers to segments that contain the particular log type or variable value. CLP is I/O bound reading segments, so this serves the purpose of a coarse-grained search index. We index at the granularity of segments since any query that has a hit in a segment requires the segment to be decompressed from its beginning to the matched message. Without the index, any search that matched a dictionary entry required searching all segments in the archive.

For each archive, CLP also stores metadata about the log files and directories that were compressed. For each file, the metadata contains the original filesystem path of the file, the number of log messages it contains, the starting and ending timestamp of the messages in the file, the format of its timestamp (used to reconstruct the timestamp during decompression), and the segment that contains the compressed messages from the log file. In addition, the metadata contains the three offsets in the segment corresponding to the starting locations of the messages in this log file: one for each of the timestamp column, log type column, and variable column. These offsets are used to speedup the search when users use search filters. For example, a user could search for filenames that match a specific pattern, such as `yarn.log` (the log produced by YARN in a Hadoop cluster). Users can also specify the time range of the search, so CLP will first filter log files based on the starting and ending timestamps. In such cases, the metadata as well as the content in the data columns themselves allow CLP to skip scanning parts of data columns or files.

For directories, the metadata in the archive stores the paths of any empty directory that was compressed. An empty directory may be indicative of missing logs or it may be named after an identifier that the user wishes to keep. Thus, to ensure lossless decompression, these paths must be stored.

CLP also supports different compression modes that can offer improved compression at the cost of a minor reduction in performance. This is achieved by changing the lightweight compressor's settings. CLP currently ships with three modes: "Default" that uses Zstandard at level 3 and is concurrently optimized for compression speed and search performance; "Archive" that uses 7z-lzma at level 1 and offers higher compression with slightly reduced search performance; and finally, "Ultra" that uses 7z-lzma at level 9 and offers even higher compression with further reduced search performance. CLP can migrate between these modes by simply decompressing and recompressing the segment.

2.2 Search

Given a search phrase, CLP processes it in the same way that it compresses a log string: CLP tokenizes the phrase, extracts variable values, and constructs the log type. Next, CLP searches the archive's dictionaries for the log type and

#	Log type	Variables	CLP's processing
1	"Task * assigned to container*:\x11\x01"	"172.128*" (IP address)	Log type, var. search, scan segments
2	"Task * assigned to container*:\x12?"	"172.128*" (float num.)	Log type search (no match)
3	"Task * assigned to container*:178.128*"	-	Log type search (no match)
4	"Task * assigned to \x11\x02*:\x11\x01" "172.128*" (IP address)	"container*" (container ID)	Log type search (no match)
5	"Task * assigned to \x11\x02*:\x12?" "172.128*" (floating point num.)	"container*" (container ID)	Log type search (no match)
6	"Task * assigned to \x11\x02*:178.128*"	"container*" (container ID)	Log type search (no match)

Table 1: Processing of the search example in Figure 2. Each row is a sub-query generated by CLP.

dictionary variables. If matches are found for the log type and all dictionary variables, CLP proceeds to search the segments for encoded messages that contain the matching log type ID, dictionary variable IDs, and encoded non-dictionary variables.

However, wildcards in the search phrase complicate this process. CLP supports search phrases that can contain two types of wildcard characters: '*' (henceforth referred to as a *-card) that can match zero or more characters and '?' (henceforth referred to as as a ?-card) that matches any single character. First, it is nontrivial to tokenize a string with wildcards. For example, the string "Task*assigned" could be a single token or two tokens ("Task*" and "*assigned") since a *-card can match both non-delimiter and delimiter characters. Furthermore, it is nontrivial to determine if a token with a wildcard matches a variable schema. Finally, without wildcards, a token will be unambiguously categorized as either a log type, a dictionary variable, or a non-dictionary variable; but with wildcards, a token could belong to multiple categories. We address the first two issues in Section 3 and continue a discussion of the third challenge below.

2.2.1 Handling Ambiguous Tokens

Consider the search command in Figure 2. CLP first inserts a *-card at the beginning and end of the search string, turning it into a substring search to match user-expectations. Then after tokenization, CLP recognizes the following tokens: "*Task", "assigned", "to", "container*", "172.128*". Note that CLP does not consider a lone *-card as a token. For example, the *-card after "Task" is not treated as a token.

Each token is then compared against all of the variable schemas. CLP determines that "*Task", "assigned", and "to" do not match any schemas, hence they are part of the log type. Ambiguity exists for the other two tokens: "172.128*" could match an IP address schema or a floating point number, "container*" could match a container ID, and both could also be part of the log type. This creates a total of six combinations, and CLP generates a sub-query for each possibility.

Table 1 lists the six generated sub-queries. The first three treat "container*" as part of the log type, and "172.128*" is treated as part of an IP address, a floating point number, and the log type in sub-query 1, 2, and 3 respectively. When treating "172.128*" as a floating point number, CLP does not know the value's exact precision, so it inserts a ?-card to match all possibilities. Sub-query 4–6 in Table 1 consider the cases that "container*" is treated as a dictionary variable container ID.

Each sub-query will be processed in three steps. First, CLP searches the ltDict for matching log type. Only when there is a matching log type, it proceeds to the next step of searching the vDict for the dictionary variables. And only when there is at least one matching result for every dictionary variable, CLP proceeds to the third step. It takes the intersection of the segment indexes of the matching log type and dictionary variables, and for each of these segments, CLP decompresses the segment and searches for encoded messages matching the encoded query. If any of the first two steps return no matching result, or the intersection of the segment indexes is empty, the sub-query processing returns with no match. Different sub-queries will be processed in parallel.

For the six sub-queries shown in Table 1, only the first sub-query will exercise all three steps and return the matching log message shown in Figure 3. The processing of the other five sub-queries will return after step one, because the generated log type does not match any log types in the ltDict (these log types are impossible).

2.2.2 Optimizing CLP Queries

The way users write their search phrase can significantly affect the speed of the search. In our evaluation, dictionary search time is negligible compared to a segment scan; furthermore, log type dictionary search time is negligible compared with variable dictionary search. Therefore the best practice is to provide enough information in the search phrase to help CLP narrow down the log type or the dictionary variable values or both. The user can also speedup the search by filtering for a specific time range or log file path, dramatically shrinking the search scope. For example, the user could use a search phrase "172.128" to perform the previous query, but the search performance may be much worse. CLP will determine that "172.128" could be part of an IP address, floating point number, or the log type, and generate three sub-queries with log types being "\x11\x01", "\x12?", and "172.128". However, the first two sub-queries will likely result in numerous matching log

types in practice, i.e., any log type that contains an IP address or a floating point number, so CLP will end-up scanning a large number of segments.

Currently CLP does not use any additional index on its dictionary entries. A search on the ltDict, for example, will sequentially scan each entry. This is not a problem for now as the bottleneck in search is in scanning the segments, because the dictionaries are small. CLP also does not have any index on non-dictionary variables. We plan to add index (e.g., B-trees) to non-dictionary variables in the near future.

2.3 Handling Special Cases

Users have two options for changing variable schemas after log data has already been compressed. The new schema can be applied only to newly compressed data, in which case each archive will also need to retain the schemas that were used to compress the data. Alternatively, the users can ask CLP to update existing archives to use the new schema, and CLP will have to decompress and recompress the data.

CLP can also warn the user if the schemas they provided are not optimal. For example, if the user forgot to specify the schema of a variable, that variable would be encoded as part of the log type, and could "pollute" the log type dictionary where a large number of similar log types are created, with the only difference being that variable value. CLP can detect this case by comparing the edit distance between log types and issue a warning.

Although rarely used, CLP also supports the deletion of log messages. The encoded messages will be deleted from the affected segments, which involves recompressing the segment data using the general-purpose compressor and writing it to disk. The segment index in the dictionaries will also need to be updated.

Currently CLP does not support SQL-style join operations in a single query. However, users can perform joins in their client program using CLP's APIs.

2.4 Distributed Architecture

CLP adopts a simple controller and data node design that enables high scalability, similar to other widely used big data systems [9, 11, 22, 41]. The central controller simply manages metadata and node failures, while the data intensive computation of compression and search as described above are performed by each data node independently. Compressed data is stored on a distributed filesystem to ensure reliability.

The controller maintains three metadata tables: 1) log files, 2) archives, and 3) empty directories. The log files table stores the metadata of each raw log file (its file system path, the number of log messages, etc.) as well as the archive that contains this log file. Note that if the log messages are directly streamed to CLP using a log aggregation tool (e.g., rsyslog, Fluentd, Logstash, etc.), CLP still splits them into logical files once the buffered log messages reach a certain size or time frame. The archives table stores the metadata of each archive including which data node stores this archive. The empty directories table stores the paths of empty directories compressed in each archive.

The purpose of these metadata tables is only to speedup the search. For example, a user can specify a filter to only search log files whose file names match a certain pattern. The information stored in these tables is also stored in the archives, so even if the tables are lost, there is no risk of data loss. Nevertheless, we replicate the metadata tables three times with failover handling.

In order for CLP's compression and search to scale in a distributed system, each archive is independent of other archives and immutable once written. This independence makes compression easily parallelizable without any synchronization between threads writing different archives. The immutability ensures that a search thread can query an archive without synchronizing with a compression thread. To avoid coordination between search threads, each archive is only queried by a single thread for a given query. Thus, CLP parallelizes compression and search at the granularity of individual archives.

File System Integration Using FUSE. CLP has the ability to transparently integrate with a user's existing environment. For example, a user can use GNU `find` to search for files, and use `VIM` to open a compressed log file. We implement this by intercepting the file system operations using FUSE (Filesystem in Userspace) [44]. It walks the directory hierarchy stored in the log files table and decompresses the required data on demand to satisfy I/O requests. Common I/O optimizations such as caching, I/O request re-ordering and batching are performed to further increase CLP's efficiency and performance.

3 Wildcards and Schemas

We face two fundamental challenges in handling wildcards. Recall that CLP's encoding process requires tokenizing the input, extracting each token that matches a variable schema, and finally composing the log type. The first challenge in handling wildcards is determining how to tokenize a string containing wildcards, given that a wildcard could either match a delimiter or non-delimiter character. The second is determining if a token containing wildcards (*a wildcard token*) *could* match a given variable schema. Both of these challenges occur because a wildcard string has a range of possible inputs that it could match, and CLP's task is to encode all possible inputs so that they can be used for search.

3.1 Wildcard String Tokenization

To tokenize a wildcard search string, we need to consider each possible interpretation of every wildcard in the string.

#	*-card interpretation	Spans
1	Delimiters only	"*to", "*", "container*"
2	Non-delimiters only	"*to*container*"
3	Both	"*to*", "*", "*container*"

Table 2: The spans generated by tokenizing "*to*container*" depending on the interpretation of the central *-card.

For example, consider the search string "*to?container*". If the ?-card is interpreted as a delimiter, the string will generate three *spans*: "*to", "?", and "container*". We use the term span to refer to either a contiguous set of non-delimiter characters, or a contiguous set of delimiter characters. Using the schemas in Figure 4 on these spans, CLP will find that the last span matches a variable schema and the rest match the log type in Figure 3. However, if the ?-card is interpreted as a non-delimiter, then the entire string will be treated as a single token and CLP will find neither a matching schema nor log type. Accordingly, CLP must generate sub-queries from each unique tokenization of the search string.

To handle *-cards, CLP technically needs to consider that a *-card can be interpreted as either 1) matching non-delimiters only, 2) matching delimiters only, or 3) matching non-delimiters and delimiters. However, because a *-card matches *zero* or more characters, we can skip a case.

Consider the search string "*to*container*". To simplify the discussion, we only consider the interpretation of the central *-card and assume the others are interpreted as non-delimiters only. Table 2 lists the spans generated for each case. Note that the third tokenization is from interpreting the *-card as zero or more non-delimiters, followed by zero or more non-delimiters and delimiters, followed by zero or more non-delimiters. Comparing the first and third tokenization, we can see that the third is a more general version of the first. As a result, CLP does not need to consider the first tokenization. We can generalize this as follows: If a *-card is interpreted to have a different type than either of the characters surrounding it, the tokenization should split the string at the *-card while leaving *-cards attached to the surrounding character.

3.2 Comparing Expressions

To compare a wildcard token to a variable schema, CLP needs to determine if they overlap in the words that they could match. More formally, let U represent the words matched by the wildcard-containing token, and V represent the words matched by the variable schema. CLP needs to determine if $U \bigcap V \neq \emptyset$. For example, the wildcard token, "task_?", and the variable schema, "task_\d+" both match task IDs with one digit. Therefore, CLP can consider that this token matches the schema. However, this intersection does not imply that $U = V$, so CLP must still consider that "task_?" may be part of the log type (e.g., if the ?-card matches an alphabet). To determine if $U = V$, CLP could verify that $U \bigcap V^c = \emptyset$, where V^c is the set complement of V, but we find that this is rarely true in practice.

This is a standard problem of comparing the accepted input sets of two regular expressions. However, modern regular expression engines support *irregular* expressions (e.g., back-references) that prevent them from supporting this standard operation. Furthermore, we could not find a widely-used engine that supported strictly regular languages. So we built our own engine and use it to compute this intersection as well as enforce rules on the supported variable schemas.

4 Schema Design

Up until this point, we have only discussed schemas that match a single token. However, there are several variable values that fall outside this definition. For example, using the delimiters in Figure 4, the variable value "0.0.0.0:80" is a set of two tokens ("0.0.0.0" and "80") joined by a delimiter (':'). Similarly, " block id : 1073741827 " is a variable value that can only be categorized as a block ID if the schema takes into account the tokens before the actual variable value. To handle these cases, we extend our definition of a schema to include multiple regular expressions.

A schema in CLP is a sequence of regular expressions, where each expression exclusively contains non-delimiters or delimiters; we refer to the former as a *token expression* and the latter as a *delimiter expression*. In addition, the sequence must alternate between token and delimiter expressions, or else the tokenization rule could be violated. Finally, a schema may include non-capturing prefix and suffix expressions that are used to contextualize the schema.

CLP ships with a few default schemas that we have found are effective in capturing most variables. Specifically, we have a schema each for non-dictionary integer and floating point values. In addition, we have a schemas that match any token with a digit or any token preceded by an equals sign. Finally, we treat most non-alphanumeric characters as delimiters except for a few like underscores and periods.

5 Compressed Persistent Caching

Our experience with CLP shows that it is typically bottlenecked by I/O. Although, the dictionaries and segment index help to avoid much of this I/O, queries that match rare log types can still end up reading an entire segment. Thus, we designed a caching mechanism to improve the performance of these queries.

Consider a segment with two log types: *ltA* comprising 90% of messages in the segment and *ltB* comprising 10% of messages. A query for either log type without applying any filtering requires reading the entire segment since *ltA* and *ltB*'s messages are interspersed. A query for *ltB* would read

90% more data (i.e., those belonging to *ltA*) than necessary, and a query for *ltA* would read 10% unnecessary data.

One possible solution is to sort the log types in each segment, but this introduces two problems. First, since the segment is compressed as a single stream, if a log type's messages start in the middle of the segment, queries for that log type will require decompressing all messages before it. Second, since messages are no longer ordered as in the raw log file, each message would also need to store its position in the original log file so that we could maintain lossless decompression.

Another solution is to store each log type in its own segment, but this too introduces complications: For example, compression performance will be decreased since CLP will have to repeatedly open and close several segments, one for each log type in a file (in reality, the number of such single-log-type segments may exceed the number of open files per process that the OS allows, preventing CLP from keeping the files open at the same time). As a result, we do not use this strategy as our primary storage method but rather as a cache.

CLP's policy is to cache recently read, *infrequent* log types by storing each log type in its own segment. These segments are created in addition to the existing segments compressed by CLP, instead of replacing them. Specifically, when a user runs a query containing one or more log types, CLP will attempt to cache messages with those log types (henceforth referred to as caching log types) if the query does not return too many messages. The specific number of messages is configurable and will depend on the user's performance requirements and system resources. Only infrequent log types are cached because they offer both the best speedup and least additional storage cost.

When CLP tries to cache a new log type and the cache is full, it will need to decide whether to evict an existing log type or discard the new log type. Its policy is to evict log types that 1) have not been recently queried, and 2) that contain more messages than the new log type to be cached. The first condition is necessary to ensure that the cache does not eventually become filled with the most infrequent log types due to the second condition. The duration that is considered recent can also be configured by the user and again depends on their deployment. Note that in practice, a user's query may match multiple log types, in which case, CLP creates a persistent cache file for each log type independently.

The format of each log type segment in the cache is similar to the regular segments, but with a few key differences. First, there is no log type column since the entire file has the same log type. Second, each message additionally includes a log file path identifier, a timestamp format identifier and an optional message number. These identifiers are necessary since messages in this file may come from many different log files. Finally, the log type segment is named in a way that it can be easily referenced using the corresponding log type ID.

With the cache enabled, CLP processes a query in two parts: one for the log type cache and one for the non-cache segments. To determine which log types are in the cache, CLP simply uses each log type's ID to locate its corresponding segment. If one exists, it is searched like any other segment and the log type is removed from the query. Then, any remaining uncached log types are searched for in the non-cache segments, completing the query.

Name	Files	Log Messages	Size (GB)
/var/log/*-7GB	9,335	63,197,765	7
OpenStack-33GB	810	74,188,154	33
Apache-6TB	5,293	26,135,489,184	6,304
Hadoop-14TB	18,170	57,323,941,112	14,510

Table 3: The log datasets used to evaluate CLP.

6 Data Scrubbing and Obfuscation

A useful feature of CLP's design is the ability to quickly obfuscate data (e.g., to comply with data privacy laws) using the compression dictionaries. Consider a case where a user wants to obfuscate a username, "johnsmart9", from all log messages. Since this username will be stored in the variable dictionary, it can be easily replaced with an obfuscated string like "x93n4f9". Similarly, if a user wanted to hide all usernames from a certain log type, they could simply modify the log type in the dictionary to contain a generic username in place of the actual username. Moreover, since the dictionaries are typically much smaller than the segments or the raw data, these replacement operations will be much faster than they would be if the logs were not compressed.

7 Evaluation

CLP has been used to compress petabytes of logs from hundreds of different applications, and we have verified that its compression is lossless in all cases. Our evaluation focuses on CLP's performance. Specifically, we explore: 1) CLP's compression ratio and speed; 2) CLP's search performance; and 3) CLP's scalability and resource efficiency.

7.1 Experiment Setup

Table 3 shows the log corpuses used in our evaluation. The /var/log/* corpus contains all of the logs in the `/var/log/` directory generated by a cluster of more than 30 Linux servers over the past six years. The OpenStack-33GB log set was gathered by running the cloud scalability benchmark tool, Rally [45], on top of OpenStack. Apache-6TB contains Apache httpd access logs collected over a 15-year period by the U.S. Securities and Exchange Commission's EDGAR system [43]. The Hadoop-14TB logs were generated by three Hadoop clusters, each containing 48 data nodes, running workloads from the HiBench Benchmark Suite [25] for a month.

Note that the datasets generated by benchmarking tools may be artificially uniform, as benchmarks do not always capture the randomness of real-world deployments. However, this should not affect our claims since we compare CLP relative to other tools on the same datasets.

Experiments were performed on a cluster of 16 Linux servers connected over a 10GbE network, each with an eight-core Intel Xeon E5-2630v3 processor and 128GB of DDR4 memory. Unless otherwise specified, all data is stored on a 3TB (labelled 3TB, real-size 2.73TB) 7200RPM SATA HDD connected to each machine.

We compare CLP with Gzip 1.6, Zstandard 1.3.3, and 7z 16.02 for compression, in addition to ripgrep 12.1.0, Elasticsearch 7.8.0 and Splunk Enterprise 8.0.3 for search. All versions were the latest releases from Ubuntu 18.04's package manager, Elastic, and Splunk at the time of the experiments.

We use ripgrep to search the archives produced by general-purpose compressors. ripgrep is a grep-derivative designed with aggressive system and algorithmic optimizations that allow it to outperform grep significantly. Moreover, ripgrep offers advanced parallelization and can directly search the contents of Gzip, Zstandard, and 7z-lzma archives.

We modified Elasticsearch and Splunk Enterprise's default configuration only enough to ensure they matched CLP's search capabilities without storing more data. In practice, we expect a user in need of CLP's capabilities would do the same. Recall CLP can perform wildcard searches on log messages as well as filtering based on file paths and time ranges. We do not explicitly evaluate the filtering features but supporting them increases the amount of data that Elasticsearch and Splunk Enterprise store in their indexes.

Splunk Enterprise's default configuration matches almost all of CLP's capabilities with the exception of wildcard searches. Due to the way Splunk Enterprise indexes tokens with punctuation like "`AA-BB-123`", it cannot perform queries with wildcards in the middle of the token like "`AA*23`" [38].

For Elasticsearch, we first had to configure an index before logs could be ingested. Typically, Elasticsearch's ingestion tool, Filebeat, configures a default index; but because Filebeat was not fast enough for our use, we ingested logs using our own parser. Elasticsearch indexes are configured with a set of fields, each of which has a type indicating how it should be indexed. Elasticsearch only supports wildcard searches on fields with type "keyword". Alternatively, Elasticsearch can perform full text searches on "text" type fields, but this does not match CLP's capabilities. For example, Elasticsearch's default tokenizer ignores stop words like "and", whereas CLP's wildcard search does not. We initially tested indexing the content of each log message as a keyword-field but found that this required 58% more storage, took 7% longer to ingest, and was 4750% slower to search compared to indexing the content as a text-field. Elastic also recommends indexing unstructured content as a text-field [13]. Thus, we configured Elasticsearch's index with three fields: message_content with type "text," timestamp with type "date," and file_path with type "keyword." Following Elasticsearch's best practices [12], we set the max heap size to 30GB for its Java Virtual Machine.

For CLP, we configured the persistent cache to store less than 0.01% of all compressed messages and used the general-purpose default schemas to parse the logs in all experiments.

7.2 Compression Speed and Ratio

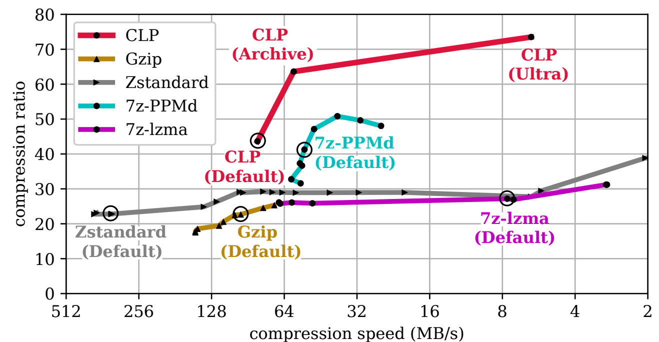

Figure 6: Compression ratio and speed trade-off for CLP and general-purpose compressors. CLP's compression generally exceeds all other compressors and its current speed is competitive.

We first examine CLP's tradeoff between compression ratio and speed compared to general-purpose compressors. Each tool was used to compress a 30GB subset of the Hadoop corpus. In addition, all data was read from and written to a tempfs RAM disk in order to minimize I/O overhead and fully expose the tools' algorithmic performance. For each tool, we measured its single-threaded compression speed since not all tools support multiple threads; for those that do, we observed a minor decrease in per-core performance when running them with multiple threads rather than independent processes. Finally, we vary each tool's compression level from low to high.

Figure 6 shows the compression ratio and speed for the evaluated tools. Overall, CLP achieves higher compression than Gzip or Zstandard. Compared to PPMd, a natural-language optimized compressor, CLP slightly exceeds its compression at their default levels and significantly exceeds it at higher compression levels. In addition, CLP's default level offers performance competitive with Gzip's default level but with double the compression. We use CLP's default mode for all remaining experiments.

To compare CLP's compression speed with Elasticsearch and Splunk Enterprise's ingestion speed, we reuse the previous experiment except each tool is configured to use the number of threads that provides the highest possible throughput. In contrast with general-purpose compressors, Elasticsearch and Splunk Enterprise are designed as multithreaded tools, so their performance generally suffers when they are forced to use a single thread. Figure 7 shows the results: Overall,

#	Query	# results	# log types	# dict. vars.
Log type queries contain no variables, so CLP only searches the log type dictionary and log type columns.				
Q1	" `org.apache.hadoop.hdfs.server.common.Storage:`↩ `Analyzing storage directories for bpid` "	12	1	0
Q2	" `org.apache.hadoop.hdfs.server.datanode.DataNode:`↩ `DataTransfer, at` "	2,026	1	0
Q3	" `INFO org.apache.hadoop.yarn.server.nodemanager.`↩ `containermanager.container.ContainerImpl: Container` "	513,893	12	0
Q4	" `DEBUG org.apache.hadoop.mapred.ShuffleHandler:`↩ `verifying request. enc_str=`"	810,033	84,922	0
Non-dictionary integer queries contain an integer non-dictionary variable, so the variable column is searched in addition to the log type search.				
Q5	" `to pid 21177 as user` "	12	3	0
Q6	" `10000 reply:` "	13,064	24	0
Q7	" `10 reply:` "	279,284	24	0
Non-dictionary float queries: contain a float non-dictionary variable.				
Q8	" `178.2 MB` "	2,800	3	0
Q9	" `1.9 GB` "	1,623,002	5	0
Dictionary variable queries contain dictionary variable, so log type dict., variable dict., and variable columns are searched.				
Q10	"`job_1528179349176_24837`"	51	89,258	3
Q11	"`blk_1075089282_1348458`"	4,261	89,258	3
Q12	"`hdfs://master:8200/HiBench/Bayes/temp/worddict`"	178,076	9	1
Non-matching query: contains a potential log type but does not match any log type.				
Q13	" `abcde` "	0	0	0

Table 4: The queries used in our search-performance evaluation, grouped based on how CLP processes them. The quotation marks in each query are used to highlight any leading or trailing spaces and are not part of the query. Similarly, the ↩ symbol indicates a newline that is not part of the query but was inserted for typesetting.

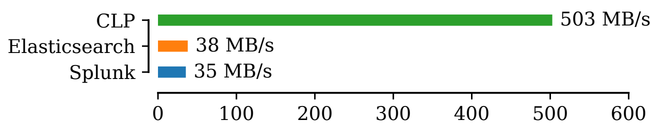

Figure 7: Single-node ingestion speed of 30GB of Hadoop logs for CLP, Elasticsearch, and Splunk Enterprise. CLP far exceeds their ingestion speed.

CLP is able to ingest the corpus at least an order of magnitude faster than both Elasticsearch and Splunk Enterprise.[2]

We also compare the tools' compression on larger datasets. To measure Elasticsearch and Splunk Enterprise's compression ratio, we shutdown the tools to ensure any in-memory data was persisted and then measured the size of their data directory on disk. For Splunk Enterprise, we used a subset of the terabyte-scale datasets since our evaluation license limited the amount of data we could ingest per day. Figure 8 shows the results for Gzip, Zstandard, and 7z-lzma using their default settings in addition to Elasticsearch, Splunk Enterprise, and CLP. For all corpuses, CLP significantly outperformed all of the evaluated tools. On average, using the default compression mode without customized parsing rules, CLP's average compression ratio is 32. CLP's advantage is evident on the

[2]Elasticsearch's 38 MB/s ingestion speed can only be achieved when we replaced its own log parsers (Logstash and Filebeat) with CLP's, as they were unable to ingest faster than 1 MB/s. We ported CLP's log parser to connect to Elasticsearch's REST API endpoint.

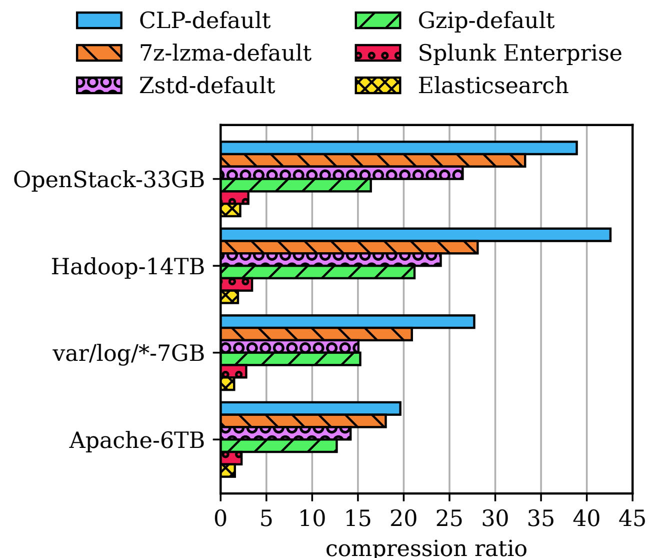

Figure 8: Compression ratio of tools on different corpuses. CLP exceeds the ratio of the others.

OpenStack and Hadoop datasets where the log formats contain a large amount of unstructured natural language. The Apache-6TB corpus has the worst compression ratio since the messages largely contain variable values.

In contrast, log indexing tools Elasticsearch and Splunk Enterprise have significantly lower compression ratios at 1.75 and 2.86 respectively. This means that their on-disk data structures, including both the index and the compressed logs, are on the same order of magnitude as the uncompressed log (57% and 35% respectively). (Both tools recommend users keep searchable data structures on fast storage such as an SSD.)

On average, across all experiments, CLP's log type dictionary accounted for 0.03% of the total compressed size and the variable dictionary accounted for 1.07%. All other CLP metadata files were negligible in size.

7.3 Search Performance

Commonly used log search tools fall into two categories: index-based search (e.g., Splunk Enterprise and Elasticsearch) and sequential search (i.e., variations of the grep tool). In comparison, CLP is a mixture. Its dictionaries serve the purpose of lightweight indexing (with the key difference being that CLP's dictionaries *deduplicate* repetitive data instead of duplicating data into a separate index), and when combined with the segment index as well as each file's metadata, CLP can skip files or jump to a specific file in a specific segment. On the other hand, CLP still searches columns sequentially. Thus, we compare CLP with tools from both categories.

We benchmark each tool using the set of queries in Table 4, specific to a 258GB subset of the Hadoop-14TB corpus. In designing the set of queries, we initially tried to make them representative, but faced two challenges. First, real-world datasets and workloads are diverse, meaning we would need a large number of queries to sufficiently represent most use cases. Second, any query set will likely be biased towards or against a tool, and so the benchmark would neglect the strengths and weaknesses of some tools over others. Instead, we designed the queries simply to test CLP by exercising its different execution paths, highlighting its strengths and weaknesses. For each query type, we used multiple queries that differ in the number of results they return from a few to many.

We used a 258GB subset of the Hadoop-14TB corpus since we were limited to one node for several of the evaluated tools. Specifically, our Splunk Enterprise evaluation license does not support distributed searches and ripgrep is a single-node tool. Conversely, we could not evaluate the full corpus on one node for Elasticsearch and Splunk Enterprise since they require more storage than the size of the hard drive attached to each machine.

In designing each query, we also had to ensure that all tools would return the same result set for each query. As explained in §7.1, Elasticsearch does not support precise substring searches on text-fields because it indexes a message by ignoring elements like punctuation. So an Elasticsearch query that includes punctuation may return results which both include and do not include the punctuation. As a result, we only chose queries where differences in interpretation did not affect the results returned. Similarly, because Splunk Enterprise and Elasticsearch cannot accurately support wildcard searches (§7.1), the queries do not explicitly contain wildcards. (CLP's wildcard handling is still exercised as it implicitly adds wildcards to the beginning and the end of each query.)

We ran each query 10 times and report the average of all runs. To emulate searching cold data stored on low-cost storage (hard drives or network storage), the file system page cache is cleared and the tools are restarted (to ensure inmemory caches are cleared) before each run. We also ran each tool with a varying number of threads, and report the configuration that yielded the fastest completion time.

Figure 9 shows the search performance of CLP and the other tools. The averaged normalized completion times of CLP, Elasticsearch, and Splunk Enterprise, are 1x, 1.3x, 4.2x respectively, hence CLP outperforms both. In addition, CLP is faster for queries that return a lot of results (i.e., Q3, Q4, Q7, Q9, and Q12), and competitive for queries that return few results. In queries where CLP is slower, Elasticsearch performed 6–22x less I/O, suggesting its gains are as a result of using search indexes.

Figure 9 also shows CLP's performance when a search is served from its persistent cache. To evaluate CLP's persistent caching, we ran each query twice—once to build the cache, and again to evaluate its performance with the cache. The cache was purged between queries to ensure the next query was not affected by prior caching. The six queries which were persistently cached (Q1–Q5 and Q12) received an average speedup of 43x and a median speedup of 8.64x. Two of those six queries which were previously 4x slower than Elasticsearch are now 5x and 51x faster than Elasticsearch, respectively. Under this configuration, CLP is faster than both Splunk Enterprise and Elasticsearch in every persistently cached query. This shows that the persistent cache can make CLP even more competitive with a negligible effect on compression ratio.

Splunk Enterprise and Elasticsearch also have caching mechanisms but they provide different functionality than CLP. In particular, Elasticsearch and Splunk Enterprise use the entire query (query phrase, timestamp filter, and so on) as the cache key, so only an identically repeated query benefits from the cache. In contrast, CLP's cache key is a log type, so new queries can benefit from the cache if they encompass a cached log type. Also, Elasticsearch and Splunk Enterprise's caches are not persistent.

Finally, Figure 9 shows that CLP is able to exceed the performance of every ripgrep-compressor combination for every query. Analyzing the machine's usage shows that Zstandard is being bottlenecked by disk I/O while both 7z-lzma and Gzip are bottlenecked by the CPU.

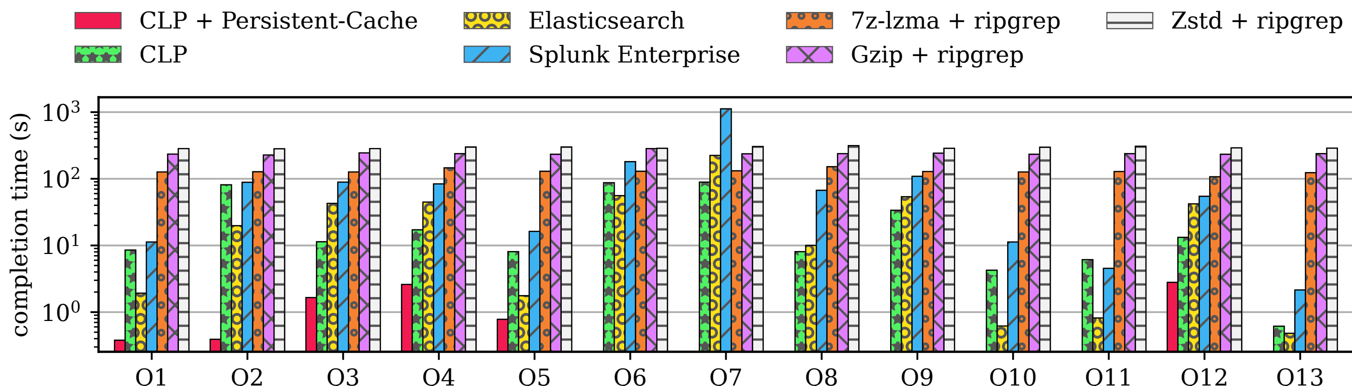

Figure 9: Search performance of CLP, Elasticsearch, Splunk Enterprise, and popular compressed sequential search combinations. CLP is faster for longer queries and competitive for shorter queries. CLP's cache greatly improves its competitiveness.

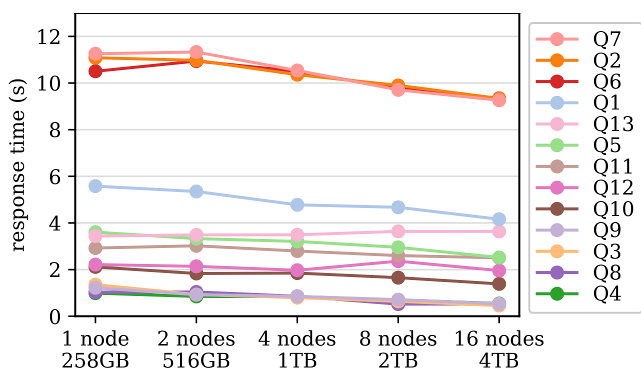

Figure 10: Response time of queries for CLP when both data and resources were horizontally scaled from 1 to 16 nodes.

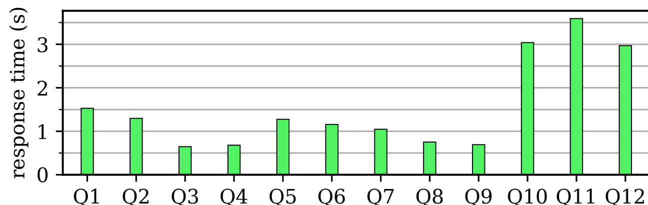

Figure 11: CLP's query response time on a petabyte log corpus. The response time continues to exhibit the pattern observed in the horizontal scalability experiment.

7.4 Horizontal, Vertical, and Capacity Scaling

Since CLP's archives are designed to be independent (§2.4), compression and search are embarrassingly parallelizable tasks. Figure 10 shows that as we scale horizontally, adding more nodes that contain an equal amount of data, CLP's search response time stays nearly constant. Response time is measured from when a query is entered to when the first matching result is returned (in the case of no matching result, it is the query completion time). We show response time instead of completion time because 1) when the output is large, the completion time will be bottlenecked by how fast the user's client can receive results, and 2) in those scenarios, users typically search and refine their queries before arriving at a small amount of output. Nevertheless, completion times also stay nearly constant except for Q3, Q4, Q9, and Q12, whose completion time grows linearly with the output size.

We also repeated the previous experiment with a petabyte of data to evaluate CLP at the scale of logs produced by large internet companies. Since the 3TB hard drives attached to each machine did not have enough free space to store the data, the archives were instead stored on a distributed file system (MooseFS [10]) running on commodity hard drives. The results in Figure 11 show CLP still maintains low response time, but the ordering (by response time) of queries differs from the the previous experiment. This is because CLP was I/O-bound in the previous experiment whereas in the current experiment, MooseFS parallelizes I/O requests across multiple drives, so CLP becomes more CPU-bound. We omitted Q13 from the figure; its response time is 140 seconds. Q13 represents a worst case for CLP as its response time is the same as the completion time, because the search returns "no result." It took 140s to search all log type dictionaries of over 61,000 archives with only 256 threads. In contrast, the previous experiment had one search thread per archive. Overall, the results show that CLP can indeed scale in large Internet companies, while reducing storage costs.

Using MooseFS, we also measured CLP's ingestion speed at the petabyte scale. By adding eight additional nodes to the existing 16-node cluster, CLP was able to reach an ingestion speed of about one petabyte of raw logs per day, exhausting each hard drive's bandwidth.

To evaluate vertical scalability, we tested CLP's search performance on the Hadoop-14TB corpus with *a single thread on a single data-node*. The fastest completion time was for a non-existing result query which took just under a minute, and most queries started emitting results within 10 seconds.

8 Related Work

We discuss three categories of related work: (1) log compression, (2) searching the compressed form of general-purpose indexes for textual data, and (3) log search. Existing log compression tools do not enable search (on compressed data). Singh and Shivanna's [35] method of log compression also aims to deduplicate static text from variable values. They rely on the applications' source code to generate patterns, akin to our concept of a log type. Variables are annotated with the variable's primitive type such as integer or long so they can be encoded into binary bits stored separately. However, they do not propose any search algorithms on the compressed data. In addition, the compression in their work is not entirely lossless in the sense that a number's precision is not encoded. For example, the value "`1.000`" can only be stored as an equivalent floating point number, failing to take into account the number's zero padding.

Separating highly redundant static text from variable values has also been used to design highly efficient log printing libraries. For example, both NanoLog [47] and Log20 [48] only log an ID for each log type at runtime, and reconstruct the textual log message in post-execution phases. Furthermore, some logging systems [34] directly output binary log messages, representing each log type with an ID. While CLP is designed to compress text logs, its search algorithms can be used to search binary logs by associating human-readable static text with each log ID. Hence, users can use CLP's intuitive text search interface to analyze binary logs, as if they are text logs, with minimal storage overhead.

General-purpose text search typically uses indexes such as suffix trees or tries, which will *add* 10-20x the size to the original text data [6, 27]. Compressed forms of these indexes, typically via smart encoding, have been proposed [17–20, 24, 30–33] such that they can be searched without decompression. Succinct [6] further proposed an entropy-based representation of these compressed indexes to further reduce the size of compressed index. However, regardless of how small the index is, it still increases storage space instead of reducing it, and search can only be performed on data that is indexed. In comparison, CLP does not add any additional index; it simply deduplicates the static text and dictionary variables, whereas these works are used to compress indexes.

Several pattern matching algorithms exist for searching data compressed with general-purpose compressors, but none operates on data compressed using an algorithm that practically achieves our compression speed and ratio. Kida *et al.* [26] implemented an algorithm for pattern matching in LZW compressed data, achieving better performance than decompression followed by a search. Similarly, Navarro and Raffinot did the same for LZ78 [49] compressed data. However, LZW has worse compression than CLP while LZ78 uses a prohibitive amount of memory for large data sizes and is more likely to experience dictionary explosion.

Tools like Splunk Enterprise [4] and Elasticsearch [15] allow users to search and analyze logs. They work by treating logs as normal text files and apply standard indexing and search techniques to achieve responsive searches. In contrast, CLP does not need to spend expensive resources to create and maintain additional indexes.

Conversely, Scalyr [3] is a log search tool that uses a "brute-force" approach, instead of using any index. It uses a number of low-level optimizations to achieve a log search speed of up to 1.25 GB/second per core without using any index [28]. It also directly works on raw logs. In comparison, by working directly atop compressed log data, CLP is able to achieve much higher search performance when translated to the raw log size, even when the data is uncached and stored on low-cost HDDs. Through our scalability experiments, we observed that our fastest queries, which return no results purely by scanning through the log type dictionaries, can effectively search through the equivalent of hundreds of gigabytes per second per core from a single HDD.

Grafana Loki [23] makes a trade-off that lies between index-based search tools and Scalyr: it only indexes the labels, i.e., selected fields of the log. Hence the index size is significantly reduced, yet users can only search the labels. Moreover, the index again adds to the storage space.

9 Conclusion

This paper presented the mutually exclusive problem of log archiving and log analytics. We present an end-to-end solution, CLP, that allows users to perform "archivalytics" across their entire history of compressed log messages without the need for decompression. Using an algorithm customized to text logs, CLP is able to achieve higher compression ratio than other compressors while enabling faster search performance than index-based search tools.

Acknowledgements

We thank our shepherd Lalith Suresh and the anonymous reviewers for their insightful comments. Michael Stumm provided invaluable suggestions throughout the project. This research was supported by the Canada Research Chair fund, a Connaught Innovation Award, a Huawei grant, the McCharles fellowship, a Mitacs grant, NetApp fellowships, NSERC discovery grants, and a VMware gift.

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019.

[2] Elastic. https://www.elastic.co/, 2021.

[3] Scalyr. https://www.scalyr.com/, 2021.

[4] Splunk. https://www.splunk.com/, 2021.

[5] 104th United States Congress. Title 45 CFR 164.316. In *United States Code of Federal Regulations*. 2003.

[6] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 337–350. USENIX Association, May 2015.

[7] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 445–460. USENIX Association, October 2018.

[8] Brian Knox. Diving in The Deep End: Logging and Metrics at DigitalOcean. Video, November 2013. https://www.elastic.co/elasticon/tour/2015/new-york/logging-and-metrics-at-digital-ocean.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218. USENIX Association, November 2006.

[10] Core Technology Sp. z o.o. MooseFS, 2021. https://moosefs.com/products/#moosefs.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04. USENIX Association, December 2004.

[12] Elastic B.V. *Setting the Heap Size*, May 2021. https://www.elastic.co/guide/en/elasticsearch/reference/7.8/text.html.

[13] Elastic B.V. *Text Data Type*, May 2021. https://www.elastic.co/guide/en/elasticsearch/reference/7.8/text.html.

[14] Elastic N.V. Annual Report. https://www.sec.gov/ix?doc=/Archives/edgar/data/0001707753/000162828020009982/estc-20200430.htm, June 2020.

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

[16] Facebook, Inc. Zstandard. https://facebook.github.io/zstd/.

[17] Paolo Ferragina and Giovanni Manzini. Opportunistic Data Structures with Applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS 2000, pages 390–398. IEEE, November 2000.

[18] Paolo Ferragina and Giovanni Manzini. An Experimental Study of a Compressed Index. *Information Sciences*, 135(1-2):13–28, June 2001.

[19] Paolo Ferragina and Giovanni Manzini. An Experimental Study of an Opportunistic Index. In *Proceedings of the 12th Annual SIAM Symposium on Discrete Algorithms*, SODA '01, pages 269–278. ACM, January 2001.

[20] Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM (JACM)*, 52(4):552–581, July 2005.

[21] Free Software Foundation, Inc. GNU Gzip, August 2020. https://www.gnu.org/software/gzip/.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, October 2003.

[23] Grafana Labs. *Loki Documentation*, May 2021. https://grafana.com/docs/loki/latest/.

[24] Roberto Grossi and Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[25] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *Proceedings of the 26th International Conference on Data Engineering Workshops*, ICDEW 2010, pages 41–51. IEEE, March 2010.

[26] Takuya Kida, Masayuki Takeda, Ayumi Shinohara, Masamichi Miyazaki, and Setsuo Arikawa. Multiple Pattern Matching in LZW Compressed Text. In *Proceedings of the Data Compression Conference*, DCC '98, pages 103–112. IEEE, March 1998.

[27] Stefan Kurtz. Reducing the Space Requirement of Suffix Trees. *Software: Practice and Experience*, 29(13):1149–1171, November 1999.

[28] Steve Newman. Searching 1.5TB/Sec: Systems Engineering Before Algorithms, May 2014.

https://www.scalyr.com/blog/searching-1tb-sec-systems-engineering-before-algorithms/.

[29] Cloud Object Storage | Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/.

[30] Kunihiko Sadakane. Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array. In *Proceedings of the 11th International Conference on Algorithms and Computation*, ISAAC '00, pages 410–421. Springer, December 2000.

[31] Kunihiko Sadakane. Succinct Representations of LCP Information and Improvements in the Compressed Suffix Arrays. In *Proceedings of the 13th Annual SIAM Symposium on Discrete Algorithms*, SODA '02, pages 225–232. ACM, January 2002.

[32] Kunihiko Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, September 2003.

[33] Kunihiko Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 41(4):589—-607, December 2007.

[34] Kedar Sadekar. Scalable Logging and Tracking, June 2012. https://netflixtechblog.com/scalable-logging-and-tracking-882bde0ddca2.

[35] Pranay Singh and Srikanta Shivanna. Method and System for Compressing Logs. *US Patent 9,619,478*, April 2017.

[36] Splunk Inc. Annual Report. https://www.sec.gov/ix?doc=/Archives/edgar/data/0001353283/000135328320000008/a01312010k.htm, March 2020.

[37] Splunk Inc. Splunk® Enterprise 8.0.3, April 2020. https://www.splunk.com/en_us/download/previous-releases.html.

[38] Splunk Inc. *Wildcards*, February 2021. https://docs.splunk.com/Documentation/Splunk/8.0.3/Search/Wildcards.

[39] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553—564. ACM, August 2005.

[40] James A. Storer and Thomas G. Szymanski. Data Compression via Textual Substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

[41] The Apache Software Foundation. HDFS Architecture, July 2020. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[42] The Apache Software Foundation. Apache Lucene, 2021. https://lucene.apache.org/.

[43] The Division of Economic and Risk Analysis. EDGAR Log File Data Set, June 2017. https://www.sec.gov/dera/data/edgar-log-file-data-set.html.

[44] The Kernel Development Community. *FUSE*, May 2021. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[45] The OpenStack Foundation. Rally, 2021. https://opendev.org/openstack/rally.

[46] Vijay Samuel. Monitoring Anything and Everything with Beats at eBay. Video, February 2018. https://www.elastic.co/elasticon/conf/2018/sf/monitoring-anything-and-everything-with-beats-at-ebay.

[47] Stephen Yang, Seo Jin Park, and John Ousterhout. NanoLog: A Nanosecond Scale Logging System. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18, pages 335–350. USENIX Association, July 2018.

[48] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 565–581. ACM, October 2017.

[49] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.