Master Thesis

Parsing Log Data for Compression and Querying

In Partial Fulfillment of the Requirements for the Degree

Master of Science

Submitted to the Faculty of Mathematics, Natural Sciences and Computer Science at the University of Applied Sciences Mittelhessen

by

Simon Stockhause

May 30, 2023

Referent: Prof. Dr. Harald Ritz

Korreferent: Dr. Dennis Priefer

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. Gießen, May 30, 2023 Simon Stockhause

Logs are descriptions of system events documented and stored in textual form. They are vital to understanding system behavior by analyzing them. The increasing complexity of systems leads to a larger volume of log data. This, combined with faster development cycles and the widespread distribution of systems, poses significant challenges in efficiently processing and storing log data while minimizing computational and storage costs. This thesis conducts a systematic literature review investigating log parsing and compression techniques. A novel log parsing approach is proposed, enabling the detection of previously elusive tokens within log records. These tokens contribute to constructing regular expressions that aid in the log compression process. The effectiveness of the parsing approach is evaluated by comparing it with twelve other parsers using a benchmark. The evaluation reveals that the impact of parsing results on the compression ratio is minimal. Additionally, data compressed by specialized log compressors show positive characteristics not found in general–purpose compression techniques.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Objectives . . . 2
- 1.3 Delimitations . . . 3
- 1.4 Methodology . . . 4
- 1.5 Outline of the thesis . . . 4

2 Background 7
- 2.1 Data . . . 7
  - 2.1.1 Unstructured Data . . . 7
  - 2.1.2 Semi–Structured Data . . . 7
  - 2.1.3 Structured Data . . . 9
- 2.2 Log Data . . . 9
  - 2.2.1 Parsing . . . 12
  - 2.2.2 Compression . . . 13
- 2.3 Problem Solving with Log Data . . . 14

3 Concept 21
- 3.1 Identification of Parsing, Compression and Query Strategies . . . 21
  - 3.1.1 Motivation . . . 21
  - 3.1.2 Review Planning . . . 22
  - 3.1.3 Study Selection . . . 26
  - 3.1.4 Results . . . 26
  - 3.1.5 Discussion . . . 34
- 3.2 Design Method . . . 35
- 3.3 Problem Statement . . . 37
  - 3.3.1 From Messy to Meaningful: The Need of Parsing Log Data . . . 37
  - 3.3.2 The Variable Template Extraction Problems . . . 40
  - 3.3.3 Requirements Derivation . . . 40
- 3.4 Design for a Logparser and Regluar Expression Generator . . . 43
  - 3.4.1 Exploiting the Structure of Log Data . . . 43
  - 3.4.2 Parser Design . . . 54
  - 3.4.3 Regex Generator Design . . . 55

4 Implementation 57
4.1 Overview of the Implemented System 57
4.1.1 Variable Template Extraction Parser (VTEP) 57
4.1.2 Variable Template Extractor (VTE) 58
4.2 Configuration 58
4.3 Variable Template Extraction Parser 60
4.4 VTEP to VTE JSON Interface 67
4.5 Variable Template Extractor 68
4.6 Parser Benchmark 77

5 Evaluation 81
5.1 Analyzing Parsing and Performance of VTEP: Evaluating Accuracy, Efficiency, and Robustness 82
5.2 Analyzing the Impact of Parsing and Custom Regular Expressions on Log Compression 85
5.3 Investigating the Impact of Parsing and Compression on Log Data's Processability 88
5.3.1 Parsing 89
5.3.2 Compression 91
5.4 Evaluation of Research Methods 92

6 Conclusion 95
6.1 Validity 97
6.2 Related Work 99
6.3 Further Research 99
6.4 Perspective 100

Bibliography 101

List of Abbreviations 112

List of Figures 113

List of Tables 115

Listings 117

1 Introduction

Storing, processing, and transmitting data is the essence of computer science. Doing so efficiently requires a broad understanding of its structure and usage. The structure defines what can be done with data. The usage defines what should be done with data. Data parsing uses the structure in the context of its usage to provide value to the log analysis processing chain. An algorithm to process log data to tailor the output to the specific needs of the use case is the heart of *log data parsing*. However, there are many algorithms designed to do precisely that. Each has unique characteristics, such as performance, compression ratio, or used data structures. In this work, an analysis of state–of–the art log parsing techniques and selected use–cases, such as compression is conducted. A short analysis and evaluation of the approaches to log data parsing is conducted to provide an understanding of their usage, benefits, and limitations. Selected log data compressors are compared and a recommendation is provided.

1.1 Motivation

Log data has the potential to provide a lot of information about the system state and computed state changes. Log data refers to a collection of documented system or application events, similar to a diary for systems. A log record, a single diary entry, usually consist of a timestamp and arbitrary content:

```
1    2022-12-15 21:01:23 INFO User login: user@example.com
```

Listing 1.1: Simple log record

Logging from a software engineering perspective is omnipresent, and the practice of logging is deeply ingrained in developing software [1]. A systematic map study shows that the interest of practitioners and researchers in log data has been continuously increasing and that this trend began in the early 2000s [2]. Business–oriented data analysis to understand user and system behavior and recognize trends is becoming increasingly relevant. However, analyzing log data is becoming more challenging because of the increasing amount of data generated by large-scale systems. Large–scale systems' sheer amount of data is hard to manage. One case study investigated the cloud system

of Aliyun Mail, a service provided by Alibaba Cloud. The system handles over 20 million user requests daily, resulting in 30–50 gigabytes of sampled log data per hour [3]. Uber reports to have generated 200 terabytes of data per day by their spark cluster [4]. The growth in applications, transformation to cloud platforms, or new architecture, such as microservices, contribute to the diversification of log data sources, formats, and information content. This suggests that log data is a relevant source for advanced analytics or security considerations.

Yuan et el. stated the importance of log data generation in common open–source software [5]. Generating data needs effective data management, and storage is a critical aspect of managing log data. A survey suggests that 68% of the participants spend more than 30% of their IT–budget on data storage, backups, and recovery [6]. Uber reports 1.8$ million dollars of potential cost for log data storage alone [4]. These recent reports indicate an industrial interest in log file processing. Also, the academic interest in log data processing is given [7][8][3][9].

1.2 Research Objectives

As the practical relevance of log data is shown, handling it is the day–to–day business for many software engineers, data scientists, and decision–makers. The primary goal of this thesis consists of researching log analysis techniques and their approach to log data parsing, compression, and querying. One goal is to identify the benefits and limitations of applying them. Providing a summary of state–of–the–art log analysis techniques for the scientific community focusing on log parsing and querying techniques on compressed log data. Nevertheless, it aims to provide guidance to practitioners in choosing the appropriate techniques for their needs.

The objectives serve as a framework for addressing the research questions, outlining the approach and methodology used to investigate and provide answers to the research questions.

O1: Research Log Data Query and Compression Strategies

Collect log data query and compression strategies. This is done by conducting a Systematic Literature Review (SLR).

O2: Investigate Log Data Query and Compression Strategies

It should be clarified when they can and cannot be applied. It also should be described how they can be applied. This is done by conducting a SLR.

O3: Analyze the Performance Implications of Each Strategy

Because each strategy requires computation, it should be described what their performance implications are.

O4: Conduct Performance Comparison of the Presented Parser

Three metrics will be evaluated to compare the performance of the presented parser against other parsers used or potentially usable in log data query and compression strategies: accuracy, robustness, and efficiency. These metrics are well defined in the log parsing literature [10]. The results of this comparison will help evaluate the performance of the presented parser relative to other parsers identified in the SLR.

O5: Recommend Log Query and Compression Technology Based on the SLR

The ideal would be the identification of a strategy that applies broadly while minimizing additional architectural complexity, compute requirements, and storage space. A discussion is conducted and a recommendation is provided based on the discussion.

The defined goals lead to the following research questions:

R1: How can the structure of log data be exploited to reduce storage requirements through compressing while maintaining the ability to query the data?

R2: How do parsing and compressing log data affect its further processability?

R3: Is there a log data compression and query technology that practitioners can use that is widely applicable while minimizing additional architectural complexity, computational requirements, and storage space?

1.3 Delimitations

This thesis explores the use of log data techniques that combine compression and queryability to achieve the goal of recommending a technology. This research focuses solely on log data and does not consider other types of data, such as unstructured, semi-structured, or structured data. The parsing techniques used rely exclusively on log events as input and do not incorporate static code analysis to derive parsing rules. This limitation intentionally narrows the scope and concentrates on techniques that solely depend on log events. Additionally, this thesis does not delve into the application of log parsing results, such as log templates, in log analysis or its subfields like anomaly detection or root-cause analysis. Considering the vast number of log parsers available in the literature, only a subset is selected for comparative analysis. The same selection

process is applied to log compressors, comparing general-purpose compressors and log–specific compressors. Lastly, this study focuses on publicly available log datasets, primarily used for research.

1.4 Methodology

The foundation for the theoretical knowledge about log data and its processing techniques, including parsing and compression, was developed by conducting a SLR following Kitchenham's work [11]. Based on the theoretical foundation, a design science research project has been defined, designed, implemented, and evaluated to address **R1**, where the knowledge gained from the SLR is vital to decide on design choices reasonably. The design science methodology employed in thesis follows the work of Wieringa [12]. The experimental results combined with the theoretical background from the SLR are used to draw conclusions regarding **R2**. Furthermore, the identified compressors are compared, resulting in a recommendation to answer **R3**.

1.5 Outline of the thesis

This thesis develops as follow:

Chapter 2 introduces the main concepts related to this thesis. It describes log data and their computational usage. It also provides an overview of problems that can be solved with the analysis of log data.

Chapter 3 begins by introducing the methodology used to design, implement, and evaluate the prototypes developed as part of the research. The chapter then delves into the methodology and presents the results of the SLR conducted. Additionally, it proceeds to define the problem statement and derives the requirements based on the findings from the SLR. Finally, the chapter presents the design for a parser and regular expression generator, outlining the approach taken to address the identified challenges.

Chapter 4 details the implementation of the prototype and the evaluation benchmark used to collect measurables for the following evaluation.

Chapter 5 evaluates the design of the prototype. It continues to describe the procedure of evaluation and conducts a analysis considering various metrics to validate the design. It concludes by providing a recommendation for log data compression technology based on the findings.

Chapter 6 draws the conclusion of this thesis, elaborates on the related work and describes the future perspective. It also addresses the validity concerns in the research procedure and describes countermeasures taken.

2 Background

This chapter describes basic concepts which are encountered throughout the thesis. It is intended to be a reference whenever terms requiring a more detailed explanation are introduced. The first Section 2.1 describes the different types of data encountered in logs and how they are structured. The second Section 2.1 describes log data in the context of log data processing. The last section 2.3 on page 14 provides a basic introduction to log analysis and topics closely related to this thesis.

2.1 Data

2.1.1 Unstructured Data

Unstructured data refers to data that lacks any specific format and does not have a predefined schema. These are typically saved as plain–text files and are not organized in any specific structure. The following example illustrates unstructured data generated by a real–world application generating log data:

```
1    2022-12-15 21:01:23 INFO User login: user@example.com
2    2022-12-15 21:01:35 WARNING Database connection error: Could not
         connect to server
3    2022-12-15 21:02:12 ERROR File not found: /var/www/html/index.html
4    2022-12-15 21:02:23 INFO User logout: user@example.com
```

Listing 2.1: Exemplary unstructured log records

It is saved as a sequence of text lines, with each line representing a single log record. The data is considered unstructured since it lacks a predefined structure or schema. The information in each log record is free–form text without any specific format or structure.

2.1.2 Semi–Structured Data

Semi–structured data, unlike unstructured data, has some degree of organization or structure but is not as structured as fully structured data. Semi–structured data includes

data that have a predefined format. Comma-Separated Values (CSV) files or JavaScript Object Notation (JSON) objects are examples of such data. They lack a strict schema that specifies each record's specific data types or fields. The following two examples illustrate semi–structured data. The first example displays log records in CSV format from a real–world application, while the second example displays artificially constructed JSON Syslog log records.

```
1  timestamp,log_level,log_message,source
2  2022-12-15 21:01:23,INFO,User login: user@example.com,/var/www/html/
       login.php
3  2022-12-15 21:01:35,WARNING,Database connection error: Could not
       connect to server,/var/www/html/db.php
4  2022-12-15 21:02:12,ERROR,File not found: /var/www/html/index.html,/
       var/www/html/index.php
5  2022-12-15 21:02:23,INFO,User logout: user@example.com,/var/www/html/
       logout.php
```

Listing 2.2: Exemplary CSV log records

```
1  {
2  "timestamp": "2022-12-15 21:01:23",
3  "facility": "auth",
4  "severity": "info",
5  "hostname": "www.example.com",
6  "app_name": "login",
7  "proc_id": "12345",
8  "msg_id": "user_login",
9  "structured_data": {
10     "user": "user@example.com"
11 },
12 "message": "User login: user@example.com"
13 },
14 {
15 "timestamp": "2022-12-15 21:01:35",
16 "facility": "daemon",
17 "severity": "warning",
18 "hostname": "www.example.com",
19 "app_name": "database",
20 "proc_id": "12345",
21 "msg_id": "connection_error",
22 "structured_data": {
23     "server": "localhost"
24 },
25 "message": "Database connection error: Could not connect to server"
26 }
```

Listing 2.3: Exemplary JSON Syslog records

The log data in both examples contains information about records of different log levels, that is, the severity of the event, system information, or errors and warnings that occur during execution. However, the log data is semi–structured because it has a predefined format. Such formats include Syslog or the format specified by the CSV headers, but it lacks a strict schema that defines the specific data types or fields for each log record. In the JSON example, the structured_data field can have an unknown type and an arbitrarily large number of sub–fields.

2.1.3 Structured Data

Structured data is data that follows a specific structure definition and is stored in a structured manner. A typical persistent target is a database, storing the data in a table. This type of data has a well–defined schema with consistent data types and fields for each record, making it easier to search and analyze than unstructured or semi–structured data.

However, managing the databases and their schemas can be challenging and inflexible, particularly in large–scale systems or corporate environments with many dynamic data sources. In context of log management, these environments often need to consider the scalability of systems, the management of new log schemas and sources and adapting the analytic pipeline can be complex and requires careful consideration. This complexity, at last, justifies the existence of the other two mentioned structure types.

2.2 Log Data

Log data is a specific type of data similar to a journal but for some sort of system. Besides the ways described in the previous section, log data can also be represented as data structures familiar in the context of computer science. Some common log data structures found in processing log data are:

Flat–file structure: This simple structure stores log data as a sequence of text lines, with each line representing a single log record. Logs with a flat–file structure are typically stored in plain–text files and can be easily read and processed using text–processing tools and programming languages. In the context of programming languages, the flat–file structure is comparable to an array.

Structured data: This more complex structure stores log data in a semi–structured or structured format, such as a CSV file, a JSON object, or a database table. Log data with such a format is organized into fields, with each field representing a specific piece of information, such as the timestamp, the log level, the log message

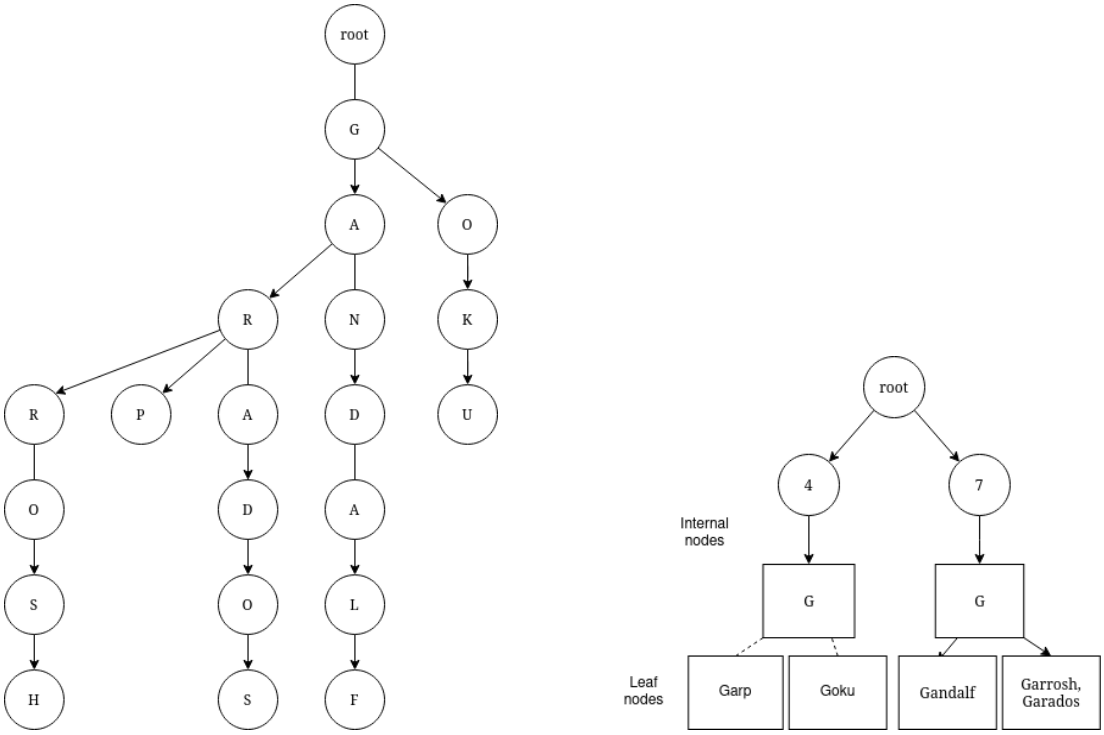
body, and the source of the log. In the context of programming languages, the structured data structure is comparable to a table.

Graph data structure: The graph data structure is a network–like structure that stores log data as a collection of nodes and edges, with each node representing a log record and each edge representing a relationship between two log entries. This data structure can be useful for representing complex, interconnected relationships between different log entries and can be quickly processed using graph algorithms. This data structure is most often found in log analysis. Tak et al. present an application for such a structure, where the absence or an exceeding amount of relationships or edges diverge from the reference graph, indicate an error behavior of the system, and enable identification of the exact location of the root–cause [13].

Hierarchical data structure: The hierarchical data structure is a tree-like structure that stores log data in a hierarchical format, with parent-child relationships between different log entries and therefore a specialized graph data structure. Hierarchical log data can be useful for representing complex relationships between different log entries and can be quickly processed and represented using tree-traversal algorithms and tree–like data structures. This data structure, notably prefix-tree, is often used in more specialized log parsers, which are exhaustively presented in Section 3.1 on page 21 and is an essential data structure of the tool presented in Chapter 4 on page 57.

Prefix–trees are one of the most prominent log parsing data structures in the literature [14, 15, 16, 17, 18, 19]. A prefix–tree in log parsing is very versatile. The most basic way of using it is placing each character of a given position as a node in the tree. Considering scenario (a) of Figure 2.1 on the next page of a string suggestion application. The input starts with the letter G. Since the root node contains a single child node with the value G, all root–to–leaf paths are suggested. The following input is O. The first layer contains two children, i.e., A and O. Next, O is chosen, and the remaining candidate, $GOKU$ is suggested. This application does not yet prove useful for log files, except for finding the Longest Common Subsequence (LCS) of an arriving log record and all available log records, for which better algorithms are known, which solve the LCS problem in quadratic time and linear space [20, 21]. However, it demonstrates the basic idea of a prefix–tree.

Now consider a scenario with a fixed depth prefix–tree. This scenario contains some definitions. There are three types of nodes. These are a single root node, internal nodes, and leaf nodes. The root node represents the starting node. Internal nodes represent the length of each value – the character count of the words – in the following leaf node. A leaf node contains a list of lists of strings and a value representing the word's first



(a) Single character prefix-tree **(b)** Fixed depth prefix–tree

Figure 2.1: Two applications for prefix–trees (a) can match an input string on the fly; (b) can group similar strings based on length and common prefix

letter. The depth is a parameter and is defined as 3. And there is a similarity function $sim(a,b)$ that provides a *similarity metric*. In this case, how similar two strings are based on whether a character at position i is equal. The tree illustrated tree state in (b) came about through inputting the strings in the following order:

```
Garp, Goku, Gandalf, Garrosh, Garados
```

Garp with length 4 creates a new internal node with the value four and a new leaf node with the representative value *G*. *Goku* is also put into this group. However, now it must decide whether it should be on the same list as *Garp*. For simplicity, the *threshold* is defined as 3. That means three characters must match to consider the compared strings similar. Next, the input is *Gandalf*. A new internal and leaf node is created, and the values are assigned. The same principle applies for *Garrosh* the $sim(Gandalf, Garrosh)$ results to 2 not greater or equal than the threshold. The last input is *Garados*. Since the leaf node reached through the internal node with length 7 has two lists, it must be decided whether it should belong to the *Gandalf* or *Garrosh* list. The similarity with $sim(Garrosh, Garados)$ confirms that the threshold is satisfied and *Garados* is put into the same group as *Garrosh*.

This example draws attention to certain characteristics of fixed depth prefix-trees in the context of log parsing:

1. Input order matters for grouping tokens together.

2. Variable length is a determining factor that excludes similarity checks even when the possibility is given that two tokens are very similar (*Garp* sharing the first three characters with *Garrosh* and *Garados*).

3. The depth parameter, that is, how many internal nodes are used, is responsible for generalization and specialization. For example, with a depth of 4, the length 7 path could have a third layer with the *a* value. This *Ga* path would have led to grouping all three length 7 values into one group, leading to a more accurate result.

4. It is an obvious choice for log parsers to enable a stream–like processing approach with this data-structure, because the grouping is not dependent on the knowledge of the complete set of log records.

This described scenario is essentially how prefix–tree–based log parsing techniques work with some nuances and deviations added, such as different similarity metrics or node definitions.

2.2.1 Parsing

Log data parsing is a crucial step in the log analysis process that involves converting unstructured log data into structured information, which makes it easier to search, analyze and extract important insights from logs. The goal is to transform log data, i.e., text, that is tailored to be human–readable to a machine-readable format.

The preceding step is to collect logs from various sources, such as servers, applications, or network devices. The collected data is then processed to identify common patterns and structures within the log data. This is done using techniques such as regular expressions, *clustering*, *heuristics*, evolutionary algorithms, machine–learning algorithms, *iterative partitioning*, and *similarity metrics*.

Clustering is the process of grouping log records based on various characteristics such as log record length, data structure–based clustering, as explained in the previous section, or the remaining characters after removing words, which contain numbers or special characters [22].

Heuristics use characteristics of logs to define rules, which for example determine a token to be static or dynamic. A very common rule in context of log data parsing is that a token that follows an equals sign is defined as an dynamic token. Other rules such as IP address or file system path are also present. Those are often enforced by applying regular expression to a specific token. If the regular expressions matches, then the corresponding rule is applied.

Iterative partitioning involves dividing a dataset into smaller, more manageable subsets that can be processed independently. Each subset undergoes a specific function or operation, and the results are subsequently merged or aggregated to produce a final outcome.

There are many similarity metrics available. Some commonly found metrics are LCS, Jaccard Distance or Cosine Similarity. In context of log data parsing, these metrics are used to determine whether a pair of elements should be grouped, clustered, partitioned or merged. The metrics usually is combined with a certain *threshold*. A threshold is a value that determines the boundary between two decision, such as is *is element X similar to Y*, which is either yes or no based on the outcome of $similarityFunction(X,Y) \leq threshold$.

2.2.2 Compression

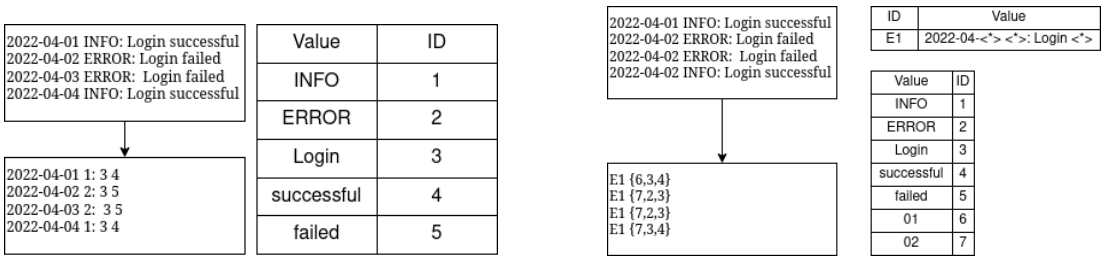
Dictionary–based Compression

Dictionary-based compression is a lossless data compression technique that replaces repeated occurrences of data with references to a dictionary or a table containing previously encoded data. This technique is used extensively in log data compression [16, 17, 23].

The following example in Figure 2.2 on the following page illustrates this concept. In example (a), a single dictionary compression is done. First, all tokens that occur more than once are identified and stored in a dictionary. All identified tokens are replaced with their shorter and less space–requiring encoding identifier. This results in a dictionary of dynamic tokens and their mapping to an identifier, effectively compressing the data.

The example (b) uses two dictionaries to compress the data further. First, the template is extracted that represents all four log records. The dynamic tokens are stored in another dictionary, like in the previous example. The resulting compressing now stores the log template id and a list of variables, where the position in the list corresponds to the token that needs to be retrieved from the dictionary when decompressing. The multi-level dictionary compression of log files is the central idea of Compressed Log

Processor (CLP), a promising tool for this task and an object of further investigation in this thesis [23].



(a) Single–level dictionary compression

(b) Multi–level dictionary compression

Figure 2.2: Exemplary dictionary compression with single and multiple levels

Content–Defined Chunking

Content–Defined Chunking (CDC) is a method of splitting large files into smaller, more manageable pieces. Compression algorithms need to consider large file sizes. They simply can not load the entire file into memory for compression. Therefore chunking is required. The intuitive way of chunking a large file is to define static values by whom the file is chunked, i.e., 64KB or 4MB chunks. Due to insertion, removal, or editing operations, this approach suffers from shifting boundaries [24]. CDC does not split every N byte, but instead determines the boundary of a chunk by checking specific bytes of a hash to meet criteria, such as the least significant 13 bit of a 48 byte *rolling hash* has to be zero [25]. A rolling hash most prominent for this task is the Rabin fingerprint [26]. Rolling hashes are calculated by depending only on the preceding hash. So given the assumption that the outcome of meeting the criteria is evenly distributed, the chunks are evenly distributed as well [25, 26]. Applications for CDC are file transmission, file synchronization, or log file compression [25, 27, 28]. Especially in file compression CDC is used in two ways. The first is redundancy elimination, a well–researched characteristic of this technique [24, 29, 30, 31]. The second log–specific characteristic is derived from the redundancy elimination and serves as a clustering technique [28].

2.3 Problem Solving with Log Data

At the 2018 Observability Practitioners Summit, Cantrill presented his view on observability and said that one must resist the temptation to quick hypotheses. To constrain a problem, such as complex debugging scenarios, one must observe the system and ask questions to get answers [32].

> 'To debug methodically, we must resist the temptation to quick hypotheses, focusing rather on questions and observations' [32]

He argued that distributed systems reached such an immense complexity because they are – as the state–of–the–art modern computing systems – built upon the history of software engineering and its abstraction layers, which also are built upon abstraction layers yielding multi–layered abstractions. The states of a modern (distributed) computing system very complex and strongly interconnected. Therefore, finding the cause of a performance issue or complex failures by only thinking and hypothesizing about it is not an efficient and sane approach to tackling these problems [32].

To solve these problems, asking questions that help narrow the investigation is essential. Cantrill says, 'Software is observable when it can answer your questions about its behavior' [32].

Shkuro defines observability as the 'capability to allow a human to ask and answer questions' and continues this definition by saying, 'the more questions we can ask and answer about the system, the more observable it is' [33]. Therefore, observability in software engineering can be built up and gradually improved by increasing the understanding of the software systems with the help of tooling. The possible observable quantities can be categorized into observability signals.

Metrics, logs, and traces are commonly called the *pillars of observability.* This is a historic artifact, since the signals are often isolated collected, processed and analyzed, resulting in three different tech-stacks. However, these three **data types** are not what defines observability. Still, this raw data must be correlated and interpreted to be helpful. They are more data types, such as alerts or Real User Monitoring (RUM). More importantly, observability is how to make sense of and use these signals. Analysis and visualization tools can aid in this process. Log data holds a unique position in this context, as metrics and traces can be derived from it. Specific metrics, such as the number of log records received, can be extracted from log data. The log data can be enriched with additional information, such as adding a field that identifies a specific request. Humans still need to ask the right questions and make sense of the data to solve problems. In other words, humans need to understand the information to solve problems.

Metrics

Metrics are functions of numerical measurables or derivatives of one or more measurables. Metrics are used to observe specific events over a given time. The following questions are common:

- How many requests went through service X?
- How much time took a request to enter and leave the Application Programming Interface (API) gateway?
- How many services did my service discovery system detect?

These questions could be answered by making sense of metrics data. The usage and usefulness of metrics can differ by quite a margin. Metrics have to have a specific goal or question in mind, which they should address. In particular, a metric should only be defined for a goal-oriented purpose. Imagine a contrary situation. A software system offers hundreds or thousands of different metrics in that hypothetical situation. The cognitive load to keep track of all that metrics may very well be higher than its use for the observability infrastructure. In addition to that, metrics should continuously be reevaluated. A metric that no longer serves a question or metrics whose corresponding question no longer matters should not be tracked. The difficulty and overhead of reevaluating too many collected metrics must be considered. Kua names some pitfalls of metrics, including the purpose orientation [34]. The purpose should be aimed at the 'whole means ensuring the metrics in use do not drive sub-optimal behavior towards the real goal of delivering useful software' and not at, as lean software development describes, trying to optimize every single aspect of a system without the result in mind, for example, Central processing unit (CPU) ticks per request in service X, which 'is often a very bad strategy.' [35].

Logs

Developers intuitively tend to depend solely on logs. When learning software development the `STDOUT` and its output is often the first choice in debugging a system. This does make sense, because the system is an imperative synchronous system without any concurrency, therefore inside a single process. This is especially true in the early stages of a developer's life. Printing a logline provides all the application's state information the developer may require in the combination of complete control of where he wants the log to appear. In the worst case, e.g., the process stops with an uncaught error, most of the time, the log provides a stack trace containing the error message, which caused the process to stop. However, production systems are not that one-dimensional anymore. Shkuro describes multiple difficulty levels of logging [33]. The first level is describes above and consists of a single process. The second easiest level Shkuro describes contains no concurrency, but multiple processes. That means processes may fork child processes, which handle single requests alone. Webservers, e.g., Nginx, use the reactor pattern to handle multiple requests by spinning up a child process to handle the request. The logging is now distributed across multiple processes. This logging is still manageable to understand but

with increased cognitive load. The third level includes basic concurrency. Not only a single child process can be created, but a single thread could also compute a request. To track the request, one must identify the thread and collect only the logs corresponding to the thread computing the request. The fourth level includes asynchronous concurrency. Shkuro names concepts like 'actor-based programming, executor pools, futures, promises, and event-loop-based frameworks' [33]. These concepts allow the distribution of a single request across multiple threads. Therefore one has to collect logs from different threads for a single request. Memory interference, race conditions, starvation problems, and many more challenges are introduced. Making sense of logs alone becomes significantly difficult. The fifth and last level is described as distributed concurrency. Network communication and multiple hosts are introduced. Challenges like aggregating all logs from multiple hosts, which by themselves could contain asynchronous concurrency, lead to the near impossibility of making sense of the logs alone. Challenges like clock drift, distributed fault handling, and complex request routing throughout the distributed system are introduced. Shkuro describes this difficulty level of debugging with logs:

> 'trying to troubleshoot request execution [...] is like debugging without a stack trace: we get small pieces, but no big picture' [33].

However, as stated, questions need to be asked with the big picture in mind. This leads to the next major datatype or signal of observability.

Distributed Tracing

Distributed tracing is the concept of tracking a single request throughout a distributed system. They differ from metrics and logs in that distributed tracing does not solely focus on a single program, process, or thread. It aims to provide the possibility to extract data by instrumentation. Instrumentation means that additional code has to be implemented at certain critical events of the system. Such events could be the beginning of a request since a common need is to track a request throughout the distributed system or the end of a request at the point the system sends the response. In order to be able to keep track of the request, including the communication between hosts over a network or the change of the thread inside a process, the technology has to provide a way to observe related events. In the past, this has been implemented in different ways.

Magpie implemented a scheme-based approach in which they generated 'a named event, timestamped with the local cycle counter' [36]. These events had to be collected and afterward processed offline. Magpie shows an early adaption of distributed tracing and its application for performance debugging. Barham et al. implemented anomaly detection with a probabilistic state machine using Magpie [36] . Each state transition is an event that has a specific probability. Event sequences with a very low probability

can be categorized as suspicious and, therefore could be an anomaly. The downside of the scheme-based approach is its specialization in a specific system and its offline processing.

Blackbox Inference is another approach, which is, in addition, a very attractive one because it does not require an additional modification of the existing source code, which is necessary for, e.g., the metadata-propagation approach. It uses statistical regression techniques to infer the association between two events. Blackbox inference comes with some downsides. Shkuro describes the following two downsides. The first is the expensive computation of generated data and the increased latency because of it. The second downside is the difficulty of creating causality between recorded events in highly concurrent and asynchronous environments [33].

The last and the de–facto standard approach is metadata propagation[37, 38]. The request is processed and then passed down with enriched trace metadata. This metadata contains, e.g., a unique trace id, which can be used to identify all corresponding spans. Spans are an abstraction of low-level events, such as making an API call or a database query. The low-level events are used in the scheme-based approach but are hard to understand and use as a developer because of their complexity and abstraction. Distributed tracing enables log aggregation. They are, in fact, extended log data with a new name. Since the trace id is unique per request, logs can be enriched by the trace id and filtered by it. Distributed tracing requires additional infrastructure, such as agent services per host and centralized service for trace collections, but offers insight, such as relationships of components, on a very detailed level. An subjective observation is that, distributed tracing is less commonly used in the real–world than metrics or logs, which could be subject of further research.

Anomaly Detection

Anomaly detection is a concept with a broad range of applications, not necessarily related to log data.

> Anomaly detection refers to the problem of finding patterns in data that do not conform to expected behavior [39].

Among the applications for anomaly detection are credit card fraud detection, insider trading detection, network intrusion detection, or failure detection based on log data [40, 41, 42].

Take, for instance, an application designed for detecting failures. First, observe the service and establish a baseline of *normal* behavior using the previously defined types of log-related data, such as metrics, log records, and traces. Suppose the service is

responsible for serving images, and every time an image is served, a specific sequence of log records is generated:

```
1     2022-12-15 21:01:01 INFO requesting image of foo
2     2022-12-15 21:01:02 INFO image /var/www/foo.png is found
3     2022-12-15 21:01:02 INFO serving image /var/www/foo.png
```

Listing 2.4: Log records flow for serving an image

This three-log-record sequence represents a standard procedure for serving images. The image name *foo* and its path */var/www/foo.png* are dynamic tokens, and any variation representing an image could occur. The normal sequence usually occurs for all kinds of requested images. However, for a specific image *bar*, the sequence deviates:

```
1     2022-12-15 21:01:01 INFO requesting image of bar
2     2022-12-15 21:01:02 ERROR image /var/www/bar.png is found
3     2022-12-15 21:01:02 ERROR aborting
```

Listing 2.5: Log records flow for serving an image with an error

An automated anomaly detection pipeline could report such an incident, leading to a investigation of the problem.

3 Concept

This chapter begins by providing the details of the planned and conducted SLR. The results of the review are presented in Section 3.1. In Section 3.2 on page 35, the method employed to design, validate, and implement the prototypes is introduced. A comprehensive description of the problem can be found in Section 3.3 on page 37. From this problem statement, the technical design questions are derived. Finally, Section 3.4.2 on page 54 presents the design of the parser, while Section 3.4.3 on page 55 focuses on the design of the regular expression generator.

3.1 Identification of Parsing, Compression and Query Strategies

To address the research questions **R1** and **R2** outlined in Section 1.2 on page 2, a SLR is planned and conducted. Firstly, the need for conducting an SLR is explained, highlighting the reasons behind its inclusion. This is followed by a comprehensive overview of the review planning, which encompasses the scope and methods used. The relevant studies are categorized and discussed, providing a analysis of the literature in relation to the research questions.

3.1.1 Motivation

The first identification of primary studies was made to evaluate the need for a SLR. No exhaustive SLR focusing on log parsing and compression techniques was found, but many recent studies were identified, which contributed to the field [17, 23, 43]. Each research described log compression techniques. Also, recent literature described the unification of log compression and queryability on compressed data [23, 43]. Chen et al. conducted a SLR and provided an overview of log compression algorithms [44]. However, the recent additions of log parsers and techniques, such as searching compressed data, need to be included.

A systematic approach is chosen because identifying literature related to a specific subject without a systematic approach is prone to bias. It also helps to structure the literature investigation to identify literature specifically relevant to a topic. Furthermore,

it encourages to search more thoroughly. Systematic research identifies, evaluates, and interprets literature to research a subject. In particular to answer specific research questions located in that field. This SLR is systematic because a process is defined that conforms to the three phases described in [11]. The three phases are planning, conducting, and reporting a SLR.

3.1.2 Review Planning

The planning phase includes the following stages [11]:

- Identification of the need for a review
- Specifying the research question(s)
- Developing a review protocol
- Evaluating the review protocol

The following sections define the review protocol, which is one stage of the planning phase, also present an overview of the found literature as shown in Figure 3.1.3 on page 26. Furthermore, the SLR is conducted, and the results are the basis for the following chapters. The development of the review protocol consists of the description of the following section, including how literature is searched, analyzed, and the results published. The methodology is evaluated with a peer–review.

Study Selection

The keywords are derived from the research objects and questions and the information of the initially identified research literature.

Therefore, the following criteria were introduced when constructing the query:

1. The process of searching specific data in compressed log data is an important aspect for the paper's authors
2. The result set needs to be manageable in the context of this thesis (no more than 150 papers returned)
3. The fact of searching compressed log data must be explicitly highlighted by the authors of this paper

The results served as the starter set for further investigation.

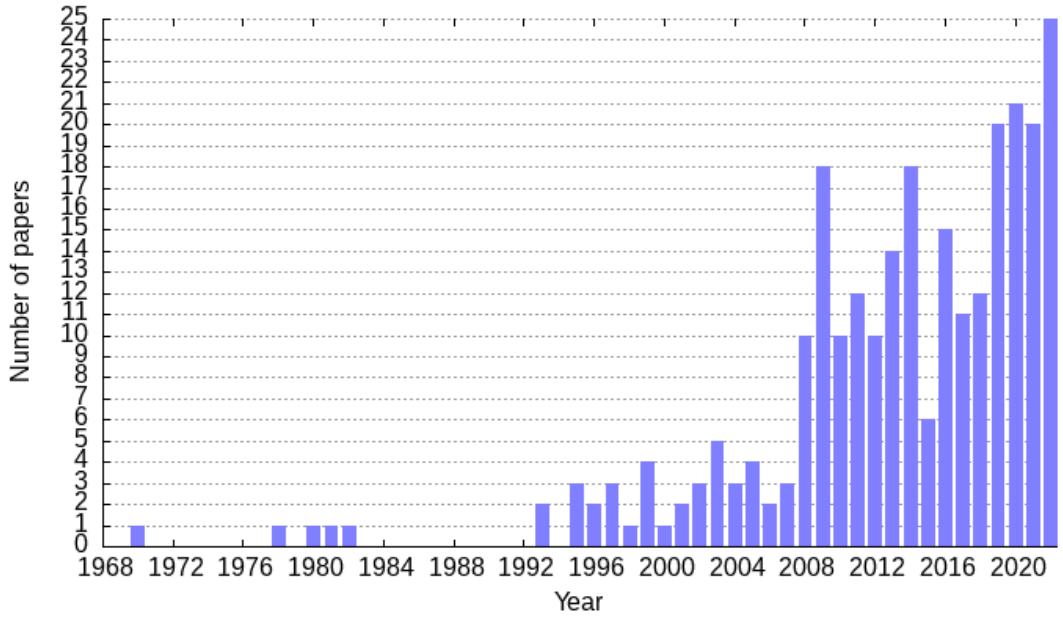


Figure 3.1: The number of literature returned by the query on IEEE Xplore before applying the inclusion and exclusion criteria.

Snowballing

There are two kinds of snowballing methods. These are backward and forward snowballing. Forwards snowballing refers to identifying new literature based on the citation of the examined literature. Backward snowballing refers to identifying new literature based on the reference list of the examined literature. This thesis uses forward snowballing to identify relevant literature further, to ensure to find the most recent publications. The examined literature in the starter set is in the final analysis set. The references in the examined literature in the starter set are used to add potentially relevant new literature that undergoes the same selection process. This procedure is iterative. The snowballing process ends as soon as all relevant literature is found, and therefore, no new literature can be determined to enter the iteration. This results in the final set for the analysis.

Inclusion Criteria

The inclusion criteria of this review were driven by research questions **R1** and **R2**. **R1** focuses on techniques that exploit the redundant structure of log data to maximize storage reduction while still allowing the searchability of compressed data. **R2** focuses on the effects of compression on post-storage processabilities, such as the ability to decompress and retrieve a subset of data for analysis. Literature that describes log parsing or log compression techniques was considered for inclusion, particularly emphasizing those enabling searching through compressed data. If any findings on the effects of compression

on processability were presented in the paper, they were also included to answer **R2**. The inclusion limit for literature was from 2003 to 2023, covering the most significant interest in log compression as shown in Figure 3.1.2 on page 22. All literature considered for inclusion was required to be available in full text.

Exclusion Criteria

The scope of this review is limited by focusing on primary studies written in English. Research literature that focuses on audio and video logs is also excluded, leading to a focus on log data generated by system, first-party, and third-party applications. Moreover, literature focusing solely on fault or anomaly detection based on log analysis is excluded because they often lack technical details about the parsing and, if present, compression techniques.

Investigation of Literature

The review's literature source was *IEEE Xplore*. The search was complemented with *dblp*.

The three defined criteria were considered in constructing the database search query. The first criterion was fulfilled by defining that *log* must be part of the paper title. The term log is ambiguous. It is used as a term for a chunk of wood and in math with usage in different contexts. This ambiguity required adding search terms relating to compression algorithms or terms narrowing down to processing log data to reduce used storage. This results in ("Document Title":Log) AND ("All Metadata":log compression) OR ("All Metadata":compressed log). The search was further refined, and the emphasis on searching through logs was included because the number of resulting literature had exceeded the amount defined in criterion (2). This refinement resulted in adding OR ("Author Keywords":log search). Criterion (3) was addressed by adding author keyword conditions, assuming they reflected the authors' intention to describe log compression techniques. This resulted in the following search query for IEEE Xplore:

```
("Document Title":Log)
AND ("All Metadata":log compression)
OR ("Author Keywords":log abstraction)
OR ("Author Keywords":log parsing)
OR ("Author Keywords":log search)
OR ("Author Keywords":log reduction)
AND ("All Metadata":compressed log)
```

Listing 3.1: IEEE Xplore database query

Dblp was also included to diversify the sources. The criteria still applied, but the resulting search query differed because of different search capabilities. Spaces represented AND statements, and | represented OR statements. Furthermore, the search needed to be more strict, otherwise the results were unmanageable and therefore contradicting criterion (2). This led to defining the most relevant and most commonly found keywords in the dblp query. The dblp search query was as follows:

```
1 log compression|compressed search|query
```

Listing 3.2: dblp database query

Assessment of Quality

Constructing a query that returns only relevant literature is unlikely. After defining inclusion and exclusion criteria, a quality assessment must be performed based on the previously defined query construction criteria.

1. Whether the title is related to the research questions?
2. Does the abstract describe log compression, log parsing, or searching compressed log data?
3. Whether the content describes a technique for either log compression, log parsing, or performing a search on compressed log data?

The result of the assessment is either Yes or No. The assessment is performed consecutively. For each consecutively performed step, the question must result in a 'Yes' to show that the paper is relevant to this thesis.

Data Analysis

After the quality assessment, a definition of the data extraction strategy is provided. The goal is to extract relevant data to answer the research questions.

1. Technique that is applied
2. Aim of the paper
3. Analysis of the parsing, compression, or search techniques

This data extraction strategy serves as a guideline for working through each study.

3.1.3 Study Selection

The selection process is depicted in Figure 3.1.3. The trend of increasing literature over time in the defined scope is illustrated in Figure 3.1.3.

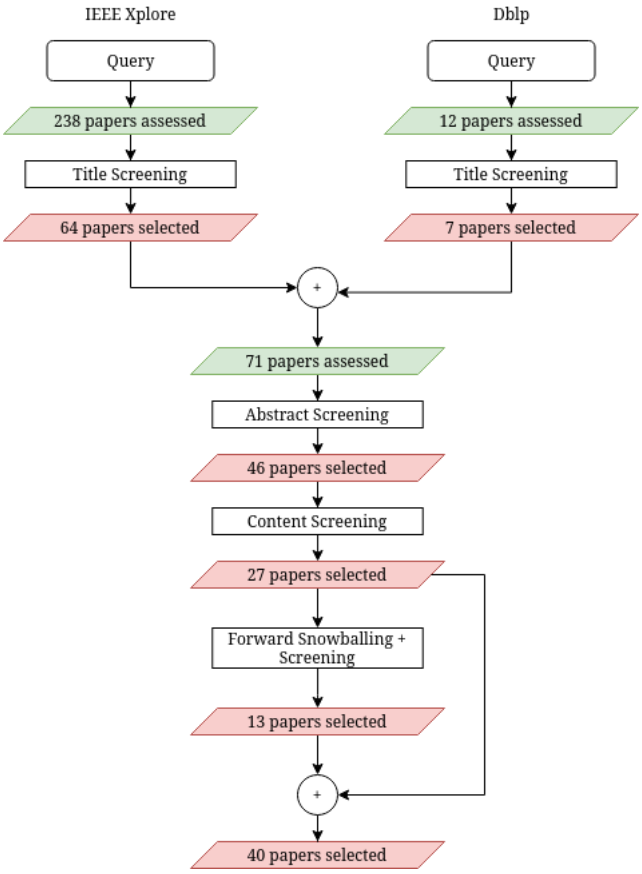


Figure 3.2: The selection process of the conducted SLR

3.1.4 Results

In recent years, an increasing amount of research has been conducted in the field of log data processing, as shown in Section 3.1 on page 21 on page 13. The examined publications can be categorized into three domains: (1) Log template extraction, which involves parsing the raw log messages into structured data for further processing; (2) Log compression, which utilizes the extracted templates to implement log–specific compression techniques that enable data reduction; Furthermore, (3) Querying the compressed data, where the search phrase is analyzed and used to find matching entries in the compressed data. The research presented herein introduces a series of tools, concepts, and techniques aimed at addressing the challenges associated with log template extraction, log compression, and querying of compressed data. These tools are

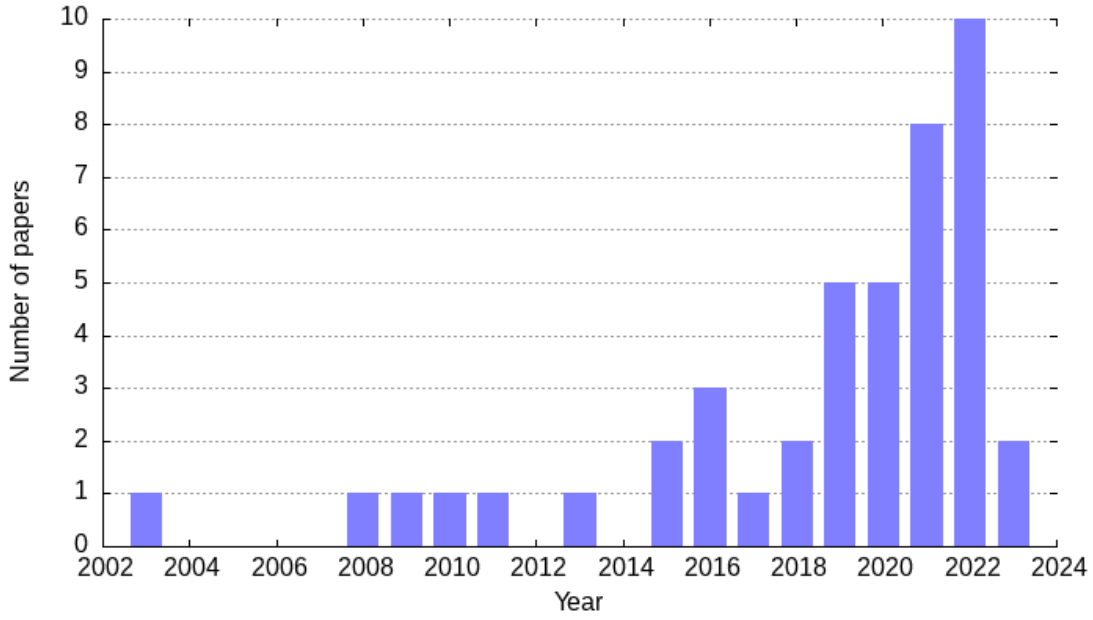


Figure 3.3: The number of literature in the final set after applying all criteria

interdependent, with those falling under the (3) domain utilizing techniques from both the (1) and (2) domains.

It is defined that the complete process of exploiting the structure of log data to achieve queryable and compressed log data requires the integration of these three domains. This comprehensive summary of existing research serves as the basis for addressing research questions **R1** and **R2**.

Log Template Extraction

Log template extraction is by far the most widely researched area in the literature taken into account. Starting with an early application of template extraction, SLCT [45] is also one of the most present. The three–step algorithm starts by constructing a summary of the dataset by applying frequency analysis. The encountered tokens and their corresponding position and occurrence count are recorded during this process. A token is considered frequent if its occurrence exceeds a user-defined percentage threshold relative to the number of log records. In the second step, cluster candidates are identified by iterating over the dataset again. Again, each log record is inspected, and the token at each position is checked to determine if it is a frequent token. Log records that contain such tokens are used to construct the candidate and are tracked in a table, incrementing their occurrence count. A candidate is selected and reported as a template if the count again exceeds the threshold, and the tokens of the template that are not considered frequent are replaced with a wildcard.

LenMa [46] uses a clustering approach that analyzes the length of each token log record. A new cluster is created if the sequence of token lengths is absent or inserted into an existing cluster if the sequence is known. The cluster is represented as a vector of token lengths and a vector of tokens. A variable is identified if the cluster word vector at a given position differs from the word vector identified as part of that cluster based on a *cosine similarity score*. Cosine similarity is an algorithm that calculates the similarity between two vectors.

LogMine [47] uses a clustering approach that is based on a custom similarity score. The algorithm starts by identifying the initial clusters, which are constructed by calculating the score of new records with a cluster representative. The first iteration uses the preprocessing to abstract tokens to token types if possible. The types are defined by the user. If the score is smaller than a cluster's continuously updated max score, then it is inserted into that cluster. If no fitting cluster can be identified, a new cluster is formed. This approach is then iteratively used to identify similar templates within a cluster and merge them.

LogSig [48] is basically an optimization algorithm for sequences of tokens tailored to log data. Each line is tokenized, and pairs of tokens are constructed. A fixed user–defined number of clusters is randomly filled with the list of pairs. The algorithm optimizes groups by moving log records, their common pairs, to different groups. A higher score indicates a better group. It tries every movement and evaluates the overall clustering quality each movement would result in. The algorithm then commits to the best improvement change.

Log Key Extraction (LKE) [49] uses A divisive hierarchical clustering algorithm. A divisive hierarchical clustering algorithm starts with very broad clusters, i.e., one or two clusters, and iteratively splits them into more differentiated clusters. Their clustering approach uses a *weighted string edit distance* as a similarity metric between two log records. A weighted string edit distance is a metric that measures how many operations would be needed to get from one string to another. The weight is meant to emphasize tokens that occur later in the sequence. Fu et al. argue that developers tend to place variables at the end of log messages. Initially, preprocessed sequences of tokens are put into two groups. The iterative group splitting measures a score for a given token for each token across all records in the group. The token with the smallest score decides whether further splitting is needed. This decision is made by comparing this score with a threshold.

Jiang et al. introduced AEL [50], an approach to the log template extraction problem. They define the anonymization step as the replacement of tokens based on heuristics. Their heuristic rule is defined as tokens that follow an equal sign are considered dynamic tokens. They define the tokenization step as a technique to group logs using a clustering

technique. Specifically, the tokens were counted to determine the bin in which the log entry belongs.

SHISO [51] presents a log parsing algorithm, which is one of the earliest design for an online parser, e.g., able to handle stream–like data. The algorithm that consists of two phases. The first phase tries to look up an existing log template for a given incoming log record. To do so, it splits the incoming log record into a list of words. The words–boundaries are defined by delimiters. Each log record is then put as a node into a tree–structure, if two nodes are similar, determined by threshold, a merge is induces, which is defined as second phase. The second phase is responsible for adjusting existing log templates. It utilizes n–grams to determine the next best candidate for a template merge.

IPLoM [52] is an iterative partition log parsing algorithm. That means it iterativly performs four steps that lead to a finer partitions. The steps perform partition by token count, token position, search for bijection and finally the discovery of log templates. It uses four parameters, which is high count relative to other techniques.

Nagappan and Voulk expanded on the approach above by improving the used SLCT algorithm, introducing the tool LFA [53]. They employed frequency tables to identify the wildcards, which required parsing the logfiles twice. During the first run, they created a pre–built frequency table. During the second run, they utilized the frequency table to look up the frequencies of tokens, which enabled them to determine the wildcards. The authors defined clusters using the identified token frequencies within a log line. Consequently, they combined clustering and heuristic techniques. This approach is a recurring trend in efforts to improve performance metrics, such as *accuracy*, *robustness*, and *efficiency*.

Zhu et al. [10] published this set of performance metrics for log parsers, along with a benchmark that serves as the de facto standard for exemplary log data used in research.

Drain [14] is one of the most prominent log parser publications and is often used as a baseline for comparison by other parsers. It provides an interface to define regular expressions used to preprocess log lines and identify the most frequent log patterns. Drain utilizes a fixed–depth tree structure to represent log messages and efficiently extracts common templates. The constructed tree describes the root, internal, and leaf nodes. Internal nodes represent heuristic rules, such as the count of tokens followed by specific tokens. Leaf nodes represent log groups, which encompass templates and IDs. IDs function as identifiers for specific log events. Dynamic variables are tokens containing digits and are replaced by wildcards. The tree structure allows for an online approach since it is continually updated. This approach is made possible by the heuristics that leverage the unique characteristics of logs.

MoLFI [54] uses a more unconventional heuristic approach. The authors define the log template extraction problem as a multi–objective optimization problem that aims to balance two competing objectives: frequency and specificity. The goal is to identify a template that matches the maximum number of log messages while maintaining the highest specificity level possible. As such, they utilize a contextualized Non–dominated Sorting Genetic Algorithm II (NSGA–II) to tackle this multi–objective optimization problem.

POP [3] utilizes recursive partitioning, which partitions the log messages based on their length into groups and further partitions them into subgroups by static parts. Once the partitioning reaches a certain level of completeness, the templates are constructed. However, POP still necessitates predefined regular expressions to identify dynamic tokens.

To develop a log filtering system for log data, Delog [55] implements a log parsing technique that groups similar logs into initial blocks. These blocks are then verified using the LCS algorithm. A variation of the Multiple Sequence Alignment algorithm generates an alignment matrix, which is then reduced to a single pattern. This process is repeated iteratively until the resulting patterns are stable. However, parameters such as the Jaccard threshold must be appropriately tuned for optimal performance.

Logan [56] is an online log parsing technique that uses LCS and a tolerance threshold to identify log templates. It performs periodic merges to eliminate duplicates and identify missed dynamic tokens. The architecture of Logan is similar to that of POP in that it uses Spark for distributed computation.

Spell [19] employs LCS to identify log templates based on the assumption that log entries with a common sequence are likely to share a common log template. The dynamic variables are stored in a list associated with the log entry. LCS is combined with backtracking, and once two log entries differ, a wildcard is inserted. If tokens do not differ, it can be assumed that they are static and, therefore, part of the template. Additionally, Spell uses a prefix–tree to enable online processing.

Logram [57] utilizes n–grams to identify similarities between log events. The approach assumes that frequent n–grams represent static tokens, while rare n–grams represent dynamic tokens. Logram automatically determines the threshold that separates static and dynamic tokens.

LPV [58] implements an Natural Language Processing (NLP) algorithm, word2vec, to extract log templates by generating a vector representation of each token in a log message. The log entry vector is constructed by summing up the token vectors and is used to cluster and categorize incoming log messages in the online phase.

Paddy [59] introduces an online parsing technique that uses a dictionary to store inverted indices mapping static tokens to their log templates. Paddy provides an interface for defining dynamic tokens as regular expressions. The found dynamic tokens are replaced by wildcards. However, it does not provide information on the location of the replaced variables, leading to potential information loss and making it unsuitable for log compression.

METING [60] is an offline log parser that, like Logram, uses n–grams to identify similarities between log events. However, instead of directly clustering logs, it builds a dendrogram, a tree-like data structure inspired by hierarchical clustering techniques. The closer two nodes are in the tree identified by distance, the more similar they are.

The USTEP [61] log parser, which employs a prefix–tree data structure for real–time processing, shares similar characteristics regarding its data structure, assumptions, and algorithms with Drain. It uses preprocessing to remove common variables such as IP addresses, file paths, and URLs. USTEP assumes that log messages with the same templates always have the same number of tokens, a limitation similar to that of Drain. No information is provided about separators, but it can be assumed that a simple separation rule is applied, which is also a limitation similar to Drain. The paper introduces a distributed architecture by adding a 'monitoring instance in charge of standardizing knowledge, and divide saturated leaves' [61].

QuickLogS [62] is an offline log parser that uses the Hamming distance and cosine similarity algorithm for template merging. Partitioning is based on the number of tokens, which introduces a strong assumption similar to Drain. An interesting feature of QuickLogS is its use of hashed values to represent high–dimensional vectors, such as template vectors.

LogPunk [63] is a parser that utilizes punctuation marks to determine a log template signature and uses clustering techniques to group templates based on these signatures. One of the challenges that LogPunk faces is generating a *punctuation table*, a list of delimiters. The authors suggest that automatically generating such a table for unknown log data would be desirable.

Prefix–Graph [64] is an online log parser that utilizes a prefix graph to merge nodes, forming templates by traversing to a leaf node. Each edge in Prefix–Graph contains a static or dynamic value that is part of a complete template. Like LPV and others, Prefix–Graph uses a vector–based similarity comparison of tokens to determine whether a merge should be performed once a certain threshold is reached.

Linnaeus [65] is a machine learning–based log classification pipeline presented by the authors. The paper focuses primarily on the architecture and industrial case study

of Ericsson's CI pipeline, with less emphasis on the technical details of the parsing technique employed.

ULP [22] is an offline log parser. The technique used in this method is to leverage the frequency of words to differentiate between static and dynamic tokens in logs. Preprocessing is done to remove common dynamic tokens. Similar log events are grouped before applying local frequency analysis to identify dynamic tokens better.

Decker presents a method for online log parsing called eLP [66], which uses intervals instead of absolute positions to tolerate variations in word positions. This is a machine learning technique that uses a combination of decision trees and interval–based classification. The method creates a grammar based on log events and forms a vector of tokens to classify related events. Preprocessing removes common dynamic tokens, and grouping similar log events before applying frequency analysis helps identify templates.

SPINE [67] is an online log parser that uses various techniques and data structures such as a prefix–tree, partitioning, deduplication, preprocessing, and parallelization. One unique aspect of SPINE is using a novel method of manual user feedback to determine whether a leaf node in the prefix–tree should be further partitioned into sub–nodes. This manual feedback, usually once a week or month, helps to improve the online partitioning of log messages and enables the parser to respond effectively to evolving log events.

Drain+[68] was developed as an improved version of existing state–of–the–art parsers, which were found to perform poorly when applied to an industrial environment. The unsatisfying effectiveness was attributed to their inability to handle variable lengths of log entries and their reliance on only static and basic separators. A limitation that has already been identified for Drain. To address these limitations, Drain+ incorporates two additional components – Separator Generation and Template Merging – which resulted in significant improvements in parsing effectiveness for private log data, and marginal improvements for public log data. Hence, Drain+ was created as an enhanced alternative to Drain.

PatCluster [15] utilizes frequency analysis to construct a tree–based data structure. The nodes in the tree correspond to token frequencies, with those closest to the root having higher frequencies than those further away. The root node is continuously updated and represents a single, mined template. The tree's depth is adjustable. The depth affects how well the algorithm performs and can detect rare patterns. PatCluster is an offline parser.

The Multi–layer Parser (ML–Parser) [69] is an approach that combines coarse and fine–grained techniques to extract templates from log data. The ML parser combines techniques, including Jaccard Distance, Hamming Distance, Longest Common Subsequence, Inverted Index, and Prefix–Tree, to achieve this goal. It operates on multiple

layers, where each layer performs more processing than the previous one, allowing it to identify finer–grained templates.

Log Compression

LogDAC [16] is a log compression technique designed to enable more effective compression of log files using general compression tools, such as LZMA or gzip. The approach differentiates between three data types: text, numerical, and dictionary. To handle these types more efficiently, the numerical type is divided into subgroups such as small integers, large integers, small decimals, and simple mixed data like file sizes. Then *elastic encoding* is applied. LogDAC researchers have found that the underlying parsing technique does not significantly affect the compression ratio. Therefore, they chose to use Drain because it balances efficiency and accuracy.

Cowic [70] introduced a dictionary–based compression technique that operates on structured logs, with each token being associated with a dictionary. The approach is demonstrated by using an Apache access log with a small size of templates in its content. This content, for example, includes URLs that can be efficiently categorized using *phrases*, a data model combining tokens representing a pattern. The technique allows for searching on compressed data using an inverted index, decompressing relevant log lines as needed. Cowic operates on structured log data.

MLC [28] utilizes CDC, to cluster log entries into buckets. Each bucket consists of log entries that are similar to each other. The Jaccard distance, is then used to compare chunks in a cluster with the chunks of an incoming log record. If they the distance conforms to a threshold the log record is assigned to that cluster, else a new cluster is created. MLC employs delta encoding, which captures only the changes between two log entries, again utilizing chunks and jaccard distance. MLC operates on structured log data.

LogZip [17] preprocesses log events by identifying commonly occurring dynamic tokens, such as timestamp and log level, using regular expressions that can be defined through an interface. LogZip then iteratively creates log templates based on a sample of the log events and attempts to match each log entry with these templates. If no match is found, the log event generates a new template. LogZip employs a prefix–tree to represent these templates, similar to Drain. Compression and decompression are achieved by representing the templates and variables as dictionaries and using dictionary mappings to restore the original log event, similar to CLP.

Querying Compressed Log Data

CLP [23] is a lossless compression technique for unstructured log data that allows for searching on compressed data by specifying a search phrase. The compression process involves tokenizing a log line, extracting dynamic tokens, and storing them in a variable dictionary. Log templates are stored in a separate log–type dictionary. When performing a search, CLP applies the same tokenization and dictionary mapping process to the search phrase and searches through the dictionaries to find matching log lines. Only the matching log lines are decompressed and returned.

LogParse [18] introduces an online log parsing technique that generalizes the template extraction process and assumes an adequate offline technique is applicable for building the initial template set using historical logs as input. To accomplish this, LogParse utilizes a prefix–tree data structure where a node represents each unique token in the template set, and a root–to–leaf path forms a template. LogParse defines the problem of updating a template as a word classification problem, where an unknown log event is divided into static and dynamic words. LogParse assumes that similarities between existing and newly arrived templates indicate a potential relationship that can be used to update existing templates with additional information from the unknown templates. To solve this problem, LogParse uses Support Vector Machines (SVM), which are supervised learning models with associated learning algorithms. LogParse applies this technique to log compression and is able to search the data, although it is not further described whether it decompresses the whole data or single entries.

3.1.5 Discussion

The results show a brief summary of the latest advancements in log template extraction, log compression, and querying compressed log data techniques. Log template extraction has progressed from basic rule-based approaches to more complex machine learning-based strategies. This results in improved handling of unstructured log data. The improved template extraction techniques are a significant driver for more efficient log compressors. The addition, enabling search on compressed data eases the log analysis and handling of log compressed log data. In contrast to the previously used general-purpose compression, which requires complete decompression of the entire archive, log compression formats facilitating partial decompression provide enhanced usability in log analysis through a faster querying process.

Few real–world applications for compression combined with partial decompression in the context of log data are known so far. CLP refers to it as ‘archivalytics’ [23]. These approaches could enhance the efficiency of log analysis and cut costs for storage.

However, a significant application is made by Uber. They report a '169x compression ratio' of their log data by integrating CLP into their log4j logging framework and this new process into their logging pipeline [4]. Having more data available enabled far more sophisticated analytics. To further elaborate on the scale of improvement and benefits of integrating such tools, Uber reported that storing logs with a retention policy of three days would cost them 180.000$ per year. With a retention policy of one month, this would increase to 1.8$ million dollars per year. Integrating advanced compression techniques resulted in 10.000$ per year with a monthly retention policy, cutting costs by over 99% [4].

In conclusion, advancements in the techniques presented in the three problem domains have improved the performance of tasks needed that enable log data analysis. As modern systems grow in complexity and scale, ongoing research in these areas is crucial to develop even more efficient, accurate, and adaptive methods for log data management. One major problem of current log parsers are the identification of dynamic tokens consisting only of alphabetical characters and how this further affects log analysis, such as compression techniques [10].

3.2 Design Method

This section summarizes the relevant concepts of *Design Science Methodology* [12] for information systems and software engineering for the design process of the presented tool. Additionally, the practical approach, or the method for applying this methodology, is described.

Methodology Description

Design Science in software engineering is an approach to software development that focuses on creating new *treatments* to address specific *problems* or needs. It involves a systematic and iterative process of designing, building, and evaluating software artifacts intended to either contribute to new *knowledge* in the field or improve a specific *context*. Two primary activities need to be carried out: designing and investigating. Both activities seek to address improvement problems: design problems and knowledge questions. Design problems require a change in the real world, and thus it is essential to consider the goals of stakeholders when designing treatments. In this context, a solution itself is a design, and there may be many possible treatments to a given problem, all of which aim to fulfill and are measured by the defined requirements.

On the other hand, knowledge questions aim to understand reality as it currently exists without requiring any changes. In this case, there is typically assumed to be only one correct answer, evaluated based on their correctness according to established criteria and standards within the field.

In both cases, design science methodology aims to develop effective treatments for improvement problems, whether by designing new artifacts or generating new knowledge. By studying the interaction between artifacts and their contexts, design science methodology seeks to identify contextually–appropriate treatments that address the needs and goals of stakeholders while also contributing to the field's knowledge base. Knowledge and design are intertwined, and addressing one problem may lead to new challenges in the other.

Design Science Research Projects

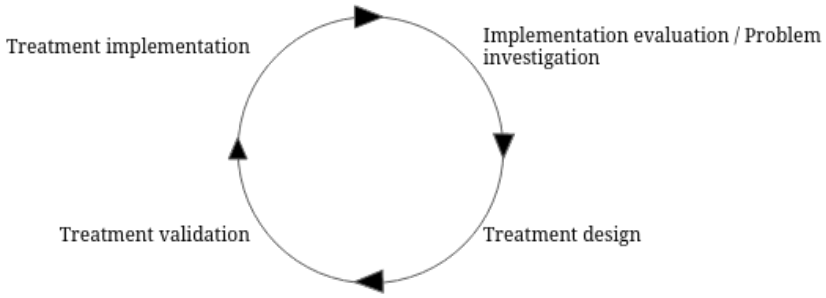


Figure 3.4: The engineering cycle, as proposed by Roel Wieringa in 'Design Science Methodology for Information Systems and Software Engineering' [12]

Design science projects operate within two types of contexts: social and knowledge. The social context includes stakeholders who impact or are impacted by the project. In contrast, the knowledge context encompasses various sources of information, such as specifications, common sense, natural science, design science, and other relevant facts. The design science project bundles the social and knowledge context, stakeholder goals, and the design and investigation cycle, which consists of subtasks shown in Figure 3.4. The figure illustrates the relationship between all the elements and the iterative nature of the process. When designing and investigating an artifact in a project, *requirements* must be defined as a direction of expected functionality. A template can be used to identify all the necessary elements, which in turn form a *technical research question* [12]:

> 'How to <(re)design an artifact>
> that satisfies <requirements>
> so that <stakeholder goals can be achieved>
> in <problem context>' [12]

3.3 Problem Statement

This section presents a formalization of the variable template extraction problem, along with a discussion on its relevance, research questions, and a requirements analysis.

3.3.1 From Messy to Meaningful: The Need of Parsing Log Data

One challenge that often comes up with logs is the desire to summarize or aggregate similar instances. The aggregation of information can have different directions. One direction is to process the logs to help humans understand their content and contextualize the data. This is often called *log analysis*. Another direction is to efficiently represent log data. This can be archived by transforming the log events into a more appropriate format or structure. This is called *log compression*, which includes *parsing* the log data.

The explanation of the challenge and the associated problem will be approached by presenting a reasonable model of the life cycle of logs as described by D. Jaglowski (personal communication, 28.09.2022):

1. Logs are generated by an application
2. "Collected" by a collector (either pushed directly from the application or written to a medium that a collector can pull from)
3. (Optionally) processed by the collector
4. Sent to a backend (such as a SaaS observability application or in–house monitoring application)
5. (Optionally) processed by the backend
6. Stored by the backend
7. Retrieved from storage (for a variety of reasons, but often for presentation to a user)

Parsing techniques can be applied during steps (3) and (5), and can prove useful to steps (4), (6) and (7), as these techniques can be directed towards various problem domains.

These problem domains are:

1. Storage

2. Transmission
3. Analysis

Storage

The accumulation of log data generated by any system can require a substantial amount of storage capacity. Log retention policies are typically employed as the primary approach to managing this. However, such policies restrict using a valuable data source for analytics, as the data is only available within a predetermined time frame. Analyzing trends over a period exceeding the retention policy's duration is difficult. Archiving the data and reducing the amount of storage used can effectively gain a more comprehensive understanding of system behavior by preserving the relevant data for an extended period.

Transmission

Data transmission, particularly with cloud providers, involves costs, especially for outbound data transmission (egress), which can be expensive. While uploading data (ingress) to cloud provider storage is generally free, egress is not free. For instance, Amazon Web Services (AWS) egress costs are usually in the range of $0.08–$0.12 per GB, excluding the free tier [71]. Comparatively, Google Cloud (GC) egress costs typically range between $0.08–$0.22 per GB [72]. The total log data generated and transmitted varies significantly depending on the organization, and thus, no assumptions can be made regarding the overall log volume and costs.

Analysis

Aggregating data has an immediate impact on the possibilities of representation. By not aggregating at all, having every piece of information available allows the analyst to shape the data in every desirable way. Aggregating the data beforehand limits the representation. This is caused by reducing the overall data set and storing it in a specific way. On the other hand, precisely aggregated data reduces the possibility of visualizing unnecessary data. One common issue encountered is alert fatigue, where the data visualized is often employed in alerting systems. If all data is visible, it is harder to determine what data matters. The ideal way to visualize log data would be to present it in a format that makes it effortless to identify good decisions and take the appropriate actions while making it challenging to make bad decisions.

It is also helpful to know that there were three occurrences of this log type and the set of obfuscated values. This aggregation strategy would affect the visualization because three unique events are merged into one, with enriched information.

There are different types of aggregation functions. In the following case, an example usage of log template extraction and of a sum aggregation function grouped over time is given:

```
1     Time: 12:00 "Info: Sold item for 1$"
2     Time: 12:05 "Info: Sold item for 10$"
3     Time: 12:09 "Info: Sold item for 100$"
```

Listing 3.3: Unprocessed exemplary log messages

```
1     Starting Time: 12:00
2     End Time: 12:09
3     sum: 110$
4     msg: "Info: Sold item for {}"
```

Listing 3.4: Aggregation based on the exemplary log messages

Aggregation functions are used to summarize data and provide useful insights. In the context of this work, several aggregation functions that may be useful for analyzing log data have been identified. These include:

List of possible aggregation functions:

1. Average: Used to calculate the mean value of a numerical data set.
2. Count: Used to count the number of events or occurrences.
3. Count distinct values: Used to count the unique values in a dataset.
4. Earliest event based on timestamp: Used to identify the earliest event in a dataset based on timestamp information.
5. List of all captured values: Used to produce a list of all the values captured in a dataset.
6. Maximum: Used to identify the highest value in a dataset.
7. Minimum: Used to identify the lowest value in a dataset.
8. Median: Used to identify the median value in a dataset.
9. The most frequent value or the N most frequent: Used to identify the most commonly occurring value(s) in a dataset.

10. Events per second: Used to calculate the rate at which events occur in a dataset.
11. Percentile value: Used to identify the value below which a certain percentage of the data falls.
12. Additional statistical aggregation functions (e.g., stdev): Used to provide additional statistical insights into the data.

It should be noted that the aggregation process involves grouping selected fields derived from parsed and typed fields in the case of unstructured logs or using a specific key in semi–structured logs. By applying these aggregation functions to log data, researchers can gain valuable insights and identify patterns that would otherwise be difficult to discern.

The presentation of several use cases and their associated problem domains underscores the importance of parsing unstructured log messages into a structured and processable format for log analysis or log compression.

3.3.2 The Variable Template Extraction Problems

The preceding discussion to structure unstructured data to make sense of it while also aiming to improve the experience with log compression tools such as CLP requires identifying the related design problems.

> 'So being able to automatically and accurately identify variable patterns would definitely help users to tune CLP's compression.' (K. Rodrigues, personal communication, 21.01.2023)

This leads to two design problems. These are based on the identification of the problem domains while focusing on the storage domain:

> **Variable Template Extraction Problems (VTEP)**
> **VTEP1** *How to extract dynamic tokens from unstructured log data*
> **VTEP2** *How to generalize the representation of dynamic tokens*

3.3.3 Requirements Derivation

This section outlines the challenges and the requirements derived from the defined variable template extraction problems, combined with the knowledge gained from the review.

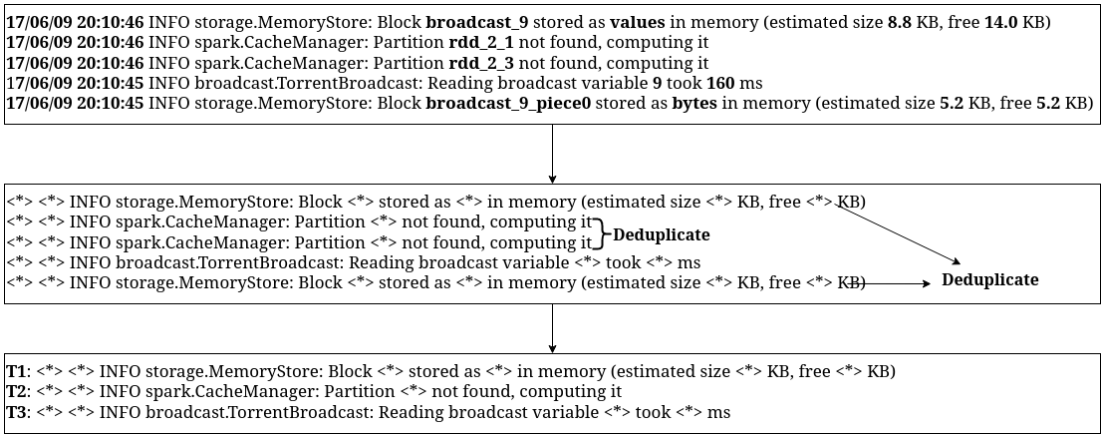


Figure 3.5: An illustrative extraction of log templates which uses log entries from the loghub Spark dataset [73]

The log template extraction problem involves improving the extraction of dynamic tokens from unstructured log data by designing a log parser that outputs sets of strings. These strings can be used to construct regular expressions that represent a generalization of the parsed dynamic tokens. This approach removes the need for CLP users to craft regular expressions using their domain knowledge manually. However, several challenges must be addressed while designing such a system to approach **VTEP1**.

Firstly, unlike traditional log template extraction that replaces dynamic parts with wildcards, the focus here is solely on the dynamic parts of the log message. Secondly, the delimiters used in tokenization are static, and the context of the particular token is important in deciding whether the token should be split further.

Finally, there is a need to consider both offline and online processing. Online processing is preferable for real–time analysis, such as monitoring, where logs are frequently streamed to a centralized machine and need to represent the state of the application or machine as soon as possible. In contrast, offline processing can be used for batch processing, where logs are stored temporarily, in a manageable size, and compressed later.

VTEP1.R1 Automatically Extract Patterns

> *If* the extraction of templates can be done automatically
> *and* accurately without requiring domain knowledge,
> *then* the user is actively supported by reducing the amount of knowledge and manual labor needed to use log compressors and can further automate the process of log template extraction.

VTEP1.R2 Efficiently Extract Patterns

If the time–efficiency of template extraction is competitive with state–of–the–art parsers
and assuming the parser is used in a different context
then the parser can be a valid and generally applicable alternative to state–of–the–art parsers.

VTEP1.R3 Standalone Application

If the tool operates independently of the log compressors,
and assuming that the artifact is provided with an sample of the compressed data,
then the artifact can be used as a standalone application to help users tune log compressors by providing instant feedback in the form of data analysis consisting of the extracted templates and variables.

VTEP1.R4 Accurately Extract Pattern

If the extraction of templates can be done accurately without requiring domain knowledge,
and assuming the parser achieves high quality log templates, identifying dynamic tokens
then then the output can be used for further analysis, such as variable template generation for file compression.

Constructing Regular Expressions from Dynamic Tokens

The problem of representing and generalizing dynamic tokens in log messages, which may include new tokens with similar patterns to existing ones, involves improving the construction of regular expressions that log compressors use to identify complex dictionary variables.

Identifying the set of variables that belong together is the first step to approaching this problem. Given, as per VTEP.R1 assumed, that the templates are accurately extracted, and the log record event can be identified, a set of variables that represent a specific dynamic token is provided. Based on this set, a generalization in the form of a regular expression is to be determined.

One challenge is the intersection between sets B_1 and B_2 encompassed by two regular expression, R_1 and R_2 respectively. For example, given the input sets $A_1 = \{"a\_1", "b\_1"\}$ and $A_2 = \{"a\_1", "c\_1"\}$, resulting in $R_1 = [a-b]\_1$ and $R_2 = (a|c)\_1$, there exists a set of strings $B_1 \cap B_2 = \{"a\_1"\}$ for which the implications on the log compressor are unknown.

The following requirements are derived from the problems statement for constructing regular expressions:

VTEP2.R1 Construct Regular Expressions

> *If* the algorithm can provide regular expressions based on a set of input strings,
> *and* assuming that the input strings are the dynamic tokens extracted and clustered by the log template extraction algorithm,
> *then* the algorithm contributes to the goal of reducing the required domain knowledge and manual labor of the log compressor user.

VTEP2.R2 Generalization of Dynamic Tokens

> *If* the algorithm can provide a generalization of the given set of input strings
> *and* assuming that the resulting regular expression captures all unknown dynamic tokens,
> *and* assuming that the input strings are sampled from the whole data,
> *then* the algorithm contributes to the goal of capturing unknown tokens, which are part of the sample data and increasing the compression ratio of the entire log data.

3.4 Design for a Logparser and Regluar Expression Generator

In this section, the research question **R1** in Section 1.2 on page 2 and the technical research questions proposed in Chapter 3.3.2 on page 40 are addressed.

In Section 3.4.1, the knowledge gained by conducting the SLR is used to provide a foundation for a comprehensive summary of current log parsing techniques that is required to answer **R1** and to propose a design in following section.

In Section 3.4.2 on page 54, **VTEP1** is addressed by proposing a log parser design based on the knowledge obtained from the SLR. This design serves as the starting point and therefore first iteration of the engineering cycle presented in 3.4 on page 36.

In Section 3.4.3 on page 55, a regular expression generator design is proposed to address **VTEP2**.

3.4.1 Exploiting the Structure of Log Data

The research question is divided into smaller components in order to approach R1.

1. How can the structure of log data be exploited
2. to reduce storage requirements through compressing
3. while maintaining the ability to query the data

Starting with the first component, the source, life cycle, and structure of log messages are analyzed to understand which techniques are applicable and why they are suitable. This analysis is used to select, mix, and build upon existing techniques for the further design process of the proposed approach. The entry points for log parsing are highlighted in the log life cycle model presented in Section 3.3.1 on page 37. Log parsing typically occurs after logs are collected at a (remote) collector or after they are sent to a centralized backend. It has been stated by He et al. that 'a crucial step of automated log analysis is to parse semi–structured log messages into structured log events' [74]. While this statement is fundamentally true, it only captures part of the picture. The majority of publications examined unstructured log data, which could be due to the use of a very common benchmark and the corresponding provided test data presented in [10]. Describing logs as semi–structured might be too specific, as it does not encompass unstructured log messages. The findings become more broadly applicable by focusing on the most challenging case, which is unstructured log data. Leveraging the structure of log messages becomes simpler as their structure is more specified, as some of the work to structure unstructured data is already done.

The Structure of Log Data

The modeling of log data and analysis of its structure was initiated by investigating the 16 log datasets presented in [10]. In doing so, the following characteristics were identified in the samples. It should be noted that due to the large size of the entries, it cannot be ensured that this is an exhaustive list of all characteristics, however, the list serves the purpose of identifying commonalities.

- Timestamp: The date and time when the log message was generated.
- Log level: The severity of the log message (e.g., Error, Warning, Info, Debug).
- Source: The component, module, or subsystem that generated the log message.
- Process Identifier (PID) or Thread Identifier (TID): The identifier of the process or thread that generated the log message.
- Message text: The actual log message describing the event or situation.

- Error code or Exception: If applicable, the specific error code or exception associated with the log message.
- Contextual information: Additional information that provides context to the log message, such as user ID, session ID, or request ID.
- File or function name: The file or function where the log message was generated.
- Line number: The line number in the code where the log message was generated.

Dataset	Time-stamp	Log Level	Source	Process	Message text	Error Code	Contextual informa-tion	File or function name	Line number
Android	Yes	Yes	Yes	Yes	Yes	No	No	No	No
Apache	Yes	Yes	Yes	No	Yes	No	No	No	No
BGL	Yes	Yes	Yes	No	Yes	No	Yes	No	No
HDFS	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No
HPC	Yes	No	Yes	No	Yes	No	Yes	No	No
Hadoop	Yes	Yes	Yes	Yes	Yes	No	No	No	No
HealthApp	Yes	Yes	Yes	Yes	Yes	No	No	No	No
Linux	Yes	No	Yes	Yes	Yes	No	Yes	No	No
Mac	Yes	No	Yes	Yes	Yes	No	No	No	No
OpenSSH	Yes	No	Yes	Yes	Yes	No	Yes	No	No
OpenStack	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No
Proxifier	Yes	No	Yes	No	Yes	No	Yes	No	No
Spark	Yes	Yes	Yes	No	Yes	No	No	No	No
TB	Yes	No	Yes	Yes	Yes	No	Yes	No	No
Windows	Yes	Yes	Yes	No	Yes	Yes	No	No	No
Zookeeper	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes

Table 3.1: Log characteristics of 16 investigated datasets

Based on the observations made, it is evident that there is significant variation in the structure and content of the log data samples analyzed. A common characteristic found in each sample is the presence of timestamps, which are typically located at the beginning of the log message, except for samples that follow the Syslog RFC 3164 format [1], where a Priority Facility Severity (PRI), a syslog specific severity attribute, may precede the timestamp. The format of the timestamps themselves varies widely across the samples, with some utilizing contextual formats that are difficult to interpret.

Another common characteristic present in each log message is a free–form plain text message body. The content of these messages varies greatly across samples, with some containing detailed information such as error codes or contextual information like user or session IDs, while others provide only minimal context.

The following log data samples illustrate a wide spectrum of complexity in log message structure. Each of which was also compared in [10] and supports the observations presented.

1 Syslog RFC3164 https://datatracker.ietf.org/doc/html/rfc3164 [75]

For example, the Apache log can be considered relatively simple due to its small message templates, which can be easily identified by many log parsers.

```
1 [Sun Dec 04 04:51:18 2005] [error] mod_jk child workerEnv in error state
      6
2 [Sun Dec 04 04:51:37 2005] [notice] jk2_init() Found child 6736 in
      scoreboard slot 10
```

Listing 3.5: Apache example log messages

In contrast, the Thunderbird log is of medium complexity, with diverse templates that are difficult to parse due to the high number of tokens in each message.

```
1 - 1131567053 2005.11.09 tbird-admin1 Nov 9 12:10:53 local@tbird-admin1 /
      apps/x86_64/system/ganglia-3.0.1/sbin/gmetad[1682]: data_thread() got
       not answer from any [Thunderbird_A2] datasource
2 - 1131567053 2005.11.09 tbird-admin1 Nov 9 12:10:53 local@tbird-admin1 /
      apps/x86_64/system/ganglia-3.0.1/sbin/gmetad[1682]: data_thread() got
       not answer from any [Thunderbird_B5] datasource
3 - 1131567054 2005.11.09 tbird-admin1 Nov 9 12:10:54 local@tbird-admin1
      gmetad: Warning: we failed to resolve data source name dn910 dn911
      dn912 dn913 dn914 dn915 dn916 dn917 dn918 dn919 dn920 dn921 dn922
      dn923 dn924 dn925 dn926 dn927 dn928 dn929 dn930 dn931 dn932 dn933
      dn934 dn935 dn936 dn937 dn938 dn939 dn940 dn941 dn942 dn943 dn944
      dn945 dn946 dn947 dn948 dn949 dn950 dn951 dn952 dn953 dn954 dn955
      dn956 dn957 dn958 dn959 dn960 dn961 dn962 dn963 dn964 dn965 dn966
      dn967 dn968 dn969 dn970 dn971 dn972 dn973 dn974 dn975 dn976 dn977
      dn978 dn979 dn980 dn981 dn982 dn983 dn984 dn985 dn986 dn987 dn988
      dn989 dn990 dn991 dn992 dn993 dn994 dn995 dn996 dn997 dn998 dn999
      dn1000 dn1001 dn1002 dn1003 dn1004 dn1005 dn1006 dn1007 dn1008 dn1009
       dn1010 dn1011 dn1012 dn1013 dn1014 dn1015 dn1016 dn1017 dn1018
      dn1019 dn1020 dn1021 dn1022 dn1023 dn1024
```

Listing 3.6: Thunderbird example log messages

Lastly, the Linux log is highly complex, with poorly identified templates across all investigated parsers, making it challenging to extract meaningful information from the log messages.

```
1 Jul  3 23:16:09 combo ftpd[768]: connection from 62.99.164.82
      (62.99.164.82.sh.interxion.inode.at) at Sun Jul  3 23:16:09 2005 .
2 Jul  5 13:36:37 combo sshd(pam_unix)[6560]: authentication failure;
      logname= uid=0 euid=0 tty=NODEVssh ruser= rhost=210.229.150.228
```

Listing 3.7: Linux example log messages

The usage of log levels across the samples is inconsistent, with some using abbreviated or full text representations for each level. When log levels are used, they generally conform

to the levels defined in the Syslog specification (RFC5424)[1]. Additionally, the source of the log messages, which indicates the component, machine, or subsystem that generated the log message, is usually included in each message, although this characteristic may vary widely depending on the specific system or application.

In conclusion, despite the significant variation in the structure and content of the log data samples analyzed, there are some commonalities that can be identified. These commonalities provide a foundation for evaluating a log data model that accurately represents the characteristics of this type of data.

The *OpenTelemetry log data model specification* is used to represent log data, and the relevant parts are described accordingly, with observations made being related to the model.

> 'The purpose of the data model is to have a common understanding of what a log record is, what data needs to be recorded, transferred, stored and interpreted by a logging system' [77].

OpenTelemetry is a significant initiative that aims to consolidate the collection and analysis of observability data, including metrics, logs, and traces. By providing a unified log data model, OpenTelemetry facilitates more efficient and standardized monitoring and analysis of various log data sources.

OpenTelemetry's log data model is designed to represent three types of log data: system logs, where there is no control over the data format (e.g., syslog); third–party application logs, where there is some control over the data format (e.g., Apache logs); and first–party application logs, where there is full control over the data format. The log data model defines a log message as a record containing two types of fields: named top–level fields with specific types and meanings, and fields stored as key–value pairs that can represent any data, potentially adhering to semantic conventions (e.g., syslog.facility for syslog format) [77].

According to the specification, the named top–level fields include Timestamp, TraceId, SpanId, TraceFlags, SeverityText, SeverityNumber, Name, Body, Resource, and Attributes. While all these fields are optional, they are generally expected to exist in log data or be represented in the future. For instance, TraceId is included to provide context information for easier correlation between observability data. OpenTelemetry's log data model provides a high flexibility, necessary considering the nature and history of log data, and it is possible to map the investigated log samples to the log data model.

1 Syslog RFC5424 https://datatracker.ietf.org/doc/html/rfc5424 [76]

In the following example, the timestamp from the log message has been translated into an ISO 8601 format that can be used in the JSON format. The "module" resource has been derived from the log message. The severity has been set to "ERROR" with a corresponding severity number of 3. The original log message has been preserved in the "Body" field.

```
1 {
2   "Timestamp": "2005-12-04T04:51:18Z",
3   "Resource": {
4     "module": "mod_jk",
5   },
6   "SeverityText": "ERROR",
7   "SeverityNumber": 3,
8   "Body": "[Sun Dec 04 04:51:18 2005] [error] mod_jk child workerEnv in
        error state 6"
9 }
```

Listing 3.8: Apache log message mapped to Opentelemetry data model in JSON format

The analysis reveals a commonality between the examined data shown in Table 3.1 on page 45 and the log data model. The model is designed in a highly generalized manner. In the investigated data, it has been observed that timestamps, sources, and message texts are consistently present. Intuitively, one might have considered these fields as required in a custom model. However, the OpenTelemetry model defines them as optional. This finding aligns with the observation made in the analysis that log data in an unstructured format exhibits significant variation, and it is not possible to assume that certain fields will always be present.

In conclusion, it can be confirmed that the observed characteristics of the investigated log data samples are compatible with the log data model specified by OpenTelemetry. The model's flexibility and optional fields align with the wide variation in log data structure and content. Thus, the OpenTelemetry log data model's definition of a log record, as described in [77], is adopted. This model provides a foundation for parsing, compressing, and querying log data while maintaining its ability to provide meaningful context and insights.

Field Name	Description
Timestamp	Time when the event occurred.
TraceId	Request trace id.
SpanId	Request span id.
TraceFlags	W3C trace flag.
SeverityText	The severity text (also known as log level).

SeverityNumber	Numerical value of the severity.
Name	Short event identifier.
Body	The body of the log record.
Resource	Describes the source of the log.
Attributes	Additional information about the event.

Table 3.2: OpenTelemetry's log data model as specified in [77]

Techniques to Exploit the Structure of Log Data

In Section 3.1 on page 21 three problem domains were identified. The techniques to exploit the structure focuses on the log template extraction domain described in Section 3.1.4 on page 27. Log template extraction involves transforming raw, unstructured log messages into structured and organized data for further processing.

The primary goal of log template extraction is to identify patterns and templates in log messages, separate static and dynamic tokens, and provide a structured representation that facilitates further analysis, such as log compression and querying.
Figure 3.6 shows the framework consisting of four steps: Preprocessing, Template identification, Template extraction and Post–processing.

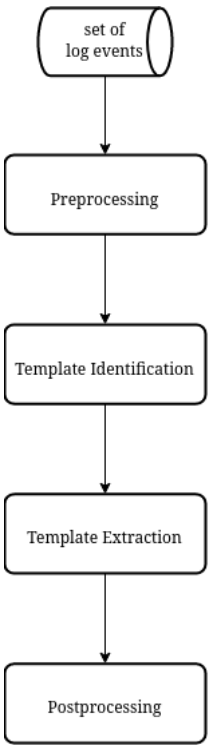


Figure 3.6: Log template extraction framework in offline mode

Preprocessing: In this step, raw log data is cleaned and prepared for extraction. This may involve removing irrelevant or redundant information, replacing common dynamic tokens (such as IP addresses or timestamps) with placeholders or wildcards, and tokenizing log messages into individual words or phrases.

Template identification: Various techniques are employed to identify templates and patterns within the log data. These techniques include clustering [45, 53, 63], evolutionary algorithms [54], heuristic methods [14], frequency analysis [22, 66], LCS [19, 55, 56], n–grams [57, 60] or machine learning [65, 66].

Template extraction: Once the templates are identified, they are extracted and represented in a structured format. This may involve creating a tree or graph–based data structure [14, 19, 60, 61] or constructing a vector representation [58, 62, 66].

Post–processing: After extracting the templates, some parsers may perform additional processing to refine the results further. This can include merging similar templates, eliminating duplicates, or optimizing the data structure for efficient querying or compression.

There are two modes in which log data can be processed: **offline** and **online** mode. In offline mode, the four steps of preprocessing, template identification, template extraction, and postprocessing are performed once on the entire log file. The steps for the offline processing are shown in Figure 3.6 on the preceding page where each step is executed consecutively with a log file as input at the preprocessing step. For example, by using a complete set of log data, the data model has access to all the necessary information to build an optimal output for the given input, since no unknown elements are expected. This is particularly useful for log compression, as offline mode allows for a more comprehensive analysis of the log data, e.g. enabling complete identification of templates, resulting in a more compact and efficient log file.

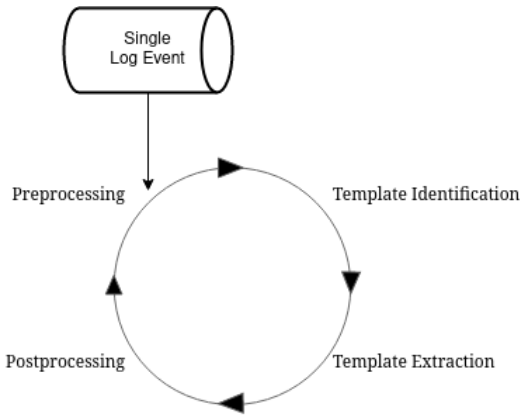


Figure 3.7: Log template extraction framework in online mode

On the other hand, in online mode, the same four steps are performed iteratively based on a single log event that is part of a set of log messages, such as a log file or log stream. This mode is suitable for real–time systems, where the processing steps are performed continuously as new log events arrive. The steps for the online processing are shown Figure 3.7 on the preceding page, where these steps are presented as a cycle with a single log event as input at the preprocessing step. For example, by using each log event, the data model can adapt and add new nodes and leaves to a prefix graph, which enables the categorization of new and previously unknown log messages.

Log Data Compression

Log data compression techniques, as described in Section 3.1.4 on page 33, are essential for efficient storage and transmission. Compression addresses the second part of **R1**. Dictionary-based methods are the most common in the context of log data compression, where queryability is a requirement [18, 23]. These techniques help reduce storage costs and enhance transmission efficiency. In order to address the second component of the research question, i.e., 'to reduce storage requirements through compressing', the CLP approach is analyzed and described, representing the techniques applied in both investigated papers that include compression and queryability [18, 23].

The CLP approach, detailed in section 2.1.2 of the CLP paper [23], compresses log messages using a two–level variable dictionary. The first level maps each dictionary variable schema to a unique ID, while the second level stores the actual variable values. Non–dictionary variable values are stored directly in the encoded log message if they can

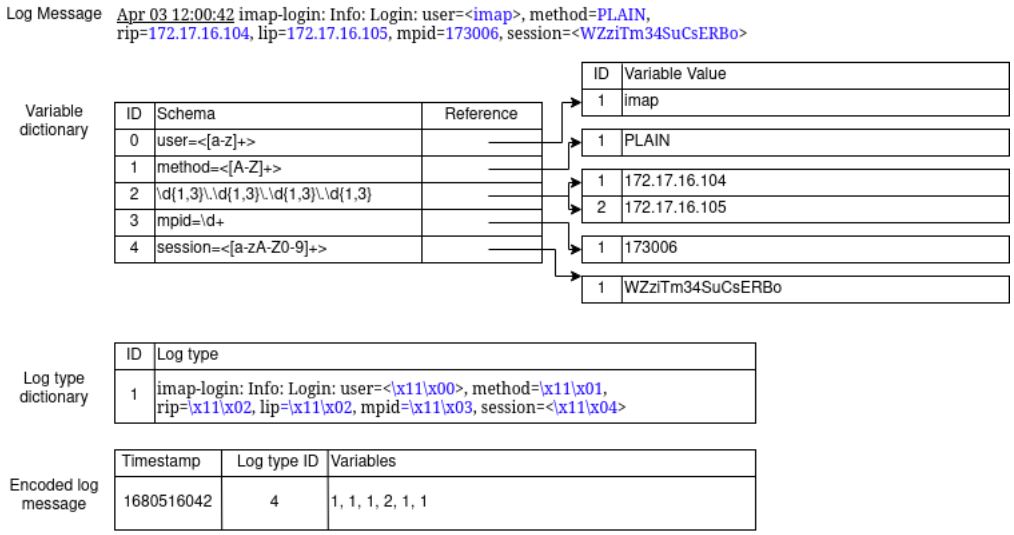


Figure 3.8: CLPs variable dictionary approach applied to production logs

be precisely encoded within 64 bits; otherwise, they are stored as dictionary variables [23].

The first level dictionary schema is defined by either manual and domain–knowledge–driven definitions, or as explored in this thesis and defined as the problem **VTEP2** in Section 3.3.2 on page 40.

The key observation is that the schema is manually provided and the essential assumption is that the schema for the variable dictionary can be automatically generated.

Figure 3.8 on the preceding page illustrates the relationship between a log message, the first–level variable schema dictionary, and the second–level dictionary of actual values, based on the CLP example applied to production logs [23]. Log messages consist of a timestamp, a resource, a log level, and a body composed of key–value pairs.

Various value types exist within unstructured log messages, such as strings containing only lower–case or upper–case letters, IP addresses, integers, and strings containing a mixture of upper and lower–case letters and numbers. Deriving actual types is non–trivial for unstructured log messages.

In the CLP paper, the authors explain that each value or variable has a schema defined [23]. The log type represents the extracted templates and the encoding for the variables. The type of the variable determines how it is stored. Variables use the dictionary approach and encode their reference ID as a 64–bit value. Non–variables, mostly floating–point values, are directly encoded as a 64–bit value. Both variables and non–variables encoding are stored in order of their occurrence in the log message [23].

Lastly, the CLP paper describes the existence of a log type dictionary containing the extracted log templates. Each value is encoded with a non–printable American Standard Code for Information Interchange (ASCII) character. The resulting encoded log message consists of a timestamp, the log type ID, and a list of variable reference IDs [23].

Searching Compressed Data

The final part of **R1** is regarding searching compressed data. All investigated publications that fulfill the three problem domains, similar to compression, use the dictionary based approach. There are alternative ways of searching compressed data such as inverted index, but since the focus is data compression this technique is suboptimal, because it is commonly stated that they need a lot of storage space to function.

In Figure 3.9, illustrating the search process of CLP, the search process is initiated by the input of the search phrase. The search phrase is tokenized, and variable values are extracted. Subsequently, the log type is constructed, and the archive dictionaries are searched for matching log types and variables. If no match is found, the process ends. However, if a match is identified, the segments are searched for corresponding encoded messages. The matched decoded messages are then output, concluding the search process. Therefore the search and also the decompression are 'generally a reversal of the compression process' [23].

In conclusion, the research question **R1** can be answered by analyzing the structure of log data and describing techniques for exploiting it. This is achieved by applying the log template extraction framework presented in Figure 3.6 on page 49, which includes preprocessing, template identification, template extraction, and post–processing to extract templates and structure the data. By doing so, the structured data can be more efficiently compressed and therefore stored and queried, addressing the storage requirements and maintaining the ability to query the data.

1. The structure of log data was investigated by analyzing various log datasets, identifying common characteristics, and mapping them to the OpenTelemetry log data model.

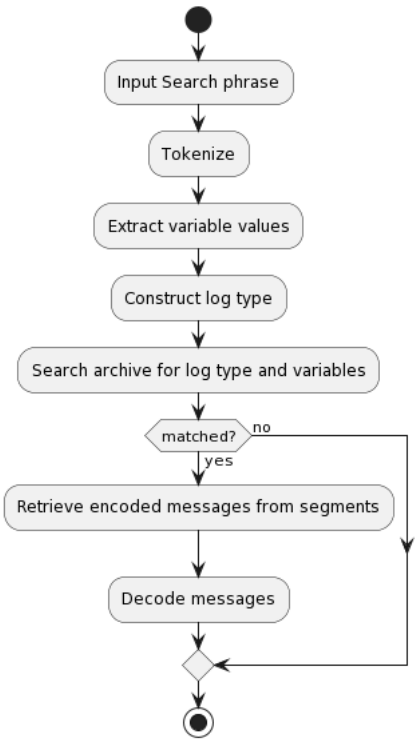


Figure 3.9: CLPs process of querying compressed data

2. Techniques to exploit the structure of log data were explored, focusing on the log template extraction domain. Various methods, including clustering, evolutionary algorithms, heuristic methods, frequency analysis, LCS, n–grams, and machine learning, can be employed to identify and extract log templates. These techniques can be applied in both offline and online modes, depending on the requirements of the system.

3. The CLP approach, which uses a two–level variable dictionary for log compression by processing the extracted templates, was analyzed and described. This method efficiently reduces storage requirements while preserving queryability.

This approach effectively addresses **R1** by exploiting the structure of log data to reduce storage requirements while maintaining the ability to query the data.

3.4.2 Parser Design

In this section, the design of a system capable of processing unstructured log data effectively is specified. The system is designed to automatically identify and extract dynamic tokens, incorporate preprocessing, and group similar log events for efficient analysis. Additionally, the system is developed as a standalone application working offline and providing a configuration mechanic for user–defined input and output settings.

1. Design an algorithm capable of automatically identifying and extracting dynamic tokens from unstructured log data without the need for manual intervention or domain knowledge.

2. Incorporate preprocessing of log events in the algorithm.

3. Include grouping of similar log events in the algorithm.

4. Perform group–based frequency analysis within the algorithm to identify dynamic tokens.

5. Develop the system as a standalone application that can operate independently of log compressors and provide data analysis based on extracted templates and variables.

6. Implement a configuration mechanic that allows users to:

    a) Specify the input file to read.

    b) Define the output file or location for the extracted patterns.

7. Aim for an template extraction accuracy of 0.8 on average on all 16 sample log data sets

8. Aim for a reasonable run time duration

The proposed design addresses the four defined requirements of VTEP1, thus effectively answering the technical design question. The design outlines an algorithm that utilizes the defined log template extraction framework, as illustrated in Figure 3.6 on page 49. Preprocessing techniques are employed, such as identifying integers, floating–point numbers, and IP addresses as dynamic tokens. Grouping and group–based frequency analysis are used for token identification. By processing the groups, the resulting outputs correspond to the template extraction process. No domain–specific assumptions or manual intervention are required.

The runtime should be justifiable and appropriate, avoiding excessively long processing times. By specifying that the system works independently of the CLP system and defining it to run as a standalone Go application with a corresponding configuration interface, the standalone requirement is fulfilled. The non–functional requirement of accurate template extraction is covered by specifying a competitive accuracy of 0.8 based on the 16 sample log datasets.

As a result, it is believed that the proposed design provides a meaningful answer to VTEP1, effectively addressing the requirements for automatic pattern extraction, extraction efficiency, standalone application functionality, and accurate pattern extraction without domain knowledge or manual labor. This approach supports users in their log compression tasks and allows for further analysis using the output from the extraction process.

3.4.3 Regex Generator Design

This section outlines the design of a regex generator system using various methods and control functionalities. The algorithm is based on input sets of dynamic tokens extracted from the previously defined log parser. Additionally, specifications are defined that guide the first iteration by determine the minimization algorithm to use Hopcroft' algorithm and specifies the Abstract Syntax Tree (AST) construction using Brzozowski algebraic method [78, 79].

1. Design an algorithm that constructs regular expressions based on input sets of dynamic tokens extracted by the log template extraction algorithm using the following methods:

    a) Represent tokens using a graph data structure.

b) Minimize the graph using Hopcroft's algorithm.

c) Convert the minimized graph into an AST using Brzozowski algebraic method.

d) Translate the AST into regular expressions.

2. Implement a configuration mechanic that allows users to:

a) Control whether to generalize digits to a character class.

b) Control whether to generalize characters to a word class.

c) Specify whether the regular expression generation algorithms should be parallelized.

The proposed design addresses the two defined requirements of VTEP2, thus effectively answering the technical design question. The design outlines an algorithm that constructs regular expressions based on the set of dynamic tokens, which are extracted in a meaningful way by the previously specified log parser. Configuration options are also specified, allowing control over the generalization of tokens. As generalization is increased, the detection of unknown or out–of–context variables is improved. However, this increase also raises the number of false positives and intersections.

As a result, it is believed that the proposed design provides a meaningful answer to VTEP2. It addresses the need for abstracting dynamic tokens in a manner that enables the CLP interface to interact with the results, effectively contributing to the goal of reducing the required domain knowledge and manual labor of log compressor users, as well as capturing unknown tokens and increasing the compression ratio of the entire log data.

Combining these two components results to an pipes-and-filters architecture capable of providing a support for CLP operators to fine–tune their configuration as illustrated in Figure 3.10.

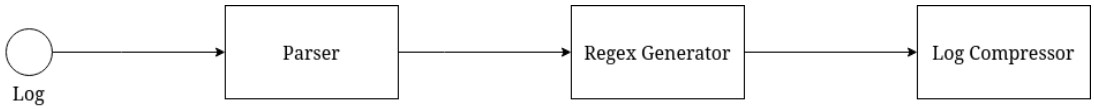


Figure 3.10: The architecture to generate regular expressions for the configuration files of log compressors for identifying difficult dynamic tokens.

4 Implementation

Manually identifying and providing regular expressions to help the compression algorithm identify dynamic tokens using domain knowledge is time–consuming. It is also hard to adapt to changing log records, which requires reevaluating the provided regular expressions. To overcome these challenges, this thesis presents the Variable Template Extraction Parser (VTEP) and the Variable Template Extractor (VTE) to analyze and understand the structure of the processed logs.

The design, as described in Chapter 3.4 on page 43, is based on the findings from the analysis of commonly occurring interfaces of state–of–the–art log parsers and by addressing common obstacles associated with these parsers. The primary goal is to provide aid in configuring the CLP, a state–of–the–art log file compression, archive, and analytics tool. This is achieved by parsing and giving structure to unstructured logs and then using this to generate regular expressions generalizing the dynamic tokens.

4.1 Overview of the Implemented System

The implemented system consists of two main components: the VTEP and the VTE. These components work together to facilitate the automatic identification and extraction of log templates and dynamic tokens from unstructured log data and the generation of corresponding regular expressions.

4.1.1 Variable Template Extraction Parser (VTEP)

The VTEP is designed to automatically process and analyze unstructured log data. It identifies and extracts dynamic tokens without needing manual intervention or domain–specific knowledge but can be fine–tuned with parameter optimization. The main functions of the VTEP include preprocessing of log records, clustering of similar log records, the group–based frequency analysis to identify dynamic tokens, and a merge algorithm to detect purely alphabetical dynamic tokens. The VTEP also features a configuration mechanism that allows users to specify whether the parsing algorithms should be concurrent, which file to parse, where the output should be placed, which log

file format is used, and which threshold should be set. The VTEP produces three types of outputs:

- The set of extracted dynamic tokens that will serve as input for the VTE
- Extracted templates
- Original log file parsed to a semi–structured form

Therefore, VTEP implements the framework illustrated in Figure 3.6 on page 49.

4.1.2 Variable Template Extractor (VTE)

The VTE is responsible for constructing regular expressions based on the input sets of dynamic tokens provided by the VTE. It employs a series of methods, such as representing tokens using a graph data structure, minimizing the graph using Hopcroft's algorithm, converting the minimized graph into an AST using Brzozowski's algebraic method, and translating the AST into regular expressions. The VTE also features a configuration mechanism that allows users to control the generalization of tokens.

4.2 Configuration

To address specification 7 of the parser design and specification 2 of the parser generator design, the *Config* struct is defined. This definition encapsulates the log file path, the output directory path, and runtime options that alter the resulting regex. It also encapsulates the log file format, which the parser uses to differentiate between contextually unimportant and important token sequences. The functionality is found in the `config` package. The parser focuses on the *Body* of log records as defined in the OpenTelemetry log data model specification in Section 3.4.1 on page 43. The configuration is passed to VTE and VTEP.

UML class diagrams are usually used to model classes in an object–oriented programming (OOP) paradigm. The Go language does not have such strict definitions of classes and objects. In contrast to languages with more common OOP principles, such as Java or C#, the class diagram shows an architectural overview. Go uses `structs`, similar to the C programming language. A struct is displayed as a S with an aquamarine background. A type is displayed as a T with an orange background. A primary data type with a green background is displayed as a C. Private fields are displayed as a red rectangle followed by the field name and the data type. Public fields are displayed as a green

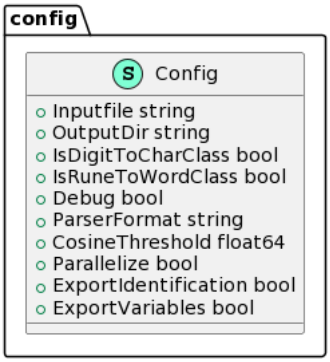


Figure 4.1: UML class diagram of VTE Configuration Struct

circle followed by the field name and the data type. The relationships follow the UML standard.

4.3 Variable Template Extraction Parser

In this section, the implementation of VTEP is described. The implementation details are oriented towards the specification defined in 3.4 on page 43 and how the implementation fulfills them.
The functionality is found in the `parser` package. VTEP is based on ULP, an approach for parsing log files designed by Sedki et al. [22], improved and extended with a specialized log line parser incorporating additional heuristic rules, a merge algorithm, concurrency and a hash lookup. VTEP is a *Go*[1] package that is part of the VTE Command Line Interface (CLI) application written in Go.

The parser consists of three main components, which are **Parser**, **LogGroup**, **LogRecord**. The class diagram illustrated in Figure 4.2 on the next page shows each component's corresponding fields and a selection of relevant functions.

Specification (1): Design an algorithm capable of automatically identifying and extracting dynamic tokens from unstructured log data without the need for manual intervention or domain knowledge

This high–level specification encapsulates the complete parsing process. Figure 4.3 on page 62 and Algorithm 1 on page 63 describe and illustrate the process. The parser initiates the processing, i.e., extraction and identification of log files, with the `Parse` function. The log file format is converted to a regular expression based on the provided configuration to differentiate the log fields. This somewhat contradicts (1), as the parser user must supply domain-specific knowledge, such as the log format. This is necessary to enable comparability to other log parser benchmark results, which only investigate the Body part of the log record. Each line is read, and a LogRecord is constructed. The slice of log records is passed to `extractTemplates` function. This results in a slice of LogGroups. Each LogGroup contains a clustered set of log records. Templates are built using the tokens in the LogGroup's *representative*. The representative is the first element of the slice. Since all LogGroups in a cluster share the same length and sequence of *token types*, the templates effectively abstract a group of log records. The log lines are combined with their *EventID* and template. Finally, the log template extractions and the log template identification files are saved to disk. The log template extraction file contains all log templates, including the wildcards. The log template identification file contains the parsed log file with its log event identification, i.e., the corresponding log event ID and the template.

1 The Go Programming Language https://go.dev/ [80]

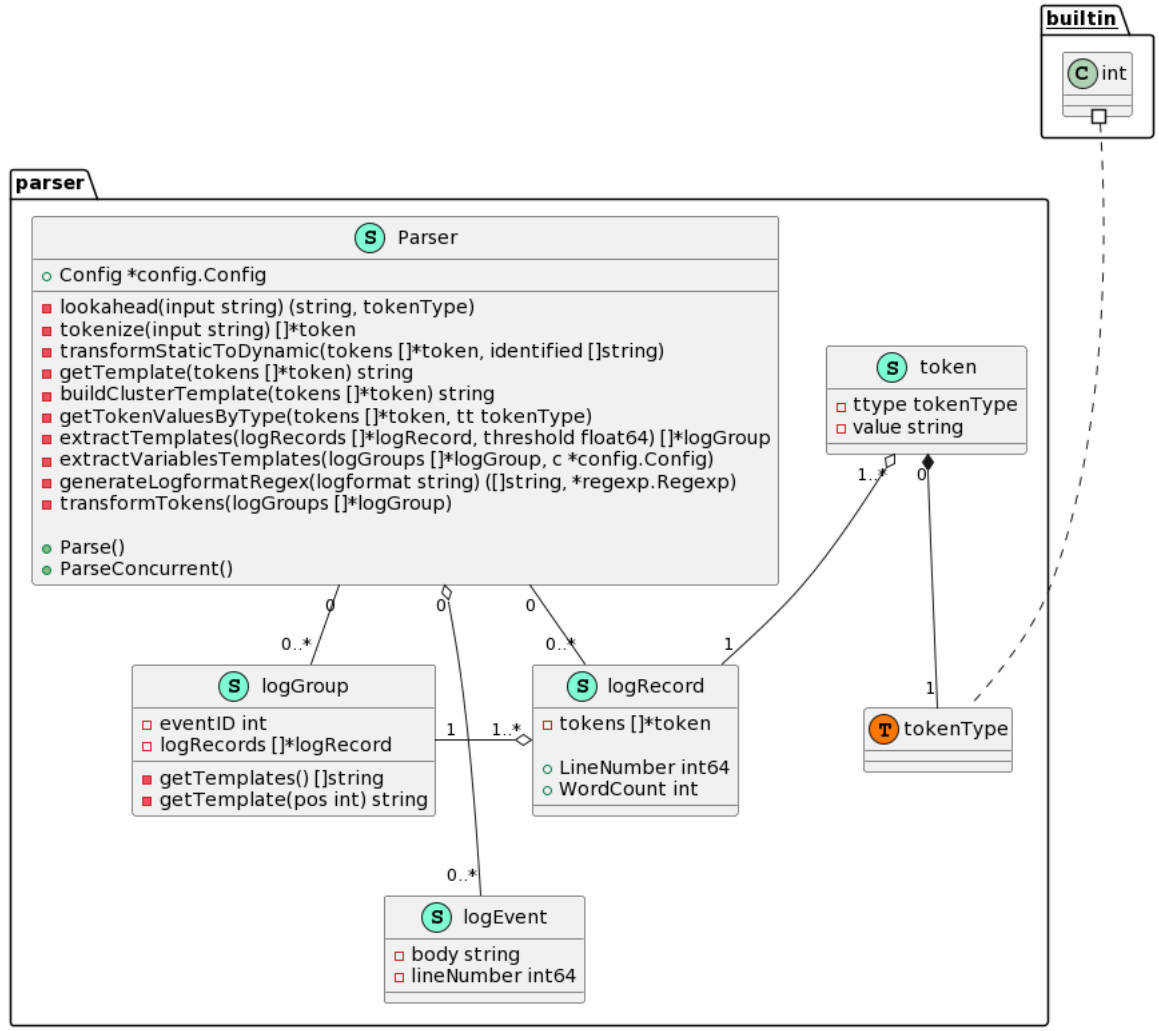


Figure 4.2: UML class diagram ofVTEP

Specification (2): Incorporate preprocessing of log events in the algorithm

Preprocessing is commonly done by using regular expressions to identify specific types of tokens considered dynamic tokens. VTEP employs a hand-crafted log body parser that breaks down the sequence of tokens separated by whitespace into tokens or lexemes. These tokens are the building blocks for all preceding operations, and their identification is well–worth the time spent, considering their advantages in the template construction and merging algorithm, such as the fast building of different token combinations and token type checking between log records and their token sequences. Whitespaces split the log body, and each resulting token is inspected with a regular expression. The regular expression checks have a specific order, and the first matching one classifies the token. The tokenization is the most expensive operation within the parsing process and takes about 85% of the total CPU time, measured with the HDFS dataset. The token

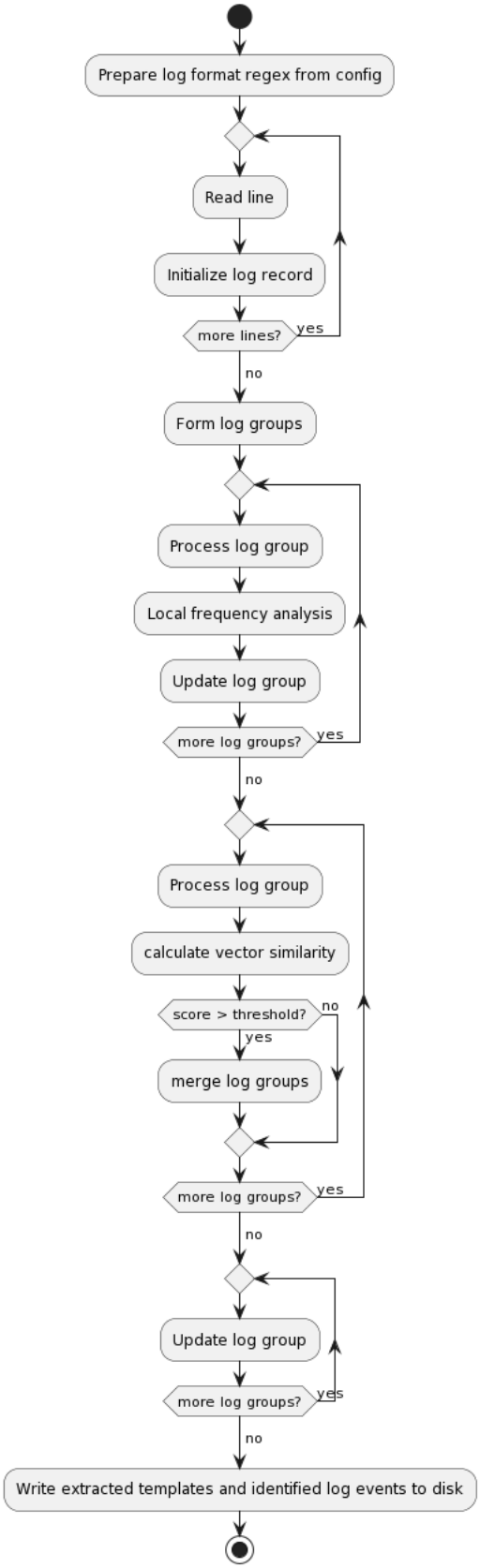


Figure 4.3: UML activity diagram of the parsing process

Algorithm 1 VTEP Parser Algorithm

Input $logFile$, $config$
Output $templateExtractionFile$, $templateIdentificationFile$
Procedure $Parse$

1: $fileformat \leftarrow createRegex(config)$
2: **for** each $line$ in $file$ **do**
3: $logRecords \leftarrow parseLine(logRecord)$
4: **end for**
5: $logGroups \leftarrow extractTemplates(logRecords, threshold)$
6: $exportLogTemplateExtraction(logGroups)$
7: $exportLogTemplateIdentification(logGroups)$

types are defined in Table 4.1, where ¬ is used for formatting and not considered part of the expression.

Token Type	Heuristic Rule
IPAddress	`\b(?:\d{1,3}.){3}\d{1,3}\b`
URL	`\b(?:[\w-]+.)+\w+\b`
Bool	`\b(?i)(true|false)\b`
Float	`\b-?\d+.\d+\b`
Path	`^/(?:[^/\s]+/)[^/\s\W.][^/\s]\b`
Quote	`^"([^"]*)"`
UUID	`\b[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}` ¬ `-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}\b`
Date	Identified by the dateparse package [81], combinined with additional heuristic rules to detect false positives
Separator	`(\s|[%s])`, where %s is a exhaustive list of predefined and commonly occurring separators
Dynamic	Identified by a heuristic rule that considers the preceding token.
Static	Everything not matching

Table 4.1: Regular expressions and heuristic rules to identify token types

Specification (3): Include grouping of similar log events in the algorithm

The idea of clustering records into groups originates from the approach proposed by Sedki et al. [22], which is paraphrased as follows:

> The function that measures the similarity of two log records considers both the number of tokens they contain and the tokens that are most likely static tokens. Tokens are most likely static if they do not contain digits and/or special characters. Each processed log record is split by a space character into tokens. Only tokens that contain alphabetical letters are used to construct the log record template string. The other tokens, i.e., containing digits and non–alphabetical characters, are most likely variables.

> The resulting tokens and the total number of tokens, represented as a string, are concatenated. Log records with matching templates are classified as part of the same LogGroup.

This approach effectively and efficiently serves as the initial clustering operation and is very cheap considering the preprocessing and identified tokens. The template is constructed by concatenating all static tokens of each LogRecord. In addition to the original design, instead of comparing the string representations of each LogGroup, a hash has been introduced that identifies the LogGroup by hashing the template of the representative. The hash is created on LogGroup instantiation. The Algorithm 2 on page 66 depicts the log extraction algorithm. The clustering compares the template by hash comparison and additionally the count of total tokens between the current log record and the LogGroup representative.

Specification (4): Perform group–based frequency analysis within the algorithm to identify dynamic tokens

The subsequent step involves analyzing the grouped log records. The goal is to identify static and dynamic tokens within the group of logs. Sedki et al. suggest employing local frequency analysis for this purpose [22]. A frequency analysis involves counting occurrences. In this context, frequency analysis refers to counting static tokens across all log records within a LogGroup. The frequency analysis is deemed local, as the counting occurs within a group of tokens already sharing template–based similarity.

The local frequency table is then used to transform each static token that does not meet the *upper–bound condition*. The upper–bound condition is the maximum of distinct token occurrence. As self–explanatory example, consider the following scenario:

```
Input: [Goku is 120cm tall ]
Input: [Garados is 300cm tall ]
Token sequence: [static static dynamic static]

Goku: 1
Garados: 1
is: 2
tall: 2

upper-bound: 2
transformation of sequence position 0 to dynamic
resulting token sequence: [dynamic static dynamic static]
```

In this approach, a merging strategy is incorporated after the initial identification of log groups. Algorithm 3 on page 67 outlines this process. The algorithm iterates through each LogGroup, assessing its similarity to other log groups and identifying the most similar LogGroup. If no LogGroup surpasses the threshold, the function proceeds to the next LogGroup. Otherwise, the representatives of each LogGroup, which are the first LogRecord in each group, are compared. This secondary comparison involves assessing the sequence of token types. For instance, in the example above, both inputs have a sequence of `[dynamic static dynamic static]` and are deemed mergeable. A merge combines log records into a single group, after which one of the LogGroups is removed. As the algorithm operates in place, the shifting indices are tracked accordingly.

This approach fulfills specification (4).

Specification (5): Develop the system as a standalone application that can operate independently of log compressors and provide data analysis based on extracted templates and variables

The application has been developed to operate independently of log compressors and provide data analysis based on extracted templates and variables. To achieve this, the software has been designed as a Command Line Interface (CLI) application in Go, which bundles the main functionalities:

1. benchmark
2. parse
3. regex

However, for this specification, the parsing functionality is the most relevant. Users must provide input and config flags when invoking the 'parse' CLI command. The input flag specifies the file to be parsed, while the config flag describes the path to the configuration file. The configuration file contains essential details such as the log format, e.g., for Apache:

```
1     \[<Time>\] \[<Level>\] <Content>)
```

Listing 4.1: Apache log format

Moreover, further details, such as output directory, cosine similarity threshold, and whether the process should run concurrently using the *parallelize* flag.

Once the parsing process is invoked, the output files are written to disk, which include the file with the log templates, the file identifying each log line with the template, and a JSON file that contains dynamic tokens corresponding to their position.

Additionally, performance metrics such as total processing time are displayed on STDOUT.

These output files can be used as input for other commands, such as *regex*, which generates regular expressions representing alphabetical dynamic tokens that are difficult for CLP to detect. The application is compiled into a single small binary of about 12 MB in size. By providing a straightforward interface between different domains through output files, the software achieves independence and adheres to specification (5). Defining and describing the configuration also adheres to specification (6).

Algorithm 2 VTEP Template Extraction Algorithm

Input *logRecords*, *threshold*
Output *logGroups*
Procedure *extractTemplates*

1: **for** each *logRecord* in *logRecords* **do**
2: **for** each *logGroup* in *logGroups* **do**
3: **if** *logGroup[0].TemplateHash* = *logRecord.TemplateHash* **then**
4: *logGroup* ← *logRecord*
5: *break*
6: **end if**
7: **end for**
8: **end for**
9: *identifyDynamicTokens*(*logGroups*)
10: *mergeSimiliarGroups*(*logGroups*)
11: *identifyDynamicTokens*(*logGroups*)

Algorithm 3 VTE Merge Algorithm

Input *logRecords*, *threshold*
Output *logGroups*
Procedure *extractTemplates*

if *threshold* ≥ 1.0 **then**
  return
end if
for $i \leftarrow 0$; $i <$ *length(logGroups)* **do**
  $max \leftarrow 0.0$
  $k \leftarrow -1$
  vectorI $\leftarrow$ *staticTokens(logRecord[i])*
  for $j \leftarrow i+1; j <$ *length(logGroups)*; $j \leftarrow j+1$ **do**
    vectorJ $\leftarrow$ *staticTokens(logRecord[j])*
    if *wordCount(logGroup[i])* $\neq$ *wordCount(logGroup[j])* **then**
      continue
    end if
    similarity $\leftarrow$ *cosineSimilarity(vectorI, vectorJ)*
    if *similarity* $>$ *threshold* $\wedge$ *similarity* $> max$ **then**
      $max \leftarrow$ *similarity*
      $k \leftarrow j$
    end if
  end for
  if $k \neq -1$ **then**
    representative1 $\leftarrow$ *logGroup[i].LogRecords[0].tokens)*
    representative2 $\leftarrow$ *logGroup[j].LogRecords[0].tokens)*
    if *isEqualTypeSequence(representative1, representative2)* **then**
      logGroup[i].LogRecords += logGroup[k].LogRecords)
      remove(logGroup[k])
    else
      $i \leftarrow i+1$
    end if
  else
    $i \leftarrow i+1$
  end if
end for

4.4 VTEP to VTE JSON Interface

A JSON format containing the relevant results of VTEP for VTE, has been defined. The dynamic tokens file is designed to be the only connection point between the parser and regex package, effectively defining their interface.

This file represents dynamic tokens corresponding to specific log templates reflected in the file structure. The JSON file contains a list of objects representing LogGroups. Each LogGroup includes its template as a value for identification purposes and is identified by its EventID. Moreover, each LogGroup object contains a list of lists of

strings representing the position of a dynamic token. The outer lists index reflects the position of the dynamic token in the template. The inner list contains the dynamic tokens at the given position of all log records in the LogGroup.

The dynamic tokens are further reduced in scope only to contain dynamic tokens considered hard to detect for CLP. These types of tokens do not contain digit characters at all. Therefore strings primarily consist of alphabetical and special characters. Not necessarily all wildcards are present for each template, because the merging algorithm considers variable length log sequences. The following shows an example of the described format, which is extracted from the Hadoop dataset parser results:

```
[
    {
        "event_id": 1,
        "template": "OutputCommitter set in config null",
        "variables": []
    },
    [...]
    {
        "event_id": 3,
        "template": "Registering class <*><*> for class <*><*>",
        "variables": [
            [
                "$EventType",
                "$EventType",
                "$EventType",
                "$JobEventDispatcher",
                "$TaskEventDispatcher",
                "$TaskAttemptEventDispatcher"
            ],
            [
                "$ContainerAllocatorRouter",
                "$SpeculatorEventDispatcher",
                "$ContainerLauncherRouter"
            ]
        ]
    },
    [...]
]
```

Listing 4.2: Snippet of the VTE input file in JSON format

4.5 Variable Template Extractor

In this section, the implementation of VTE is described. The functionality is found in the `regex` package. The implementation details are oriented towards the specification

defined in 3.4 on page 43 and how the implementation fulfills them.

This section also summarizes the iterative knowledge gains and design adaptions as intended by the design science methodology described in Section 3.2 on page 35. The class diagram illustrated in Figure 4.7 on page 76 shows each component's corresponding fields and relevant functions. Figure 4.4 describe and illustrate the overall process of generating a regular expression based on the dynamic tokens.

The `parse` function initiates the process by tokenizing each input word and uses the tokens to construct the `PrefixTree` as described in Section 2.1 on page 11. The `PrefixTree` can be interpreted as a Deterministic Finite Automaton (DFA). Assuming that a `PrefixTree` consists of a root node, intermediate nodes and leaf nodes, then the root node is the initial state of the DFA, each intermediate node's value, e.g. a string, is a transition symbol connecting two states and the leaf nodes are accepting states. The DFA is minimized with the Hopcroft algorithm [78]. The minimization leads to more concise regular expressions. This is obvious, because the minimization merges equivalent classes of states, therefore only reducing or minimizing the total amount of states and transitions without diminishing the expressiveness of the regular language represented through the DFA. The minimized DFA is then used to built an AST for PERL–compatible regular expression [82]. Finally the string representation of the AST is returned as result.

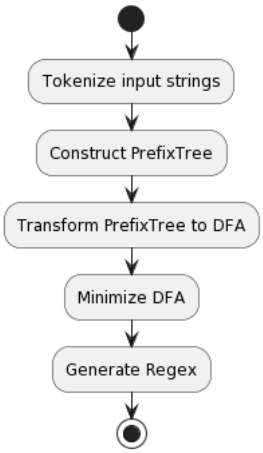


Figure 4.4: UML activity diagram of the regex parse process

Specification (1): Design an algorithm that constructs regular expressions based on input sets of dynamic tokens extracted by the log template extraction algorithm using the following methods

(a) Represent tokens using a graph data structure. The prefix tree efficiently deduplicates common prefixes, reducing the processing time for the subsequent DFA construction, minimization, and regex generation steps. Figure (a) 4.5 on page 75 presents the extracted dynamic tokens from a real-world production log file from the Hadoop dataset. These dynamic tokens, found at specific positions in a log file template, are transformed into regular expressions and used to enhance the CLP configuration for detecting such dynamic tokens. These tokens are derived from the following snippet of the VTE JSON interface file, which is extracted by VTEP and provided as input to VTE:

```
1 {
2     "event_id": 95,
3     "template": "attempt_<*>_<*>_m_<*>_<*> TaskAttempt Transitioned from
          <*> to <*>_CONTAINER_CLEANUP",
4     "variables": [
5         [...]
6         [
7             "SUCCESS",
8             "SUCCEEDED",
9             "FAIL",
10            "FAIL"
11        ]
12    ]
13 },
```

Listing 4.3: Snippet of extracted dynamic tokens from the Hadoop dataset

It can be observed that the common prefix *SUCCE* is successfully identified. Two branches feature ambiguous suffixes, namely *EDED* and *SS*. As anticipated, the *FAIL* dynamic token follows a unique path. However, it is worth noting that the input is deduplicated, as the *FAIL* token appears only once in the tree, even though it occurs twice in the input. This data structure and its application effectively meet the specification **1.a**.

(b) Minimize the graph using Hopcroft's algorithm. Hopcroft's minimization algorithm is a well understood for deterministic finite automata [78]. The algorithm aims to minimize the number of states in a DFA while maintaining its functionality, which is beneficial for reducing the space required to store the DFA, simplifying its understanding, and improving the performance of the AST construction that use it. Moreover, minimization helps to generalize regular expressions, making them more robust and better suited for CLP's configuration file. There are alternatives to the prefix–tree

& Hopcroft approach, including prefix–tree combined with either Moore's algorithm [83], Brzozowski's algorithm [84], or Aho-Sethi-Ullman's algorithm [85, pp. 182–184] and incremental minimization algorithms, such as Watson's algorithm and its successor Daciuk's algorithm [86].

Daciuk's algorithm is particularly interesting, because it does not require to construct a prefix–tree beforehand. It constructs a minimized DFA directly. Since the design specified to use the Hopcroft algorithm and the treatment analysis resulted in the knowledge gained that consisted of potential for a simpler implementation and faster execution time, the Daciuk algorithm was implemented and compared. Both algorithms result to the same qualitative outputs. The methodology to measure the execution time is simple. The time is measured using the go built-in time package. A start timestamp is constructed, the function is executed and the difference between the start timestamp and the current timestamp is used to measure the execution time. The Daciuk algorithm reduced the code complexity and increased the performance significantly. The average execution time of a `Parse` function with the Hadoop dataset over 100 runs was measured. The prefix–tree & Hopcroft resulted to 43.39ms, while the Daciuk algorithm resulted to 1,79ms. This is a execution time decrease of 95.87%. This experiment results are consistent with the results and recommendations of Daciuk, which suggests that 'Both incremental algorithms are the fastest in practical applications' and 'Trie + Hopcroft minimization is the slowest algorithm' [87].

The Figure 4.5 on page 75 shows the resulting DFA from the prefix–tree. They do not differ much, except the nodes are transformed to states, and the edges of the DFA hold the tokens. Equivalent states are minimized at this point.

(c) Convert the minimized graph into an AST using Brzozowski algebraic method Two methods for constructing regular expressions based on a list of input strings, e.g. the purely alphabetical dynamic tokens in a log file, have been explored: the Brzozowski Algebraic Method and the State Elimination Method [79, 88].

The Brzozowski Algebraic Method relies the substitution method for solving systems of equations. The method iteratively computes and substitutes regular expressions to derive a final expression composed of all subexpressions. Alternative but methodically similar algorithms are presented by McNaughton & Yamada, and Kleen [89, 90].

Brzozowski's method is based on Arden's theorem, which is a language equation of the form $X = AX \cup B$, where X, A, and B represent languages [88]. In this context, language equations resemble numerical equations, but their variables assume values of formal languages instead of numbers. The key operations involved are concatenation ($\cdot$),

union ($\cup$), and Kleene star ($*$). These mathematical notations forming the foundation of the algorithm.

The algorithm constructs an equation for each state, using the transitions that represent subsets of the language to populate the equation. It then solves the system of equations by successively substituting terms from preceding equation into the current equation. This process is repeated until only one solvable equation remains, yielding the desired regular expression.

Consider the following example with the given DFA in Figure 4.6 on page 76, with the input strings *abc* and *azc*, given ε is the terminating symbol:

$$
\begin{array}{lcl}
Q_0 & = & aQ_1 \\
Q_1 & = & bQ_2 \cup zQ_2 \\
Q_2 & = & cQ_3 \\
Q_3 & = & \varepsilon \\
\hline
Q_0 & = & aQ_1 \\
Q_1 & = & bQ_2 \cup zQ_2 \\
Q_2 & = & c(\varepsilon) \\
\hline
Q_0 & = & aQ_1 \\
Q_1 & = & b(c(\varepsilon)) \cup z(c(\varepsilon)) \\
\hline
Q_0 & = & a(b(c(\varepsilon)) \cup z(c(\varepsilon)))
\end{array}
$$

Assuming every transition symbol to be a literal expression, the concatenation and union operations results to:

$$Literal(a) \cdot (Alternation(b|z) \cdot Literal(c))$$

Finally stringifying the expression to **a(b|z)c** or **a[bz]c**, in case of the possibility to identify a *CharClass*. This further shows that a given DFA, and therefore language, can be expressed by different resulting regular expressions.

On the other hand, the State Elimination Method, while still based on Arden's theorem, works by iteratively removing states from the graph, while updating the remaining transitions to maintain the functionality of the original automaton [88]. This method systematically eliminates states until only the start and final states remain, yielding a regular expression that represents the accepted language as shown in Algorithm 4 on the next page.

To compare the performance of these two methods, their execution times were measured when constructing regular expressions for a given set of dynamic tokens. The same methodology as in specification 1.b applies. The results are as follows:

- State Elimination Method: 1.79 ms

Algorithm 4 VTE Regular Expression from DFA algorithm

Input DFA
Output string representation of regular expression
Procedure *GenerateRegex*

```
for each transition in dfa.transitions do
    initialize literal for symbol in transition
end for
add a new epsilon initial state
for each acceptedState in dfa.acceptedStates do
    add a new epsilon accepted state
end for
for each state in dfa.states do
    incoming, outgoing, remaining ← transitions(state)
    for each inc in incoming do
        for each out in outgoing do
            concat ← concatinate(inc.Expr, out.Expr)
            transitionStateMap ← concat
        end for
    end for
    for each from, toMap in transitionStateMap do
        for each to, expressions in toMap do
            remaining ← Union(expressions)
            delete(state)
        end for
    end for
end for
collect remaining transitions and build union
return string representation of union
```

- Brzozowski Algebraic Method: 2.415 ms

The State Elimination Method demonstrates a 35% decrease in execution time compared to the Brzozowski Algebraic Method. Based on these results, and considering the importance of efficient processing and analysis of log files, the State Elimination Method has been chosen as the primary method for constructing regular expressions in the VTE system. This treatment evaluation and reiteration on the treatment design ensures that the generated regular expressions are both accurate and produced with optimal performance.

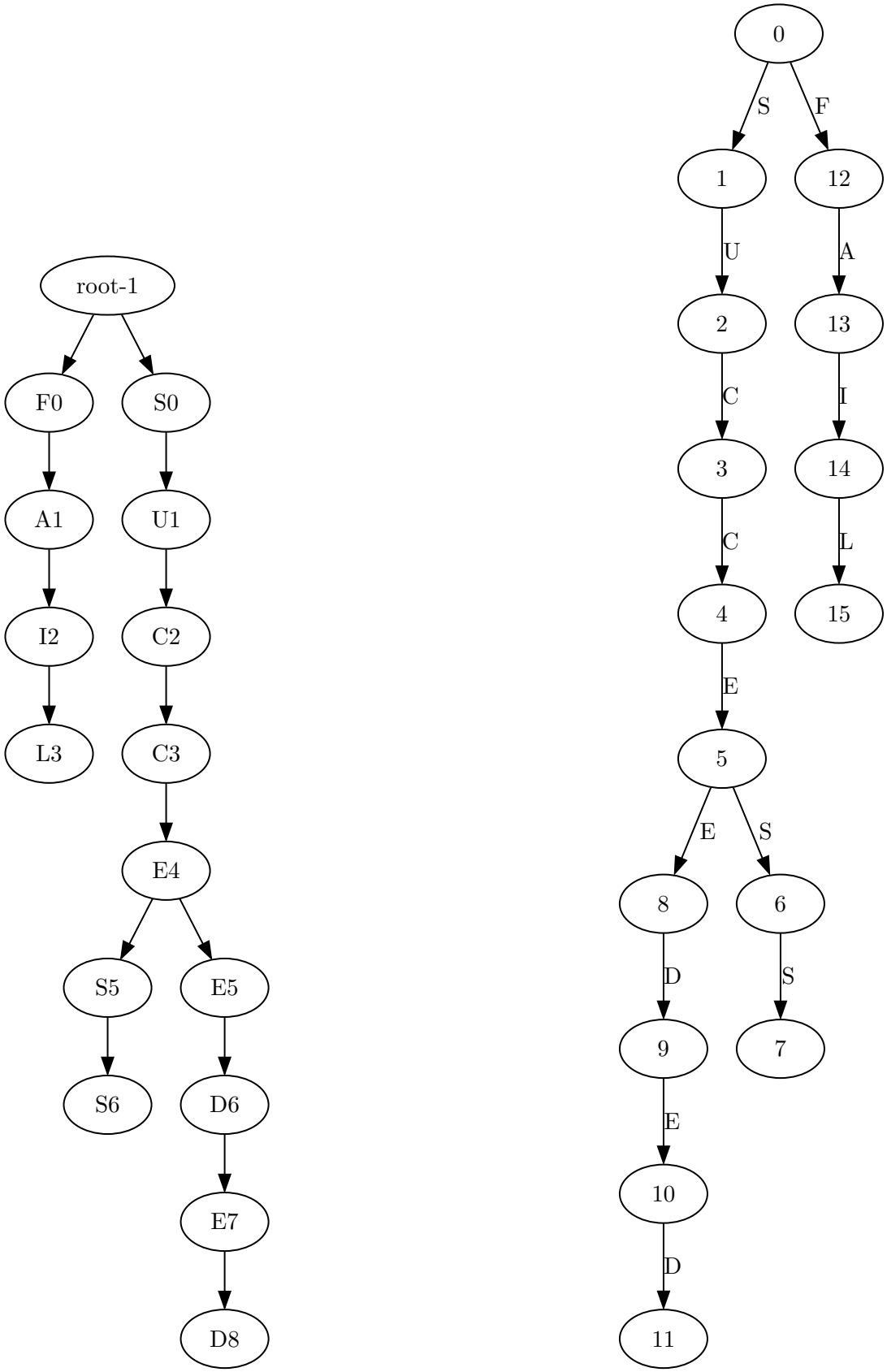
(d) Translate the AST into regular expressions. The translation of the AST to a string representing the regular expression retrieves the last remaining expression. For example, in the case of the Brzozowski Algebraic Method, this is the equation at index zero or the aggregation of all remaining transitions between two remaining states in the State Elimination Method. Finally, the `String` function is called on the struct that implements the `Expression` interface, returning the regular expression as a string.

Specification (2): Implement a configuration mechanic that allows users to

(a) Control whether to generalize digits to a character class. The configuration mechanic, as described in Section 4.2 on page 58 is similarly passed to VTE. A configuration flag `digitsToCharClass` has been implemented. This flag causes VTE to replace each numerical character with \d. \d is the charclass that conforms to 'a decimal digit character' [91].

(b) Control whether to generalize characters to a word class. A configuration flag `digitsToCharClass` has been implemented. This flag causes VTE to replace each alphanumerical character with \w. \w is the charclass that conforms to 'a single alphanumeric character' [91]. The implementation checks this after (a), so an over generalization is avoided.

(c) Specify whether the regular expression generation algorithms should be parallelized. No parallelization opportunity has been discovered given the time–constraint of the thesis. Since this is a non–functional requirement, the design is still valid.



(a) Prefix–tree **(b)** DFA

Figure 4.5: Real–world dynamic tokens from the Hadoop datasets are transformed to regular expressions using a prefix–tree and DFA minimization

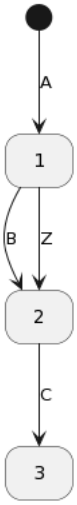


Figure 4.6: An exemplary DFA to demonstrate Brzozowski Algebraic Method

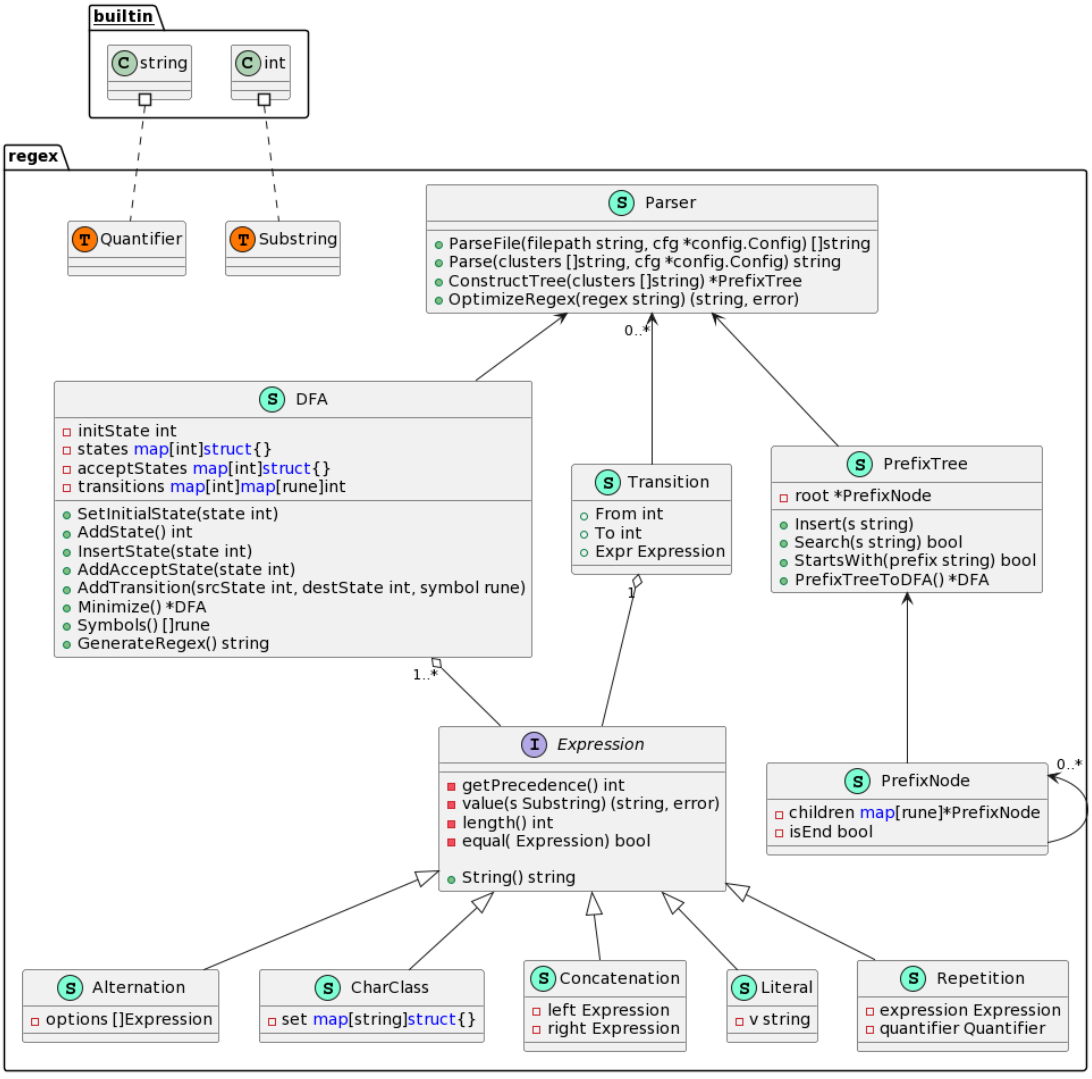


Figure 4.7: UML class–like diagram of the regex package with PrefixTree and Hopcroft minimization

4.6 Parser Benchmark

The benchmark for log parsers in this study is modeled after the work of Zhu et al., which was implemented in Python, and is ported to go to evaluate log parsers implemented in go [10]. The parsing accuracy is evaluated based on four key metrics: precision, recall, F-measure, and accuracy, which are computed for the parsed log events and compared with the ground truth events. The benchmark is a Design Science environment, which is used to evaluate the results of VTEP.

Precision measures the proportion of correctly identified positive instances out of all instances that were identified as positive. It is calculated as $\frac{TP}{(TP+FP)}$, where TP is the number of true positives (correctly identified positive instances), and FP is the number of false positives (instances that were identified as positive but are actually negative).

Recall measures the proportion of correctly identified positive instances out of all positive instances in the ground truth. It is calculated as $\frac{TP}{(TP+FN)}$, where FN is the number of false negatives (instances that are positive in the ground truth but were not identified as positive).

F-measure is the harmonic mean of precision and recall and provides a balanced view of the model performance. It is calculated as $\frac{2*precision*recall}{(precision+recall)}$.

Accuracy measures the overall proportion of correctly classified instances. It is calculated as $\frac{(TP+TN)}{(TP+TN+FP+FN)}$, where TN is the number of true negatives (correctly identified negative instances) and TP, FP, FN are as defined above.

A CLI command `benchmark` was implemented to run the benchmark process. When invoked, this command runs the benchmark using 16 datasets, all of which were collected and made publicly accessible for research purposes by Zhu et al. [10]

Each of the 16 datasets is accompanied by a log format and threshold. The benchmarking process yields outputs for each dataset, including the four metrics mentioned above and the execution time. Additionally, statistical analyses are conducted across all datasets, providing the median, mean, and standard deviation values.

To calculate the metrics, the evaluation function first counts the occurrences of each event in the ground truth and parsed log sequences. It then calculates the total number of possible event pairs for each sequence. For instance, if there are n occurrences of a given event in the ground truth, then the total number of possible pairs for that event is the binomial coefficient n choose 2, which is equivalent to $\frac{n(n-1)}{2}$. The function then identifies the number of accurate event pairs by comparing the parsed log and ground

truth events. If the parsed log contains two or more events with the same ID, and all those events match a ground truth event ID associated with the same line number, then those events are considered accurate.

The combination formula determines the number of ways to choose r items from a set of n items. For example, 2 choose 2 means choosing 2 items from a set of 2 items, which results in 1 possible combination. Similarly, 3 choose 2 means choosing 2 items from a set of 3 items, which results in 3 possible combinations: 1,2, 1,3, and 2,3.

In the log parsing benchmark, the combination formula calculates the total number of possible pairs of events that can occur in the log files. This is important for calculating the accuracy of the parsing algorithm since it provides a baseline for comparison. Suppose the parsing algorithm identifies pairs of events that occur in the log files more or less frequently than would be expected based on the total number of possible pairs. In that case, this can indicate that the algorithm is either over or under-parsing certain events.

The benchmark uses the number of pairs to calculate the precision, recall, F-measure, and accuracy metrics. Precision is the fraction of accurate event pairs in the parsed log sequence out of the total number of possible pairs. Recall is the fraction of accurate event pairs in the parsed log sequence out of the total number of possible pairs in the ground truth sequence. The F-measure is the harmonic mean of precision and recall and measures the balance between the two metrics. Accuracy is the fraction of events in the parsed log sequence that are correctly parsed out of the total number of events in the ground truth sequence.

Consider the following example sequences:

```
groundtruth := []string{
    "E1", "E2", "E3", "E4",
    "E4", "E4", "E3", "E5", "E6"
}
parsedlog := []string{
    "E1", "E2", "E3", "E4",
    "E5", "E5", "E3", "E6", "E7"
}
```

First, the number of occurrences of each event in both sequences is determined. The ground truth sequence is processed in the following manner:

```
[E1]: 1
```

```
[E2]: 1
[E3]: 2
[E4]: 3
[E5]: 1
[E6]: 1
```

In order to enumerate pairs of events, it becomes necessary to compute the number of pairs that can be established for each event with multiple occurrences. This computation is performed using the combinatorial formula n choose 2, where n denotes the frequency of the event. Thus, it is found that:

```
[E3]: 2 choose 2 = 1
[E4]: 3 choose 2 = 3
```

These calculated values are then aggregated to derive the total pairs count in the ground truth sequence, giving a sum of $1 + 3 = 4$.

The same methodology is subsequently applied to the parsed log sequence, leading to the following:

```
[E1]: 1
[E2]: 1
[E3]: 2
[E4]: 1
[E5]: 2
[E6]: 1
[E7]: 1
```

And finally, for the pairs:

```
[E3]: 2 choose 2 = 1
[E5]: 2 choose 2 = 1
```

The number of pairs in the parsed log sequence amounts to $1 + 1 = 2$.

A mapping is subsequently constructed between the ground truth event IDs and the parsed log event IDs. The process begins with an iteration over all parsed events, starting with *E1*. The log line numbers corresponding to *E1* are acquired; in this instance, there is only one *E1*, the first element of the parsed log sequence. This log line number is then

utilized to identify the corresponding event in the ground truth sequence, specifically, groundtruth[0]: *E1*. A counter for *E1* is incremented, indicating that it occurred once in the parsed log sequence. This procedure is replicated for every parsed event.

The corresponding go code:

```
precision   = float64(accuratePairs) / float64(parsedPairs)
recall      = float64(accuratePairs) / float64(realPairs)
f_measure   = 2 * precision * recall / (precision + recall)
accuracy    = float64(accurateEvents) / float64(len(groundtruth))
```

This information gained from the mapping determines the accuracy of a pairing. If only one counter was created for a parsed event ID and the number of log lines in the parsed log sequence is the same as the number of log lines in the ground truth sequence for the corresponding ground truth event ID, then the pairing is considered accurate.

5 Evaluation

Section 5.1 on the next page presents the benchmark results for VTEP. Based on these results, the VTEP design effectiveness is reported. Section 5.2 on page 85 describes the method for collecting the compression measurables that resulted from using CLP with different log files and by providing a custom configuration including the regular expressions generated by VTE. Based on these results, the VTE design is discussed. Section 5.3 on page 88 shows the effects of parsing and compression on the succeeding steps, answering **R2**.

All the experiments were conducted on a machine with 6-core AMD Ryzen 5 1600X 3.6 GHz, 32GB RAM, and an arch–based operating system.

Dataset	LogSig	SLCT	LFA	LogCluster	SHISHO	MoLFI	LKE	LogMine	Spell	AEL	Lenma	Drain	VTE	VTE+	Best
Android	0.548	0.882	0.616	0.798	0.585	0.702	0.911	0.504	**0.919**	0.682	0.88	**0.911**	**0.971**	**0.971***	**0.971**
Apache	0.582	0.731	**1.0***	0.709	**1.0***	**1.0***	**1.0***	**1.0***	**1.0***	**1.0***	**1.0***	**1.0***	**1.0***	**1.0***	**1.0**
BGL	0.227	0.573	0.854	0.835	0.711	**0.947**	0.128	0.723	0.787	**0.957**	0.69	**0.963**	**0.952**	**0.974***	**0.974**
Hadoop	0.633	0.423	**0.9**	0.563	0.867	**0.946**	0.67	0.87	0.778	0.869	0.885	**0.948**	**0.962***	**0.962***	**0.962**
HDFS	0.85	0.545	0.885	0.546	**0.998**	**0.998**	**1.0***	0.851	**1.0***	**0.998**	**0.998**	**0.998**	**0.998**	**0.998**	**1.0**
HealthApp	0.235	0.331	0.549	0.531	0.397	0.442	0.179	0.684	0.639	0.568	0.174	0.78	**1.0***	**1.0***	**1.0**
HPC	0.354	0.839	0.817	0.788	0.325	0.872	0.846	0.784	0.654	**0.903**	0.83	0.887	**0.921**	**0.951***	**0.951**
Linux	0.169	0.297	0.279	0.629	0.672	0.284	0.502	0.612	0.605	0.673	0.701*	0.69	0.363	0.418	**0.701**
Mac	0.478	0.558	0.599	0.604	0.595	0.675	0.369	0.872	0.757	0.764	0.698	0.787	0.814	**0.917***	**0.917**
OpenSSH	0.373	0.521	0.501	0.426	0.619	0.54	0.426	0.431	0.554	0.538	**0.925***	0.788	0.626	**0.925***	**0.925**
OpenStack	0.866	0.867	0.2	0.696	0.722	0.213	0.79	0.743	0.764	0.758	0.732	0.733	**1.0***	**1.0***	**1.0**
Proxifier	0.494	0.518	0.026	0.478	0.517	0.013	0.495	0.517	0.527	0.495	0.517	0.527	0.851	0.856*	**0.856**
Spark	0.544	0.685	0.994	0.799	**0.906**	0.418	0.634	0.576	**0.905**	**0.905**	0.884	**0.92***	**0.91**	**0.91**	**0.926**
Thunderbird	0.694	0.882	0.649	0.599	0.576	0.647	0.813	**0.919**	0.844	**0.941**	**0.943**	**0.955**	0.668	**0.96***	**0.96**
Windows	0.689	0.697	0.588	0.713	0.701	0.711	**0.99**	**0.993**	**0.989**	0.69	0.566	**0.997***	0.412	**0.997***	**0.997**
Zookeeper	0.738	0.726	0.839	0.732	0.66	0.839	0.855	0.688	**0.964**	**0.921**	0.841	**0.967**	**0.995***	**0.995***	**0.995**
Average	0.53	0.63	0.644	0.653	0.678	0.64	0.663	0.735	0.793	0.791	0.767	0.866	0.843	**0.927***	N.A.

Table 5.1: Accuracy of log parsers measured with 16 datasets and 14 parsers

5.1 Analyzing Parsing and Performance of VTEP: Evaluating Accuracy, Efficiency, and Robustness

In order to evaluate the prototype, a real-world setting is emulated using the benchmark. In conjunction with the artifact, this environment is structured into an architecture. As displayed in Figure 5.1, this architecture facilitates observable system behavior. It is composed of components that interact with each other.

System stimulation, for instance, providing log files to VTEP, and processing them using the `benchmark` command, yields a system response, such as the benchmark results. In the following VTE and VTE+ are the names used for the implementations of VTEP. The underlying mechanism is based on the log template extraction framework portrayed in Figure 3.6 on page 49. It includes the steps of *preprocessing*, *identifying*, *extracting*, and *postprocessing*.

The process always involves a set of log records, the *Log*, either as a log file or a stream of log records. The parser component consists of two sub–components, which apply generally as part of a log template extraction parser. This is because each log record must be classified into a cluster or grouping, resulting in a template encompassing a subset of all known log records.

The benchmark yields measurables as outlined in Section 4.6 on page 77, which serve to facilitate further inference.

Table 5.1 presents the accuracy for each dataset. VTE+ incorporates the merging algorithm, distinguishing it from VTE. The values are rounded to three decimal places during data preparation. Due to excessive processing time, the IPLoM parser was unable to complete the android and mac datasets. Although these incomplete data points caused IPLoM to be removed, the dataset's integrity remains intact, especially when comparing VTE with 12 other parsers.

Figure 5.3 displays a boxplot that offers a comparative overview of accuracy distribution across 16 log datasets for each log parser. Each box in the plot represents five key statistical measures: the minimum value (bottom line), the first quartile or 25th

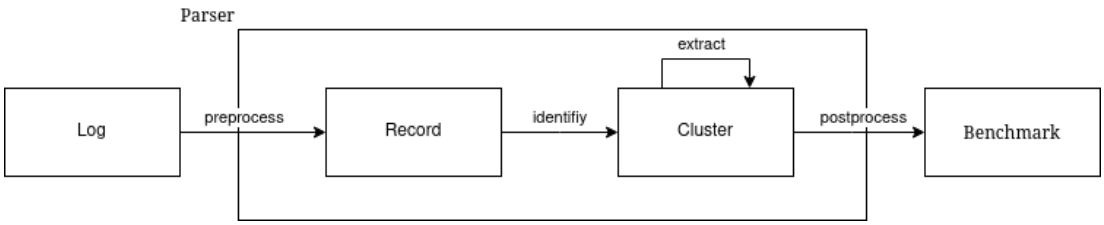
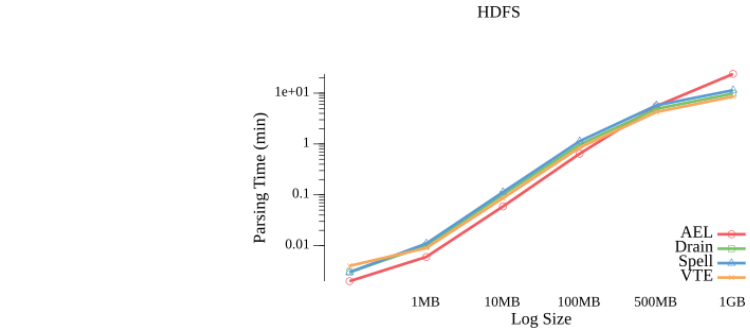
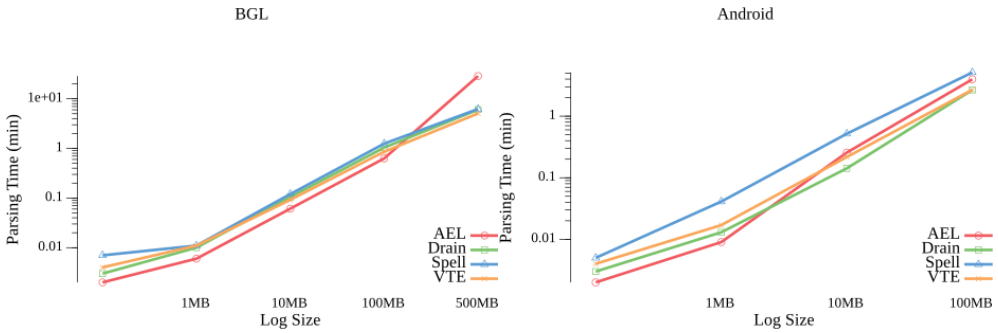


Figure 5.1: Architecture of VTEP evaluation



(a) HDFS



(b) BGL **(c)** Android

Figure 5.2: Effectivness of log parsers based on execution time of the parsing process with increasing log volumne

percentile (lower edge of the box), the median (line inside the box), the third quartile or 75th percentile (upper edge of the box), and the maximum value (top line). Circles denote outliers. The results shown in Figure 5.3 are sorted by mean, providing insights into the design's accuracy and robustness. Figure 5.2 compares the effectiveness of the parsing process by examining the execution time.

High accuracy VTE+ achieves the highest accuracy of all evaluated parsers. With a mean of 92.7% accurately parsed log lines, VTE+ achieves 6.8% higher accuracy on average than Drain, considered one of the most accurate and efficient parsers [10]. VTE+ has a single outlier. The Linux dataset contains a large number of templates. Specifically, the dataset contains dates and usernames in the log body. The dates contain purely alphabetical month and day strings, such as *Jan, Jul* and *Mon, Thu*. The usernames are alphabetical as well. This causes template explosion. The merge algorithm can detect those cases with a high enough threshold defined but over–generalizes other templates in return, reducing the accuracy. Parsing and accurately detecting alphabetical dynamic tokens remains a challenging task.

Low variance of accuracy VTE+ achieves the smallest Interquartile Range (IQR) of all evaluated parsers. The IQR describes the spread of data. A minor variance

across different datasets indicates a robust design. Therefore the design is broadly applicable and can process a wide range of log data.

Effective and scalable VTE+ shows similar behavior to Drain regarding parsing time. VTE+ has slightly higher parsing times on small data sizes (<1MB) but scales well on bigger data sizes, achieving the fastest parsing on the biggest evaluated datasets HDFS 1GB, BGL 500MB, and Android 100MB. Effectively parsing about 7 million log lines in 8:30 minutes in the HDFS dataset, compared to Drain with slightly below 10 minutes, Spell with about 11:20 minutes, and AEL with 18:15 minutes. It has to be noted, that the current implementation is Random Access Memory (RAM) limited.

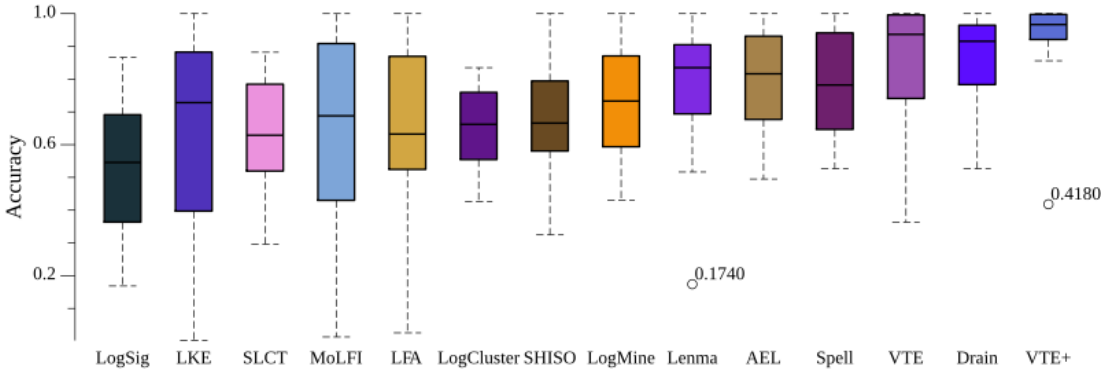


Figure 5.3: Among 12 other parsers, VTE has a competitive mean. VTE+ achieving the highest mean. Sorted by mean.

5.2 Analyzing the Impact of Parsing and Custom Regular Expressions on Log Compression

An evaluation architecture is presented, as shown in Figure 5.4, to evaluate the VTE design and its contribution to helping CLP operators to define regular expressions for the compression process. The system–prototype combines all presented artifacts to evaluate the environment, e.g., CLP.

The experiment consists of two parts. The first part uses CLP's default mode. The default mode compresses the target log file without providing a configuration file. The second part uses CLP's configuration interface. CLP takes a configuration file, which enables the CLP operator to define *delimiters*, *timestamps*, and custom regular expressions. The configuration file is adapted for each log file. VTE provides custom regular expressions based on the parsing results. These custom regular expressions represent dynamic tokens, which are hard to detect for parsers, such as dynamic tokens that consist solely of alphabetical characters. The sizes of the resulting archives are measured. Because CLP only supports a non—standard subset of regular expressions, the expressions resulting from VTE are modified to conform to the interface without changing the expressed language.

For example, the following example shows the modifications. CLP cannot process non-capture capture groups, and the | operations need the left and right expression to be encapsulated by a capture group.

```
1    Dynamic2:get(?:RunningAppProcesse|Task)s
2    Dynamic2:(getRunningAppProcesses)|(getTasks)
```

Listing 5.1: Data preparation for CLP configuration

If VTE could not identify any dynamic tokens conforming to (1) at least five occurrences across log records and (2) more than one character in the dynamic token, then the dataset result is described with *N.A.*. This requirements aim to catch the most occurring alphabetical dynamic tokens assumed to have the most impact on log compression. The results for each dataset are shown in 5.2 on page 87. The key observations are as follows:

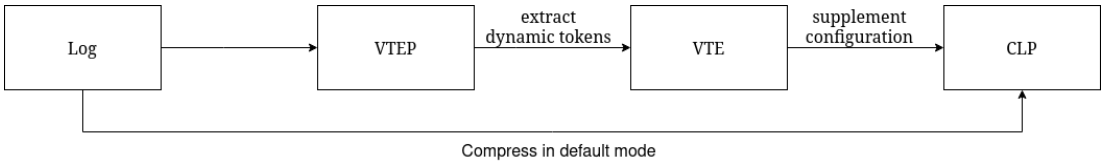


Figure 5.4: Architecture of VTE evaluation

Timestamps Accurately detecting the timestamp format is of great importance in the resulting compression ratio. CLP is able to handle the most common and simple to parse timestamp formats, such as `3/31/2014` or `2013-04-01 22:43:22`. Parsing timestamps where the format cannot be consistently detected, such as `Jul 1 09:00:55` found in the Mac dataset or `Jun 25 19:25:31` and `Jun 25 19:25:31 Jul 1 00:21:28` in the Linux dataset leads to unexpected behavior. This behavior is manifested by identifying multiple log lines as a single one, which reduces compression ratio, such as 9% less compression ratio in the Linux data set.

Alphabetical Dynamic Tokens Identifying the tokens that cause template explosion is a main concern for optimal compression. CLP is good at detecting dynamic tokens containing numbers, but has a hard time identifying alphabetical dynamic tokens. An example is the Thunderbird dataset, where log records containing alphabetical dynamic tokens is shown in Listing 5.2. For each log record with the `LNKX` where `X` is any uppercase letter, a separate template is created. By correctly identifying these occurrences, the compression can be improved. VTE is able to detect them partially. In particular the occurrences in line 4-8. Resulting to the regular expression `LNK(?:([DF])|[EG-H])`. Partially because `LNKA`, `LNKB`, `LNKC` being part of another template, not including the suffix `", disabled"` and therefore not conforming to the dynamic tokens inclusion requirement (1) having more than five occurrences in all log records.

```
1 [...] ACPI: PCI Interrupt Link [LNKA] [...]
2 [...] ACPI: PCI Interrupt Link [LNKB] [...]
3 [...] ACPI: PCI Interrupt Link [LNKC] [...]
4 [...] ACPI: PCI Interrupt Link [LNKD] [...], disabled
5 [...] ACPI: PCI Interrupt Link [LNKE] [...], disabled
6 [...] ACPI: PCI Interrupt Link [LNKF] [...], disabled
7 [...] ACPI: PCI Interrupt Link [LNKG] [...], disabled
8 [...] ACPI: PCI Interrupt Link [LNKH] [...], disabled
```

Listing 5.2: Truncated snippet of the Thunderbird dataset showing alphabetical dynamic tokens

While the default mode of CLP results in excellent compression ratio in general with good usability, the usability of the configuration interface of CLP is improvable, especially by letting CLP handle the timestamps detection similar to the default mode and improving the expressiveness of custom regular expressions.

These observations and their corresponding problems for log compressors are independent of CLP. While VTE can detect some of these alphabetical dynamic tokens and support log compression tool operators by providing a custom regular expression generated by VTE, accurately identifying all these tokens remains challenging. The defined regular expression needs to lead to a template merge, which causes only the variable to be saved

in the variable dictionary without creating a log template for each variable, therefore preventing template explosion. While CLP can prevent such template explosions, e.g., file system paths, preventing template explosions for tokens representing usernames or states with high occurrence counts is unsolved.

As shown in Table 5.2 VTE can help operators to fine–tune their configuration for log compression tools, which is dependent on the underlying data. Custom and VTE share the same configuration, except the additional custom regular expressions provided by VTE.

Dataset	Original	Default	Custom	VTE	% Custom to VTE	% Default to VTE
Android	277077	20610	11319	11305	0.12	45.14798
Apache	169240	8662	9343	N.A.	N.A.	N.A.
BGL	315151	75511	75486	N.A.	N.A.	N.A.
Hadoop	382949	6217	7286	7105	2.484	-14.28
HDFS	285848	72613	71465	N.A.	N.A.	N.A.
HealthApp	185457	14611	28076	N.A.	N.A.	N.A.
HPC	149178	4389	5338	5290	0.899	-20.52
Linux	214486	6474	15301	15312	0.0718	-136.515
Mac	317415	55371	54374	52954	2.611	4.365
OpenSSH	223217	3278	3270	3237	1.009	1.250
OpenStack	593120	43348	66322	66416	-0.1417	-53.215
Proxifier	236962	26825	31719	26589	16.173	0.879
Spark	194268	3327	9265	N.A.	N.A.	N.A.
Thunderbird	323193	43956	43783	43838	0.1256	0.26845
Windows	283434	10703	10848	10799	0.45169	-0.8969
Zookeeper	277892	4458	8434	4391	47.9369	1.5029

Table 5.2: Compression of 16 datasets with CLP. The columns Original, Default, Custom, and VTE show compressed file sizes in bytes. Custom to VTE and Default to VTE show the percentage difference, where positive values indicate a compression improvement and negative values a decrease.

5.3 Investigating the Impact of Parsing and Compression on Log Data's Processability

To research how applying parsing and compression techniques on log data affect its further processability, I analyze the subdomains this questions frames. The first domain is the results of the parsing process, which gets passed to compression process. The parsing quality measured by the accuracy metric does impact the compression ratio.

The second domain is the resulting archive of the compression process. These archive do have some characteristics, i.e., compression size, decompression speed, partial decompression. Both investigations lead to concluding answer **R2**

To evaluate the impact of parsing and compression techniques on log data's processability, an analysis is conducted on the relevant subdomains. The first domain concerns to the outcomes of the parsing process, which are subsequently passed into the compression process. The influence of the accuracy of parsing on the compression ratio is shown.

The second domain concerns the resulting archives produced by the compression process. These archives show various characteristics, such as compression size and the ability for partial decompression. Both lines of investigation contribute to providing a conclusive answer to the question posed by **R2**.

	Revised			VTE+			Drain		
Dataset	Size L2	Size L3	Match	Size L2	Size L3	Match	Size L2	Size L3	Match
Android	27146	32221	100.0%	28254	33738	99.9%	28325	33527	96.7%
Apache	6665	10182	100.0%	6670	10182	100.0%	6231	9677	100.0%
BGL	48214	58416	100.0%	48747	58951	99.7%	45041	54673	100.0%
Hadoop	24796	31460	100.0%	25245	31002	100.0%	20527	23734	83.2%
HDFS	47202	62946	100.0%	50898	66025	94.7%	47056	63057	100.0%
HealthApp	14854	20750	87.6%	15253	21223	87.6%	15048	19069	87.6%
HPC	23995	26494	100.0%	23975	26274	99.5%	24000	26232	99.5%
Linux	19172	22311	100.0%	20116	23456	100.0%	20108	23451	100.0%
Mac	65594	78128	96.7%	71317	83771	96.9%	55243	66622	90.0%
OpenSSH	11315	15547	100.0%	16825	21628	100.0%	11264	14916	100.0%
OpenStack	54835	68385	100.0%	52974	61190	49.1%	56593	64252	49.1%
Proxifier	21572	26850	84.9%	18921	20874	57.8%	20841	23903	70.8%
Spark	10249	15891	100.0%	10150	15847	100.0%	10635	16312	100.0%
Windows	12245	16587	99.9%	13084	17128	99.4%	11888	16023	100.0%
Zookeeper	23189	28498	100.0%	23086	28398	100.0%	22607	28016	100.0%

Table 5.3: Compression of 15 datasets with Logzip [17]. Three log parser results are passed to Logzip. The revised template files are found in the LogParser project [10]. L2 and L3 are Logzip compression modes, where L2 uses a template extraction file to match log records. L3 additionally creates a dynamic token encoding. The size is in bytes. The match column shows the percentage of log records that matched a log template.

5.3.1 Parsing

Logzip is operated with three sets of log template files, and the compression results are displayed in their respective column in Table 5.3 on the preceding page. The Revised set serves as the ground truth, which the templates can be found in the logparser project [10]. The VTE+ set represents the template files generated by the parser introduced in this thesis, while the Drain set corresponds to the template files generated by the Drain parser [14].

Logzip operates in three modes of which only L2 and L3 are relevant. The L2 mode employs the templates to extract dynamic tokens from the log records, while the L3 mode goes a step further by encoding the variables with a sequential 64-base number and storing the mapping in a file [17]. The match column indicates the percentage of log records matched with a template.

Contrary to the initial assumption that the templates should achieve a 100% match rate for all datasets, since they were generated based on the corresponding dataset, the Logzip tool encountered difficulties in some instances with specific datasets. For example, it struggled with whitespace at the beginning of a log template for the HealthApp dataset.

Figure 5.5 on page 91 illustrates the architecture employed to generate these measurements. The datasets are relatively small, approximately 300KB, which explains the limited effectiveness of the L3 mode, as creating a parameter mapping file does not provide significant benefits, and the mapping overhead outweighs the gain in compression ratio.

Furthermore, datasets with generally lower matching rates, such as the OpenStack, Proxifier, and Hadoop datasets, tend to achieve better compression than those with higher matching rates.

Differences in compression ratios can be observed in datasets where all parser templates achieve a 100% match rate, such as Apache, Linux, OpenSSH, Spark, and Zookeeper. Analyzing the Apache results reveals that Drain's templates are more specific than the Revised and VTE+ templates. This results in additional variables being stored in separate files with the Revised and VTE+ templates rather than directly referencing the log record to the template, as in the case of the Drain templates. While the Revised and VTE+ templates are more versatile in matching records and their dynamic tokens, being more specific, i.e., not recognizing potential dynamic tokens, can lead to a better compression ratio. As a result, fewer wildcards are present while still achieving a 100% match rate for the log records.

On the contrary, when examining the Spark dataset, VTE+ contains 57 wildcards with 35 templates, whereas Drain contains 29 wildcards with 29 templates. Although Drain achieves higher overall accuracy than VTE+ on the Spark dataset, as shown in Table 5.1 on page 81, VTE+ yields a better compression ratio.

While the accuracy metric commonly used in the literature is valuable for comparing sequences of log events against ground truth, the data indicate that unknown factors influence the effectiveness of templates for log compression.

Comparing the VTE+, Revised, and Drain templates shown in Listing 5.4 for the log displayed in Listing 5.3 provides insights into the quality of a template for compression. Logzip further splits tokens identified by a wildcard, which leads to redundant data being stored. Although the Drain template is more accurate than the VTE+ templates, the VTE+ templates achieve higher compression with Logzip. In the case of Revised and Drain, the token `attempt_` is redundantly saved as a dynamic token, while VTE+ includes this information in the template itself.

Therefore, balancing generalization and specialization is crucial in determining the quality of log templates for log compression. A *high–quality* template is as specialized as possible while maximizing the matching coverage. The quality of log templates utilized by the compressor is a determining factor in the resulting compression ratio.

Specialization also encompasses the encoding of data types. For this data types would need to be identified, which remains challanging.

For instance, a single float value can be stored more efficiently than two integers delimited with a dot.

```
1    [...] attempt_201706092018_0024_m_000004_1026: Committed
2    [...] attempt_201706092018_0024_m_000002_998: Committed
3    [...] attempt_201706092018_0024_m_000003_1012: Committed
4    [...] attempt_201706092018_0024_m_000001_984: Committed
5    [...] attempt_201706092018_0024_m_000000_970: Committed
```

Listing 5.3: Snippet of Spark log records

```
1    Drain:
2        <*>: Committed
3    VTE+:
4        attempt_<*>_<*> Committed
5        attempt_<*>_<*>_m_<*>_<*>: Committed
6    Revised:
7        <*> Committed
```

Listing 5.4: Templates for a specific spark log record

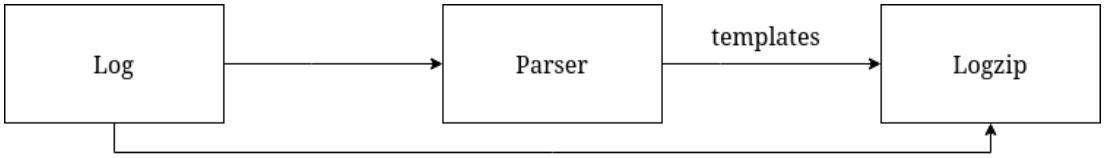


Figure 5.5: Architecture of parser results evaluation using Logzip as compressor.

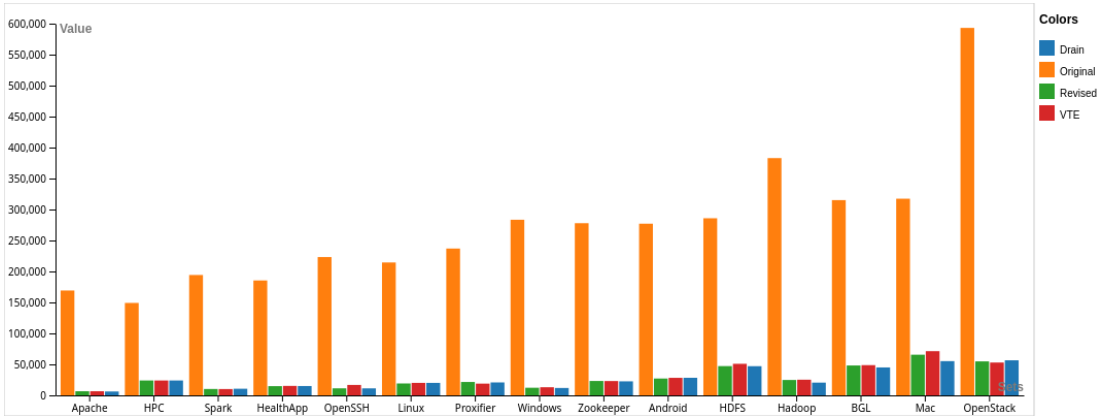


Figure 5.6: Logzip compression with templates from VTE+, Drain, and Revised compared to the original file size. Values in bytes.

5.3.2 Compression

The following section investigates the impact of various log compression techniques on resulting archive files. Figure 5.7 presents a comparison between CLP, Logzip with revised templates, and zstd, a general-purpose compressor.

Among the 16 datasets evaluated, zstd achieves the lowest compression ratio on 7 datasets, while Logzip performs the worst on 6 datasets. The Thunderbird dataset did not finish compressing, resulting in the lowest compression ratio. CLP achieves the lowest compression ratio on 3 datasets.

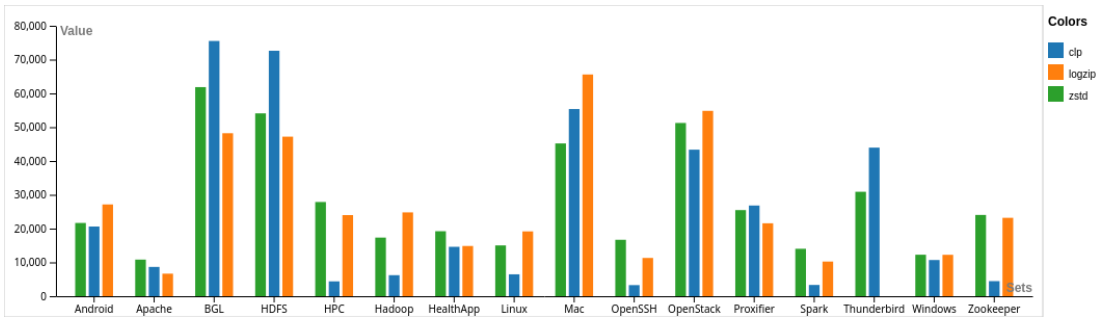


Figure 5.7: Comparison of three compressors. Logzip and CLP are log–specific compressors. zstd is a general–purpose compressor. Values in bytes.

CLP encounters difficulties in correctly identifying the timestamp in the Apache, BGL, HDFS, HealthApp, Mac, OpenStack, Proxifier, and Thunderbird datasets. Partial identification of the timestamp is observed in the Linux dataset. As discussed in Section 5.2 on page 85, the timestamp plays a significant role in CLP's compression process. In datasets where the timestamp is successfully detected, CLP outperforms logzip and zstd in terms of compression ratio as shown in Figure 5.7 on the preceding page. By combining log–specific compression techniques with a heuristic-based extraction approach and a general-purpose compressor, CLP leverages the benefits of both methods.

Therefore, the superior approach for log data compression is to combine log–specific compression techniques with a general-purpose compressor. Based on the results depicted in Figure 5.5 on the previous page, the accuracy of log templates has a negligible impact on the final archive size. As long as the majority of templates are identified and dynamic tokens are detected, the resulting compression ratio averages around 90% relative to the original file size in Logzip compression.

These findings suggest that a trade-off can be made in the accuracy of log parsers to prioritize efficiency and robustness. Additionally, as demonstrated in the CLP paper [23], log file-specific compression offers the advantage of constructing searchable archives for executing search queries using log templates and dynamic token dictionaries.

In conclusion, these results highlight the effects of log parsing techniques on the compression process and explain the characteristics of resulting archives, effectively answering **R2**. Furthermore, in regard to **R3**, CLP can be recommended over Logzip, as an efficient, accurate and robust log data compression tool, which in default mode is straightforward to use, achieving high compression ratios and faster search results than comparable tools [23].

5.4 Evaluation of Research Methods

The SLR contributed to the research and investigation goals **O1**, and **O2** defined in 1.2 on page 2. It helped to distinguish parsing, compression, and search tooling, which led to a differentiated analysis of effects, described in Section 5.3 on page 88, effectively contributing to answering **R2** and **R3**. It provided a basis for analyzing **O3** and **O4**.

Considering the resource–constraints, such as time, the SLR provided more studies than this thesis could detail in–depth. Therefore only a selection of techniques and tools are evaluated. The SLR could have included more literature sources, which potentially would have increased the number of relevant studies. But considering the constrains, this would have exceeded the defined scope in 3.1.2 on page 22. Nevertheless, the summary of these findings contributes to complementing a comprehensive overview of

the field. Therefore, the SLR was a necessary method, which successfully contributed to the research goals.

The design science methodology, as a school of thought for research in computer science, contributed to plan, structuring, and realizing the design, implementation, and evaluation of the concepts introduced in Chapter 3 on page 21 and 4 on page 57 and evaluated in Chapter 5 on page 81. In particular, the engineering cycle shown in Figure 3.4 on page 36, guided the process. The iterative approach contributed to exploring design choices and helped make informed decisions based on the implementation evaluation resulting from the implemented treatments, which was vital for realizing **O4**. Conducting a benchmark evaluation as part of the treatment validation process facilitated ongoing improvements in the treatment design. However, the same time constraint as in the SLR applies in the design science approach. A more rigorously planned and designed research setup, especially for the VTE evaluation, could have helped to strengthen present arguments further or discover more insights. For example, the reproducibility of the VTE data could have been achieved by designing a benchmark. The current state of reproducibility is limited to a benchmark script, which its results require manual data preparation to result in the data shown in Section 5.2 on page 85 and Section 5.3 on page 88.

Overall the research methods contributed to fulfilling the research objectives and were vital to answering the research questions.

6 Conclusion

In this thesis, the research challenges surrounding log parsing and compression were framed and investigated, with a focus on the implications for the compression process and resulting compressed data. The background of log data and its usage, including log analysis, was provided in Chapter 2 on page 7.

Chapter 3 on page 21 addressed the need for log data parsing and introduced the design problems related to log parsers and regular expression generators. To inform the design, a SLR was conducted in Section 3.1 on page 21, exploring various log data processing strategies that aim to parse and compress data while utilizing the resulting archive structure for efficient search.

The implementation details were presented in Chapter 4 on page 57, which discussed different implementation choices, such as data structures and algorithms, and their performance implications. To validate the design, a benchmark was performed, measuring the accuracy of the parser.

Chapter 5 on page 81 showcased the benchmark results, demonstrating the competitiveness of the design compared to 12 other parsers. Additionally, the parser results were utilized to identify alphabetical dynamic tokens that could enhance the log compressor's configuration, showcasing the effects of integrating them into the compression process. The impact of log templates derived from different log parser results on the compression ratio was discussed.

Finally, a comparison of compression tools was conducted, leading to a recommendation for a generally applicable log compression tool.

With this thesis, the three central research questions were addressed:

> *R1: How can the structure of log data be exploited to reduce storage requirements through compressing while maintaining the ability to query the data?*
>
> *R2: How do parsing and compressing log data affect its further processability?*
>
> *R3: Is there a log data compression and query technology that practitioners can use that is widely applicable while minimizing additional architectural complexity, computational requirements, and storage space?*

To address **R1**, the research question is divided into three components. The first component focuses on the structure of log data and its potential for exploitation. The comprehensive description of log data structure is achieved by analyzing 16 different datasets and their mapping to the OpenTelemetry log data model. The log template extraction framework (see Figure 3.6 on page 49) and the techniques identified (refer to 2 on page 54) are applied in the presented design to leverage the log data structure for further processing. The second component pertains to log data compression. Log data compressors such as Logzip and CLP can utilize the identified log structures, specifically log templates, to efficiently compress the data. This is demonstrated through the results in Table 5.2 on page 87 and Figure 5.6 on page 91. The third component addresses the ability to query the compressed log data, allowing for searching with specific phrases. Among the identified log compressors, only CLP provides this functionality. Considering these three components, the research question **R1** regarding exploiting log data structure for reducing storage requirements through compression while maintaining queryability is comprehensively addressed.

By utilizing the implementation and insights gained from the literature review, the research question **R2** is addressed through an examination of the effects of parsing results on the compression ratio. This analysis focuses on a representative log compressor that employs dictionary-based compression. Figure 5.5 on page 91 and Table 5.3 on page 88 illustrate the observed variance in log compression outcomes. Notably, the differences in compression ratios relative to the original file size were found to be minimal. These findings suggest that a trade-off can be made, placing equal importance on the efficiency and robustness of log parsers alongside the pursuit of optimal accuracy.

The comparison presented in Figure 5.7 on page 91 highlights the difference in compression ratios between general–purpose compressors like zstd and specialized dictionary-based log compressors such as Logzip and CLP. The experimental results indicate that, in general, specialized log compressors outperformed general-purpose compressors, with only 2 cases out of the 16 considered showing different outcomes. Notably, when all positively impacting factors were present, such as accurate identification of the timestamp format and utilization of high-quality templates, a significant improvement in compression ratio was observed. Based on these results and with regard to the research question **R3**, CLP can be recommended as a technology with high compression–ratios, fast compression, and query capabilities.

Almost all corresponding objectives described in 1.2 on page 2, which were bases for the question, could be achieved. However, **O3** could only be partially fulfilled. The aim described a performance analysis of all researched parsers, but this turned out to be an overestimation of doable work. This is partially fulfilled by the results shown in Section 5.1 on page 82 and Section 5.2 on page 85 because all the tools evaluated are a subset of the found tools presented in Section 3.1 on page 21.

This thesis makes several contributions to the field of log parsing and compression. It provides a SLR that explores the field of log analysis, specifically focusing on log parsing, compression, and the ability to query compressed data. In addition, this review provides an overview of the field's current state. The thesis presents a novel log parser design demonstrating high accuracy, efficiency, and robustness. The effectiveness of this design is validated through two benchmark evaluations, supporting its practical applicability. Additionally, the thesis proposes an innovative approach for identifying purely alphabetical variable tokens, which can be utilized to fine-tune the configuration of log compressors like CLP. This approach contributes to optimizing the compression ratio by using purely alphabetical variable tokens, which the investigated log compressors struggled with. Furthermore, the thesis investigates the effects of parsing results on log compression. Analyzing and comparing the compression results provides insights into the relationship between parsing results and their impact on compression. Lastly, the thesis explores the impact of log compression techniques on the resulting compressed data. Through evaluations, it shows the benefits and limitations of different compression methods. It also shows the overall effectiveness of log compression techniques by comparing three compressors.

6.1 Validity

This section details the validity concerns and countermeasures taken.

The conclusion validity is concerned with the relationship between treatment and outcome. By using well–studied datasets for all experiments, without excluding any of them, the validity threat of unconsciously fishing for good results is avoided. This is further reinforced by using two benchmark implementations, increasing the reliability of measures. The conclusions are based on statistical observations and measurements, increasing the conclusion's validity. The elements that may have disturbed and introduced noise in the experiments are present, but countermeasures like using the same machine have been taken. Additionally, the experiments with sensible execution times are run multiple times, where the mean is used to counter these noisy results. The experiments that took a long time, i.e., more than 5 minutes, were only run once, which weakens the conclusion's validity.

External validity describes the generalization or whether the cause-and-effect relationship is valid when applied to a different population, e.g., log data, log parsers, or log compressors. This thesis does not claim the investigated datasets to represent all log data but still cover a wide–range of different log formats and origins. The 16 datasets the experiments conducted with are diverse in the context of their origin and internal structure. The logs originated from distributed systems (HDFS, Hadoop, Spar,

Zookeeper, OpenStack), supercomputers (BGL, HPC, Thunderbird), operating systems (Windows, Linux, Mac), mobile systems (Android, HealthApp), server applications (Apache, OpenSSH) and standalone software (Proxifier). Therefore the logs cover many systems, increasing the external validity. Although unusual, logs can have multiple formats within the same file. However, the analyzed logs did not exhibit such a scenario in this study. Furthermore, this thesis acknowledges numerous additional scenarios beyond the scope of its coverage, including different log data formats, timestamp formats, complex templates, or an even larger number of templates.

The internal validity is strengthened by the selection of well–studied datasets. Furthermore, using two benchmark implementations that produced consistent results adds to the strengthening of internal validity. The results of both benchmarks were compared with the results of other studies, such as by Zhu et al. [10] or Fu et al. [68], where the Parsing Accuracy (PA) is used to measure the effectiveness. However, it is essential to consider potential threats to internal validity, such as historical events. The 16 datasets and their ground-truth templates are stored in a GitHub repository. Since their initial release in 2019, the ground-truth templates have been updated once, which may affect comparability with previous studies. Therefore the current versions of the datasets could still contain errors unknown so far, which potentially had unknown effects on this thesis. This revision had a minimal impact on the previous results, resulting in a change of less than one percentage point compared to the original result, which in the future could be revised again. The results of the CLP compression, provisioned with the alphabetical dynamic tokens generated by VTE, strongly depend on the configuration file's correctness. This was mitigated by comparing the results against a baseline consisting of the same configuration file lacking the custom regular expressions. The SLR relied on only two research databases, which increases the possibility of missing relevant kinds of literature not present in those databases. Additionally, literature such as proposals not listed in the databases was ignored, potentially resulting in the omission of valuable knowledge. Moreover, there is a chance that relevant papers were overlooked during the search of the databases using the defined queries. The review was conducted solely by the author, without the involvement of external experts, which introduces a potential threat to internal validity. To mitigate the risk of missing relevant papers, a snowballing methodology is applied (refer to 3.1.2 on page 23). Another validity threat is that the content of the papers might be misunderstood, and the resulting analysis, i.e., the description of the tools and designs of the corresponding research papers, is flawed. The source code was only partially and superficially checked, which mainly focused on the parsers used for the accuracy comparison in Table 5.1 on page 81. A complete check would have required more resources.

6.2 Related Work

There are two surveys and a design proposal for a log parser that are closely related to this thesis. El–Masri et al. have conducted an SLR, investigating log parser techniques, without the consideration of log compression and the search on compressed data, but most of investigated parser techniques overlap with the techniques investigated in this thesis [9]. They presented a quality model to measure parsing techniques. They also identified and recommended research directions for log parsing. Chen et al. presented a broad overview of the field of log analysis, with a focus on log parsing, compression, and mining [44] in the context of reliability engineering. They described the characteristics, challenges, and a selection of available techniques and tools but did not compare compression ratios or the impact of parsing results on the compression process. Additionally, the parser clustering method by Sedki et al. is used in the implementation of VTEP [22]. This method provides a simple but efficient clustering method, which is extended and modified by the aspects described in Section 4.3 on page 60.

6.3 Further Research

The design of the parser currently operates in offline mode. Further research could be conducted to adapt to an online mode, enabling the processing of stream–like log data sources, which could be combined with a log compressor utilizing the results. This could lead to a design for an end–to–end log analytics or compression platform.

Additionally, a case–study in an industrial setting could be helpful to investigate the applicability of the design in a different and realistic context, such as anomaly detection, other than log compression. This would also increase the external validity by applying the concept to a different context.

Further research could be conducted on the implications of intersecting regular expressions provided to log compressors. Initial observations suggest that these regular expressions may match unwanted tokens, causing them to be included in the variable dictionary. This, in turn, leads to over–generalization of log templates. However, due to time constraints, no comprehensive research could be conducted on this topic.

A comprehensive comparison of the performance of searching compressed log data, including dictionary–based compressors, could be conducted. This thesis only investigates three compressors and their compression ratios without investigating the search performance, i.e., decompressing and applying the search term, and the difference between the need to completely decompress the archive and being able to partially decompress.

This could lead to a more comprehensive understanding of the log archives and their benefits and limitations.

6.4 Perspective

In the short term, designing problem–solving solutions utilizing state–of–the–art log parsing techniques could consolidate and prove the usefulness of these techniques. This could be in the domain of fault- and anomaly detection. Unfortunately, these solutions often only make sense in enterprise environments where highly complex systems are deployed. An easy–to–use log archiving solution whose sole purpose is to reduce the storage requirements and cost while conforming to regulations, such as security compliance requiring a certain period of log availability, could be an option with a low entry barrier. This needs to consider log data transmission with either uncompressed or compressed data or log data aggregation functionalities based on compressed data. In the long term, existing log management platforms could adapt to this technology to reduce the cost of storage while maintaining the ability to perform search on the stored data. While services, such as Elasticsearch, Datadog, or Splunk, use inverted indices to enable fast search on data, the downside of it is the storage overhead inverted indices cause. It would be beneficial if the log parsing and compression technology would prove its usefulness and experience adaptation so a more efficient alternative for log data management would be available.

Bibliography

[1] B W Kernighan and R Pike. *The Practice of Programming*. Addison-Wesley professional computing series. Addison-Wesley, 1999. ISBN: 9780201615869. URL: https://books.google.de/books?id=j9T6AgAAQBAJ.

[2] Jeanderson Candido, Mauricio Aniche, and Arie Van Deursen. 'Log-based software monitoring: a systematic mapping study'. In: *PeerJ Computer Science 7:e489* (). DOI: 10.7717/peerj-cs.489.

[3] Pinjia He et al. 'Towards Automated Log Parsing for Large-Scale Log Data Analysis'. In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (2018), pp. 931–944. DOI: 10.1109/TDSC.2017.2762673.

[4] Jack Lou and Devesh Agrawal. *Reducing Logging Cost by Two Orders of Magnitude using CLP | Uber Blog*. Sept. 2022. URL: https://www.uber.com/en-DE/blog/reducing-logging-cost-by-two-orders-of-magnitude-using-clp/.

[5] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 'Characterizing logging practices in open-source software'. In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, June 2012, pp. 102–112. ISBN: 978-1-4673-1066-6. DOI: 10.1109/ICSE.2012.6227202.

[6] Polly Traylo. *State of Unstructured Data Management Report*. Tech. rep. Komprise, 2022. URL: https://www.komprise.com/blog/2022-survey-the-top-five-trends-in-unstructured-data-management/.

[7] Max Landauer et al. 'System log clustering approaches for cyber security applications: A survey'. In: *Computers and Security* 92 (May 2020). ISSN: 01674048. DOI: 10.1016/J.COSE.2020.101739.

[8] Shubham Jain, Amy De Buitleir, and Enda Fallon. 'A Review of Unstructured Data Analysis and Parsing Methods'. In: *2020 International Conference on Emerging Smart Computing and Informatics, ESCI 2020* (Mar. 2020), pp. 164–169. DOI: 10.1109/ESCI48226.2020.9167588.

[9] Diana El-Masri et al. 'A systematic literature review on automated log abstraction techniques'. In: *Information and Software Technology* 122 (June 2020). ISSN: 09505849. DOI: 10.1016/j.infsof.2020.106276.

[10] Jieming Zhu et al. 'Tools and Benchmarks for Automated Log Parsing'. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2019, pp. 121–130. DOI: 10.1109/ICSE-SEIP.2019.00021.

[11] Barbara Kitchenham and Stuart M Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. 2007. URL: https://www.researchgate.net/publication/302924724.

[12] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8.

[13] Byung Chul Tak et al. 'LOGAN: Problem Diagnosis in the Cloud Using Log-Based Reference Models'. In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, Apr. 2016, pp. 62–67. ISBN: 978-1-5090-1961-8. DOI: 10.1109/IC2E.2016.12.

[14] Pinjia He et al. 'Drain: An Online Log Parsing Approach with Fixed Depth Tree'. In: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, pp. 33–40. DOI: 10.1109/ICWS.2017.13.

[15] Yu Bai, Yongwei Chi, and Dan Zhao. 'PatCluster: A Top-Down Log Parsing Method Based on Frequent Words'. In: *IEEE Access* 11 (2023), pp. 8275–8282. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3239012. URL: https://ieeexplore.ieee.org/document/10024775/.

[16] Ran Tian et al. 'LogDAC: A Universal Efficient Parser-based Log Compression Approach'. In: *ICC 2022 - IEEE International Conference on Communications*. 2022, pp. 3679–3684. DOI: 10.1109/ICC45855.2022.9838258.

[17] Jinyang Liu et al. 'Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression'. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 863–873. DOI: 10.1109/ASE.2019.00085.

[18] Weibin Meng et al. 'LogParse: Making Log Parsing Adaptive through Word Classification'. In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. 2020, pp. 1–9. DOI: 10.1109/ICCCN49398.2020.9209681.

[19] Min Du and Feifei Li. 'Spell: Online Streaming Parsing of Large Unstructured System Logs'. In: *IEEE Transactions on Knowledge and Data Engineering* 31.11 (2019), pp. 2213–2227. DOI: 10.1109/TKDE.2018.2875442.

[20] E. Ukkonen. 'On-line construction of suffix trees'. In: *Algorithmica* 14.3 (Sept. 1995), pp. 249–260. ISSN: 0178-4617. DOI: 10.1007/BF01206331.

[21] D. S. Hirschberg. 'A linear space algorithm for computing maximal common subsequences'. In: *Communications of the ACM* 18.6 (June 1975), pp. 341–343. ISSN: 0001-0782. DOI: 10.1145/360825.360861.

[22] Issam Sedki et al. 'An Effective Approach for Parsing Large Log Files'. In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2022, pp. 1–12. DOI: 10.1109/ICSME55016.2022.00009.

[23] Kirk Rodrigues, Yu Luo, and Ding Yuan. 'CLP: Efficient and Scalable Search on Compressed Text Logs'. In: *15th USENIX Symposium on Operating Systems Design and Implementation*. Ed. by Angela Demke Brown and Jay R Lorch. USENIX Association, 2021, pp. 183–198. URL: https://www.usenix.org/conference/osdi21/presentation/rodrigues.

[24] Wen Xia et al. 'The Design of Fast Content-Defined Chunking for Data Deduplication Based Storage Systems'. In: *IEEE Transactions on Parallel and Distributed Systems* 31.9 (Sept. 2020), pp. 2017–2031. ISSN: 1045-9219. DOI: 10.1109/TPDS.2020.2984632.

[25] Athicha Muthitacharoen, Benjie Chen, and David Mazières. 'A low-bandwidth network file system'. In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, Oct. 2001, pp. 174–187. ISBN: 1581133898. DOI: 10.1145/502034.502052.

[26] M O Rabin. *Fingerprinting by Random Polynomials*. Center for Research in Computing Technology: Center for Research in Computing Technology. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981. URL: https://books.google.de/books?id=Emu_tgAACAAJ.

[27] *The rsync algorithm*. URL: https://rsync.samba.org/tech_report/node2.html.

[28] Bo Feng, Chentao Wu, and Jie Li. 'MLC: An Efficient Multi-level Log Compression Method for Cloud Backup Systems'. In: *2016 IEEE Trustcom/BigDataSE/ISPA*. 2016, pp. 1358–1365. DOI: 10.1109/TrustCom.2016.0215.

[29] Wen Xia et al. 'FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication'. In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. USA: USENIX Association, 2016, pp. 101–114. ISBN: 9781931971300.

[30] Peng Zhou et al. 'UltraCDC:A Fast and Stable Content-Defined Chunking Algorithm for Deduplication-based Backup Storage Systems'. In: *2022 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, Nov. 2022, pp. 298–304. ISBN: 978-1-6654-8018-5. DOI: 10.1109/IPCCC55026.2022.9894295.

[31] Wen Xia et al. 'A Comprehensive Study of the Past, Present, and Future of Data Deduplication'. In: *Proceedings of the IEEE* 104.9 (Sept. 2016), pp. 1681–1710. ISSN: 0018-9219. DOI: 10.1109/JPROC.2016.2571298.

[32] Bryan Cantrill. *Visualizing Distributed Systems with Statemaps - Bryan Cantrill, Joyent - YouTube*. Dec. 2018. URL: https://www.youtube.com/watch?v=U4E0QxzswQc.

[33] Shkuro Yuri. *Mastering Distributed Tracing : Analyzing Performance in Microservices and Complex Systems.* Expert Insight. Packt Publishing, 2019. ISBN: 9781788628464.

[34] Kua and Patrick. *An Appropriate Use of Metrics*. 2013. URL: https://martinfowler.com/articles/useOfMetrics.html.

[35] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321150783.

[36] Paul Barham et al. 'Magpie: Online Modelling and Performance-aware Systems'. In: *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. Lihue, HI: USENIX Association, May 2003. URL: https://www.usenix.org/conference/hotos-ix/magpie-online-modelling-and-performance-aware-systems.

[37] *OpenTelemetry*. URL: https://opentelemetry.io/.

[38] *Jaeger: open source, end-to-end distributed tracing*. URL: https://www.jaegertracing.io/.

[39] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 'Anomaly detection'. In: *ACM Computing Surveys* 41.3 (July 2009), pp. 1–58. ISSN: 0360-0300. DOI: 10.1145/1541880.1541882.

[40] Ke Wang and Salvatore J. Stolfo. 'Anomalous Payload-Based Network Intrusion Detection'. In: *International Symposium on Recent Advances in Intrusion Detection*. 2004, pp. 203–222. URL: http://link.springer.com/10.1007/978-3-540-30143-1_11.

[41] Max Landauer et al. 'Deep Learning for Anomaly Detection in Log Data: A Survey'. In: (July 2022). DOI: 10.1016/j.mlwa.2023.100470.

[42] Maram Alamri and Mourad Ykhlef. 'Survey of Credit Card Anomaly and Fraud Detection Using Sampling Techniques'. In: *Electronics* 11.23 (Dec. 2022), p. 4003. ISSN: 2079-9292. DOI: 10.3390/electronics11234003.

[43] Junyu Wei et al. 'LogGrep: Fast and Cheap Cloud Log Storage by Exploiting both Static and Runtime Patterns'. In: *EuroSys '* 23 (). DOI: 10.1145/3552326.3567484. URL: https://doi.org/10.1145/3552326.3567484.

[44] Zhuangbin Chen et al. 'A Survey on Automated Log Analysis for Reliability Engineering'. In: *2021. A Survey on Automated Log Analysis for Reliability Engineering. ACM Comput. Surv* 1 (2021), p. 37. DOI: 10.1145/3460345. URL: https://doi.org/10.1145/3460345.

[45] R. Vaarandi. 'A data clustering algorithm for mining patterns from event logs'. In: *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003) (IEEE Cat. No.03EX764)*. IEEE, pp. 119–126. ISBN: 0-7803-8199-8. DOI: 10.1109/IPOM.2003.1251233.

[46] Keiichi Shima. *Length Matters: Clustering System Log Messages using Length of Words*. 2016.

[47] Hossein Hamooni et al. 'LogMine'. In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. New York, NY, USA: ACM, Oct. 2016, pp. 1573–1582. ISBN: 9781450340731. DOI: 10.1145/2983323.2983358.

[48] Liang Tang, Tao Li, and Chang-Shing Perng. 'LogSig'. In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. New York, NY, USA: ACM, Oct. 2011, pp. 785–794. ISBN: 9781450307178. DOI: 10.1145/2063576.2063690.

[49] Qiang Fu et al. 'Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis'. In: *2009 Ninth IEEE International Conference on Data Mining*. IEEE, Dec. 2009, pp. 149–158. ISBN: 978-1-4244-5242-2. DOI: 10.1109/ICDM.2009.60.

[50] Zhen Ming Jiang et al. 'Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper)'. In: *2008 The Eighth International Conference on Quality Software*. 2008, pp. 181–186. DOI: 10.1109/QSIC.2008.50.

[51] Masayoshi Mizutani. 'Incremental Mining of System Log Format'. In: *2013 IEEE International Conference on Services Computing*. IEEE, June 2013, pp. 595–602. ISBN: 978-0-7695-5026-8. DOI: 10.1109/SCC.2013.73.

[52] Adetokunbo A.O. Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. 'Clustering event logs using iterative partitioning'. In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM, June 2009, pp. 1255–1264. ISBN: 9781605584959. DOI: 10.1145/1557019.1557154.

[53] Meiyappan Nagappan and Mladen A Vouk. 'Abstracting log lines to log event types for mining software system logs'. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 2010, pp. 114–117. DOI: 10.1109/MSR.2010.5463281.

[54] Salma Messaoudi et al. 'A Search-Based Approach for Accurate Identification of Log Message Formats'. In: *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 2018, pp. 167–16710. ISBN: 978-1-4503-5714-2.

[55] Amey Agrawal et al. 'Delog: A High-Performance Privacy Preserving Log Filtering Framework'. In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 1739–1748. DOI: 10.1109/BigData47090.2019.9006218.

[56] Amey Agrawal, Rohit Karlupia, and Rajat Gupta. 'Logan: A Distributed Online Log Parser'. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, pp. 1946–1951. DOI: 10.1109/ICDE.2019.00211.

[57] Hetong Dai et al. 'Logram: Efficient Log Parsing Using nn-Gram Dictionaries'. In: *IEEE Transactions on Software Engineering* 48.3 (2022), pp. 879–892. DOI: 10.1109/TSE.2020.3007554.

[58] Tong Xiao et al. 'LPV: A Log Parser Based on Vectorization for Offline and Online Log Parsing'. In: *2020 IEEE International Conference on Data Mining (ICDM)*. 2020, pp. 1346–1351. DOI: 10.1109/ICDM50108.2020.00175.

[59] Shaohan Huang et al. 'Paddy: An Event Log Parsing Approach using Dynamic Dictionary'. In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 2020, pp. 1–8. DOI: 10.1109/NOMS47738.2020.9110435.

[60] Oihana Coustie et al. 'METING: A Robust Log Parser Based on Frequent n-Gram Mining'. In: *2020 IEEE International Conference on Web Services (ICWS)*. IEEE, Oct. 2020, pp. 84–88. ISBN: 978-1-7281-8786-0. DOI: 10.1109/ICWS49710.2020.00018. URL: https://ieeexplore.ieee.org/document/9283937/.

[61] Arthur Vervaet, Raja Chiky, and Mar Callau-Zori. 'USTEP: Unfixed Search Tree for Efficient Log Parsing'. In: *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE, Dec. 2021, pp. 659–668. ISBN: 978-1-6654-2398-4. DOI: 10.1109/ICDM51629.2021.00077. URL: https://ieeexplore.ieee.org/document/9679005/.

[62] Luyue Fang et al. 'QuickLogS: A Quick Log Parsing Algorithm based on Template Similarity'. In: *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, Oct. 2021, pp. 1085–1092. ISBN: 978-1-6654-1658-0. DOI: 10.1109/TrustCom53373.2021.00148. URL: https://ieeexplore.ieee.org/document/9724506/.

[63] Shijie Zhang and Gang Wu. 'Efficient Online Log Parsing with Log Punctuations Signature'. In: *Applied Sciences* 11.24 (Dec. 2021), p. 11974. ISSN: 2076-3417. DOI: 10.3390/app112411974. URL: https://www.mdpi.com/2076-3417/11/24/11974.

[64] Guojun Chu et al. 'Prefix-Graph: A Versatile Log Parsing Approach Merging Prefix Tree with Probabilistic Graph'. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2021, pp. 2411–2422. ISBN: 978-1-7281-9184-3. DOI: 10.1109/ICDE51399.2021.00274. URL: https://ieeexplore.ieee.org/document/9458609/.

[65] Armin Catovic et al. 'Linnaeus: A highly reusable and adaptable ML based log classification pipeline'. In: *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*. IEEE, May 2021, pp. 11–18. ISBN: 978-1-6654-4470-5. DOI: 10.1109/WAIN52551.2021.00008. URL: https://ieeexplore.ieee.org/document/9474393/.

[66] Leticia Decker, Daniel Leite, and Daniele Bonacorsi. 'Explainable Log Parsing and Online Interval Granular Classification from Streams of Words'. In: *2022 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. 2022, pp. 1–8. DOI: 10.1109/FUZZ-IEEE55066.2022.9882710.

[67] Xuheng Wang et al. 'SPINE: a scalable log parser with feedback guidance'. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, Nov. 2022, pp. 1198–1208. ISBN: 9781450394130. DOI: 10.1145/3540250.3549176. URL: https://dl.acm.org/doi/10.1145/3540250.3549176.

[68] Ying Fu et al. 'Investigating and improving log parsing in practice'. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, Nov. 2022, pp. 1566–1577. ISBN: 9781450394130. DOI: 10.1145/3540250.3558947. URL: https://dl.acm.org/doi/10.1145/3540250.3558947.

[69] Yu-Qian Zhu et al. 'ML-Parser: An Efficient and Accurate Online Log Parser'. In: *Journal of Computer Science and Technology* 37.6 (Dec. 2022), pp. 1412–1426. ISSN: 1000-9000. DOI: 10.1007/s11390-021-0730-4. URL: https://link.springer.com/10.1007/s11390-021-0730-4.

[70] Hao Lin et al. 'Cowic: A Column-Wise Independent Compression for Log Stream Analysis'. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015, pp. 21–30. DOI: 10.1109/CCGrid.2015.45.

[71] *AWS Data Transfer Charges for Server and Serverless Architectures | AWS Partner Network (APN) Blog*. URL: https://aws.amazon.com/blogs/apn/aws-data-transfer-charges-for-server-and-serverless-architectures/.

[72] *All networking pricing | Virtual Private Cloud | Google Cloud*. URL: https://cloud.google.com/vpc/network-pricing#internet_egress.

[73] Shilin He et al. 'Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics'. In: *arXiv preprint arXiv:2008.06448* (Aug. 2020).

[74] Shilin He et al. 'A Survey on Automated Log Analysis for Reliability Engineering'. In: *ACM Computing Surveys* 54.6 (July 2022), pp. 1–37. ISSN: 0360-0300. DOI: 10.1145/3460345. URL: https://dl.acm.org/doi/10.1145/3460345.

[75] *RFC 3164 - The BSD Syslog Protocol*. URL: https://datatracker.ietf.org/doc/html/rfc3164.

[76] *RFC 5424 - The Syslog Protocol*. URL: https://datatracker.ietf.org/doc/html/rfc5424.

[77] Tigran Najaryan. *oteps/0097-log-data-model.md at main · open-telemetry/oteps*. Apr. 2020. URL: https://github.com/open-telemetry/oteps/blob/main/text/logs/0097-log-data-model.md#prior-art.

[78] John Hopcroft. 'AN n log n ALGORITHM FOR MINIMIZING STATES IN A FINITE AUTOMATON'. In: *Theory of Machines and Computations*. Elsevier, 1971, pp. 189–196. DOI: 10.1016/B978-0-12-417750-5.50022-1. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780124177505500221.

[79] Janusz A. Brzozowski. 'Derivatives of Regular Expressions'. In: *Journal of the ACM* 11.4 (Oct. 1964), pp. 481–494. ISSN: 0004-5411. DOI: 10.1145/321239.321249.

[80] *The Go Programming Language*. URL: https://go.dev/.

[81] *araddon/dateparse: GoLang Parse many date strings without knowing format in advance*. URL: https://github.com/araddon/dateparse.

[82] *perlre - Perl regular expressions - Perldoc Browser*. May 2023. URL: https://perldoc.perl.org/perlre.

[83] Edward F Moore et al. 'GEDANKEN-EXPERIMENTS ON SEQUENTIAL MACHINES'. In: *Automata Studies. (AM-34)*. Princeton University Press, 1956, pp. 129–154. ISBN: 9780691079165. URL: http://www.jstor.org/stable/j.ctt1bgzb3s.8.

[84] Janusz A. Brzozowski. 'Canonical regular expressions and minimal state graphs for definite events'. In: *Mathematical theory of Automata. Volume 12 of MRI Symposia Series*. N.Y.: Polytechnic Press - Polytechnic Institute of Brookly, 1962, pp. 529–561.

[85] Alfred V Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[86] Jan Daciuk et al. 'Incremental Construction of Minimal Acyclic Finite-State Automata'. In: *Computational Linguistics* 26.1 (Mar. 2000), pp. 3–16. ISSN: 0891-2017. DOI: 10.1162/089120100561601.

[87] Jan Daciuk. 'Comparison of Construction Algorithms for Minimal, Acyclic, Deterministic, Finite-State Automata from Sets of Strings'. In: *Implementation and Application of Automata*. Ed. by Jean-Marc Champarnaud and Denis Maurel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 255–261. ISBN: 978-3-540-44977-5. URL: http://link.springer.com/10.1007/3-540-44977-9_26.

[88] Dean N. Arden. 'Delayed-logic and finite-state machines'. In: *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961)*. IEEE, 1961, pp. 133–151. DOI: 10.1109/FOCS.1961.13.

[89] R. McNaughton and H. Yamada. 'Regular Expressions and State Graphs for Automata'. In: *IEEE Transactions on Electronic Computers* EC-9.1 (Mar. 1960), pp. 39–47. ISSN: 0367-7508. DOI: 10.1109/TEC.1960.5221603.

[90] S. C. Kleene. 'Representation of Events in Nerve Nets and Finite Automata'. In: *Automata Studies. (AM-34)*. Princeton University Press, Dec. 1956, pp. 3–42. DOI: 10.1515/9781400882618-002.

[91] *perlrecharclass - Perl Regular Expression Character Classes - Perldoc Browser*. URL: https://perldoc.perl.org/perlrecharclass.

Abbreviations

API Application Programming Interface

ASCII American Standard Code for Information Interchange

AST Abstract Syntax Tree

AWS Amazon Web Services

CDC Content–Defined Chunking

CLP Compressed Log Processor

CPU Central processing unit

CSV Comma-Separated Values

DFA Deterministic Finite Automaton

GC Google Cloud

IQR Interquartile Range

JSON JavaScript Object Notation

LCS Longest Common Subsequence

NLP Natural Language Processing

NSGA–II Non–dominated Sorting Genetic Algorithm II

PID Process Identifier

PRI Priority Facility Severity

RAM Random Access Memory

RUM Real User Monitoring

SLR Systematic Literature Review

SVM Support Vector Machines

TID Thread Identifier

VTE Variable Template Extractor

VTEP Variable Template Extraction Parser

List of Figures

2.1 Two applications for prefix–trees (a) can match an input string on the fly; (b) can group similar strings based on length and common prefix . . 11
2.2 Exemplary dictionary compression with single and multiple levels . . . 14

3.1 The number of literature returned by the query on IEEE Xplore before applying the inclusion and exclusion criteria. 23
3.2 The selection process of the conducted SLR 26
3.3 The number of literature in the final set after applying all criteria 27
3.4 The engineering cycle, as proposed by Roel Wieringa in 'Design Science Methodology for Information Systems and Software Engineering' [12] . 36
3.5 An illustrative extraction of log templates which uses log entries from the loghub Spark dataset [73] 41
3.6 Log template extraction framework in offline mode 49
3.7 Log template extraction framework in online mode 50
3.8 CLPs variable dictionary approach applied to production logs 51
3.9 CLPs process of querying compressed data 53
3.10 The architecture to generate regular expressions for the configuration files of log compressors for identifying difficult dynamic tokens. 56

4.1 UML class diagram of VTE Configuration Struct 59
4.2 UML class diagram ofVTEP 61
4.3 UML activity diagram of the parsing process 62
4.4 UML activity diagram of the regex parse process 69
4.5 Real–world dynamic tokens from the Hadoop datasets are transformed to regular expressions using a prefix–tree and DFA minimization 75
4.6 An exemplary DFA to demonstrate Brzozowski Algebraic Method . . . 76
4.7 UML class–like diagram of the regex package with PrefixTree and Hopcroft minimization 76

5.1 Architecture of VTEP evaluation 82
5.2 Effectivness of log parsers based on execution time of the parsing process with increasing log volumne 83
5.3 Among 12 other parsers, VTE has a competitive mean. VTE+ achieving the highest mean. Sorted by mean. 84

5.4 Architecture of VTE evaluation 85
5.5 Architecture of parser results evaluation using Logzip as compressor. . 91
5.6 Logzip compression with templates from VTE+, Drain, and Revised compared to the original file size. Values in bytes. 91
5.7 Comparison of three compressors. Logzip and CLP are log–specific compressors. zstd is a general–purpose compressor. Values in bytes. . . 91

List of Tables

3.1 Log characteristics of 16 investigated datasets 45
3.2 OpenTelemetry's log data model as specified in [77] 49

4.1 Regular expressions and heuristic rules to identify token types 63

5.1 Accuracy of log parsers measured with 16 datasets and 14 parsers 81
5.2 Compression of 16 datasets with CLP. The columns Original, Default, Custom, and VTE show compressed file sizes in bytes. Custom to VTE and Default to VTE show the percentage difference, where positive values indicate a compression improvement and negative values a decrease. . . 87
5.3 Compression of 15 datasets with Logzip [17]. Three log parser results are passed to Logzip. The revised template files are found in the LogParser project [10]. L2 and L3 are Logzip compression modes, where L2 uses a template extraction file to match log records. L3 additionally creates a dynamic token encoding. The size is in bytes. The match column shows the percentage of log records that matched a log template. 88

Listings

1.1 Simple log record 1

2.1 Exemplary unstructured log records 7
2.2 Exemplary CSV log records 8
2.3 Exemplary JSON Syslog records 8
2.4 Log records flow for serving an image 19
2.5 Log records flow for serving an image with an error 19

3.1 IEEE Xplore database query 24
3.2 dblp database query 25
3.3 Unprocessed exemplary log messages 39
3.4 Aggregation based on the exemplary log messages 39
3.5 Apache example log messages 46
3.6 Thunderbird example log messages 46
3.7 Linux example log messages 46
3.8 Apache log message mapped to Opentelemetry data model in JSON format 48

4.1 Apache log format 65
4.2 Snippet of the VTE input file in JSON format 68
4.3 Snippet of extracted dynamic tokens from the Hadoop dataset 70

5.1 Data preparation for CLP configuration 85
5.2 Truncated snippet of the Thunderbird dataset showing alphabetical dynamic tokens 86
5.3 Snippet of Spark log records 90
5.4 Templates for a specific spark log record 90

.