Paddy: An Event Log Parsing Approach using Dynamic Dictionary

Shaohan Huang*, Yi Liu*, Carol Fung†, Rong He‡, Yining Zhao‡, Hailong Yang*, Zhongzhi Luan*
*Sino-German Joint Software Institute, Beihang University, Beijing, China
†Computer Science Department, Virginia Commonwealth University, Richmond, Virginia, USA
‡Chinese Academy of Scientific Computer Network Information Center, Beijing, China
huangshaohan@buaa.edu.cn, yi.liu@buaa.edu.cn, zhongzhi.luan@jsi.buaa.edu.cn

***Abstract*—Large enterprise systems often produce a large volume of event logs, and event log parsing is an important log management task. The goal of log parsing is to construct log templates from log messages and convert raw log messages into structured log messages. A log parser can help engineers monitor their systems and detect anomalous behaviors and errors. Most existing log parsing methods focus on offline methods, which require all log data to be available before parsing. In addition, the massive volume of log messages makes the process complex and time-consuming. In this paper, we propose Paddy, an online event log parsing method. Paddy uses a dynamic dictionary structure to build an inverted index, which can search the template candidates efficiently with a high rate of recall. The use of Jaccard similarity and length feature to rank candidates can improve parsing precision. We evaluated our proposed method on 16 real log datasets from various sources including distributed systems, supercomputers, operating systems, mobile systems, and standalone software. Our experimental results demonstrate that Paddy achieves the highest accuracy on eight data sets out of sixteen datasets compared to other baseline methods. We also evaluated the robustness and runtime efficiency of the methods and the experimental results show that our method Paddy achieves superior stableness and is scalable with a large volume of log messages.**

***Index Terms*—Log Parsing, Dynamic Dictionary, Log analysis**

I. Introduction

Logs are widely used to record either events that occur in an operating system or other software systems, or messages between different users of communication software. Despite logs contain rich information, how to analyze them effectively is still a great challenge [1]. First, with the increasing size and complexity of modern systems, the volume of logs is rapidly growing (e.g., about gigabytes of data per hour for a commercial cloud application [2]). The large volume of logs makes it impractical to manually extract the key diagnostic information from logs. Second, raw log message are unstructured. This further increases the difficulty in automated analysis of log data.

To enable log management on the huge volume of logs, the first and foremost step is log parsing, a process to convert unstructured raw log messages into structured events. Figure 1 shows that an unstructured log message printed by a logging code from MemoryStore class in Spark. Unstructured log messages usually contain various forms of system runtime information including timestamp , verbosity level (the severity level of an event, e.g., INFO), and raw message content. The goal of log parsers is to transfer raw messages content into structured log templates associated with key parameters (e.g., log template "Block * stored as bytes in memory (estimated size * B, free * KB)" with parameters ['broadcast_0_piece0', '93.0', '317.1']). Here, '*' is a wildcard to match parameters.

```
/*
A logging code snippet from spark/storage/memory/MemoryStore.scala
*/
logInfo("Block %s stored as bytes in memory (estimated size %s,
free %s)".format(
  blockId,
  Utils.bytesToString(size),
  Utils.bytesToString(maxMemory - blocksMemoryUsed)))
```

Log Message

```
17/06/09 20:10:48 INFO storage.MemoryStore: Block broadcast_0_piece0 stored
as bytes in memory (estimated size 93.0 B, free 317.1 KB)
```

Structured Log

TIMESTAMP	17/06/09 20:10:48
LEVEL	INFO
CLASS	storage.MemoryStore
TEMPLATE	Block * stored as bytes in memory (estimated size * B, free * KB)
PARAMETERS	['broadcast_0_piece0', '93.0', '317.1']

Fig. 1: An illustrative example of log parsing.

Traditionally, log parsing relies heavily on regular expressions [2, 3] designed by developers. Developers must check all logging statements on a regular basis, which is difficult when the scale of log messages is large. In recent years, automatic raw log parsing has been widely studied [4, 5]. Most of the existing methods focus on offline approach, which requires all log data to be available before parsing. In contrast, an online log parser parses logs in real-time, and it does not require an offline training phase. Online log parsing methods have been studied in the literature [6]–[8]. However, we observe that the parsers proposed in these works are not accurate and efficient enough (e.g., the average accuracy of Spell [7] is less than 80%) or some methods are based on a very strong assumption (e.g., assume that log messages with the same log event will have the same log message length [6]).

978-1-7281-4973-8/20/$31.00

In this paper, we propose Paddy, an online event log parsing approach. Paddy first retrieves some log template candidates from exiting templates and then ranks these candidates to find proper templates. How to rank template candidates is the key to improve the parsing precision. The ranking method can be considered from two perspectives, as the text similarity and difference in length. Retrieving template candidates affects the efficiency of Paddy. We use a dynamic dictionary structure to implement an inverted index. which helps search template candidates effectively with a high rate of recall and also avoids very strong assumptions.

We evaluated our proposed method on a total of 16 log datasets including distributed systems, supercomputers, operating systems, mobile systems, and standalone software. Our experimental results show that Paddy achieves the best accuracy on eight data sets out of sixteen datasets. Especially, the proposed method achieves the state-of-the-art on average accuracy. We also evaluated the robustness and efficiency on more than 1 GB raw log messages. The experimental results show that our method achieves comparable performance with the large volume of logs.

The key contributions of this paper are summarized as follows:

- We propose an online log parsing method (Paddy), which leverages the dynamic dictionary to improve the performance of log parsing.
- We combine the Jaccard similarity and length feature to improve parsing precision.
- We evaluated the accuracy and efficiency of our method on 16 real-world log data sets.

The rest of the paper is organized as follows. The architecture design and our methodology are described in Section II. Section III describes the experimental settings and we evaluate the performance of Paddy. Related work is introduced in Section IV. Finally in Section V, we conclude our work and state possible future work.

II. Methodology

In this section, we first present the design of Paddy, an event log **P**arsing **A**pproch with **D**ynamic **D**ictionar**Y**. Then we describe our log parser in detail and explain how Paddy parses raw log messages and builds its dynamic dictionary.

A. Overview of Paddy

The goal of log parsing is to extract the template from log messages and transform raw log messages into structured log messages. Figure 2 shows structure and workflow of our system. As it shown, our log parsing system consists of four steps: 1) preprocess raw log; 2) retrieve candidates from the inverted index; 3) rank template candidates; 4) update its template list and dynamic dictionary.

Especially, our log parser consists of two parts, namely, inverted index and template list. The inverted index maps tokens in a template to its corresponding template. The key of the inverted index is the constant tokens (e.g., 'Received', 'block', 'of', 'size') and its value is ID of template list. The

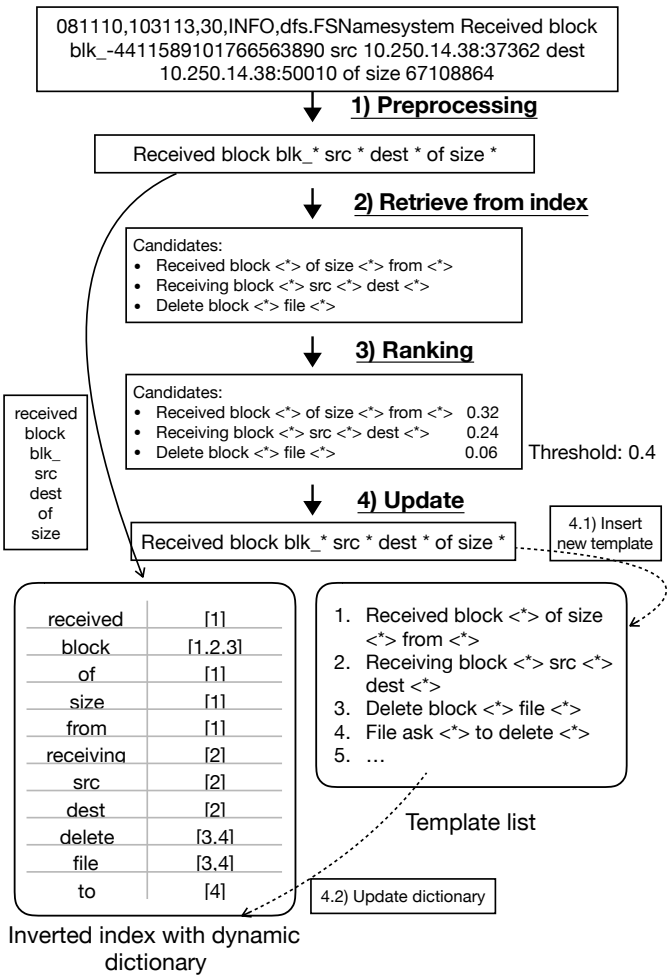


Fig. 2: Basic workflow of Paddy.

inverted index is implemented using a dictionary structure. In Paddy, we need to insert or delete elements from its inverted index dynamically. We call its structure a dynamic dictionary. We can use the template ID to find its corresponding template in template list. We adopt dynamic adjustment when we insert a new template or modify an existing template. We will introduce the details in the following section.

B. Step 1: Preprocess by Domain Knowledge

Instead of running log parsing methods directly on raw log messages, developers usually preprocess log data with domain knowledge. According to the previous empirical study [9], simple log preprocessing using domain knowledge (e.g. removal of IP address) can further improve log parsing accuracy.

Correspondingly, the first step of Paddy is to process the raw log messages. To do so, we first implement some simple rules to extract timestamp, log level, and class name, which usually have fixed length or tailed by special characters. Then our system allows users to use simple regular expressions to process raw message contents. The regular expressions based on domain knowledge can represent commonly-used variables,

such as IP address and block ID. For example, the block IDs in Figure 2 will be replaced by 'blk_[0-9]+' and the IP address is represented by '([0-9]+.)3[0-9]+(:[0-9]+)'. For the comparison purpose, we use the same preprocessing rules to each log parser.

C. Step 2: Retrieve from Inverted Index

After preprocessing, we explain how our method finds template candidates from the inverted index for preprocessed log messages.

As described in Section II-A, the inverted index maps tokens in a template to its corresponding template, which is the most popular data structure used in document retrieval systems [10]. We adopt an inverted index structure for log parsing in order to find the template candidates quickly. For example, in Figure 2, 'Received block blk_* src * dest * of size *' is a preprocessed log message into a template. We first split the log message into tokens that exclude wildcard characters. In this case, given the token list ['Received', 'block', 'src', 'dest', 'of', 'size'], we can retrieve template candidates from the inverted index. Finally, we obtain the template IDs [1, 2, 3]. Paddy also allows users to set a banned word list to filter some words in inverted index.

Specifically, we use a certain strategy to speed up and improve log parsing speed. We rank the template candidates in a simple strategy and keep up to one hundred templates as candidates. We use the number of overlap tokens to score each template candidate. We can compute the score directly through the inverted index structure without any further effort. If we do not find any template from the inverted index, we will use the log message as a new template to insert into template list. We will introduce how to insert a new template in Section II-E.

D. Step 3: Rank Template Candidates

After retrieving template candidates using the inverted index, Paddy ranks these candidates by computing their fitting scores. We compute fitting scores from two aspects, namely, *Similarity* and *Lengthfeature*. The *Similarity* feature is a metric that measures similarity between log messages and the log templates. As introduced in the work of He et. al [6], the same log event will probably have the same log message length. Because the length of log message plays an important role in log parsing, we design the *Lengthfeature* to help rank template candidates. Therefore, we rate the template candidates based on their length feature and the similarity to log message as follows:

$$Fittingscore = \lambda_1 \cdot Similarity + \lambda_2 \cdot LengthFeature \quad (1)$$

where λ_1 and λ_2 are coefficients that control the weight of $Similarity$ and $LengthFeature$. In the evaluation section, we set $\lambda_1 = \lambda_2 = 0.5$. λ_1 and λ_2 are two hyper-parameters to control the weight of different features. A larger λ means more weight on the corresponding feature. We choose 0.5 for both λ_1 and λ_2 in order to use the same weight each feature. Paddy also allows users to set different weights to adapt their own dataset.

We use Jaccard similarity [11] to compute the similarity between the log message and the log template. Jaccard similarity is used for gauging the similarity and diversity of sample sets. The $Similarity$ is defined as follows:

$$Similarity = \frac{|A \cap B|}{|A \cup B|} \quad (2)$$

where A is the token set of the log message and B is the token set of log template. The $Similarity$ ranges from 0 to 1. The $Similarity$ is 1 when $A = B$. When building the token set, we exclude the wildcard characters. For example, for log message 'Received block blk_* src * dest * of size *' and its token set is ['Received', 'block', 'src', 'dest', 'of', 'size']; for log template 'Received block <*> of size <*> from <*>' and its corresponding token set is ['Received', 'block', 'of', 'size', 'from']. The $Similarity$ of the above pair is $4/7$.

We design $Lengthfeature$ to measure the difference between the length of log message and log template. The two desired properties of lengthfeature are: first, the $Lengthfeature$ should range from 0 to 1; second, the $Lengthfeature$ should be equal to 1 when a log message has the same template length. We design the $Lengthfeature$ as follows:

$$Lengthfeature = 1 - \frac{|M - T|}{max(M, T)} \quad (3)$$

where M is the length of log message and T is the length of log template. We define the length is the number of tokens. When the log message M has the same length of log template T, $Lengthfeature$ reaches the maximum value of 1.

After identifying the log template with the highest $Fittingscore$, we compare it with a predefined threshold t. If $Fittingscore \geq t$, Paddy returns the log template to the next step. Otherwise, Paddy returns that no suitable template is found.

E. Step 4: Update Template List and Dynamic Dictionary

In this section, we introduce how to insert a new log template or update an exiting log template in an inverted index and template list.

As described in the previous section, Paddy creates a new log template if the system does not retrieve any template from the inverted index or the template with the highest $Fittingscore$ does not pass the threshold t. For example, in Figure 2, the log message is 'Received block blk_* src * dest * of size *' and the highest $Fittingscore$ of the existing log templates is lower than 0.4 (assume threshold is 0.4). We then insert the log message as a new log template into the template list. Assuming the new template's ID is 5, then we use this ID to update the dynamic dictionary. We let the log message tokens that do not contain wildcard characters to be the dictionary keys. If a key is in the dictionary, the template ID will be appended to the value of the key (e.g., dict['block'].append(5)). Otherwise, add the key value pair (e.g., dict['block']=5) into the dictionary.

TABLE I: Summary of log parsing data sets.

Dataset	Description	Data Size	#Messages	#Templates (2K)	#Max Length	#Average Length
Distributed system logs						
HDFS	Hadoop distributed file system log	1.47 GB	11,175,629	14	111	12.45
Hadoop	Hadoop mapreduce job log	48.61 MB	394,308	114	50	14.82
Spark	Spark job log	2.75 GB	33,236,604	36	22	12.76
ZooKeeper	ZooKeeper service log	9.95 MB	74,380	50	26	13.46
OpenStack	OpenStack software log	60.01 MB	207,820	43	31	20.63
Supercomputer logs						
BGL	Blue Gene/L supercomputer log	708.76 MB	4,747,963	120	84	15.32
HPC	High performance cluster log	32.00 MB	433,489	46	47	9.56
Thunderbird	Thunderbird supercomputer log	29.60 GB	211,212,192	149	132	17.52
Operating system logs						
Windows	Windows event log	26.09 GB	114,608,388	50	42	31.93
Linux	Linux system log	2.25 MB	25,567	118	24	14.39
Mac	Mac OS log	16.09 MB	117,283	341	249	15.49
Mobile system logs						
Android	Android framework log	3.38 GB	30,348,042	166	32	13.31
HealthApp	Health app log	22.44 MB	253,395	75	14	2.93
Server application logs						
Apache	Apache server error log	4.90 MB	56,481	6	14	12.28
OpenSSH	OpenSSH server log	70.02 MB	655,146	27	19	13.81
Standalone software logs						
Proxifier	Proxifier software log	2.42 MB	21,329	8	27	13.73

If a selected log template is returned in step 3, Paddy will update the returned log template based on the log message. We take the log template as a regular expression where a wildcard character * can match one or more tokens. If the exiting log template matches the coming log message, we do not modify the template list and the dynamic dictionary. Otherwise, we will modify the returned template by replacing the word token with a wildcard. We scan the tokens in the log message and the log template from left to right. If two tokens are the same or a token is a wildcard, we do not modify the token. Otherwise, we update the token by a wildcard in the log template. After modifying the returned template, Paddy will update the inverted index by removing some items from the dynamic dictionary. For example, the return log template is 'Receive from node 4' and the log message is 'Receive from node 20'. Paddy will update the template into 'Receive from node *' and remove this template from dictionary where the key is '4'.

III. Experimental Evaluation

A. Experimental Settings

We evaluate our Paddy method using LogHub [1] datasets, which is a large collection of logs from 16 different systems, such as distributed systems, supercomputers, operating systems, mobile systems, server applications, and standalone software [12]. Some of the datasets in LogHub were released by previous studies such as HDFS [3], Hadoop [13], and BGL [14]. The others (e.g., Spark, Zookeeper, HealthApp, Android) were collected by the lab environment [12].

As illustrated in Table I, LogHub contains a total of 440 million log messages which is 77 GB in size. Thunderbird dataset is the largest log dataset which contains around 211 million log messages. The smallest log dataset Proxifier contains 21 thousand messages. The large size and diversity of LogHub can be used to measure the robustness and efficiency of our proposed log parser.

To evaluate the accuracy of our log parser, we sampled $2,000$ log messages from each dataset randomly and carefully labeled their event templates manually as ground truth. In particular, the header *#Templates (2K)* in Table I indicates the number of event templates in log samples. We also list the maximum length and average length of log samples on different log datasets in Table I.

B. Baseline

We evaluate the efficiency and effectiveness of Paddy by comparing it with six popular offline log parsing algorithms. Two of them are offline log parsers and the other four are online ones. These log parsers are briefly described as follows:

- LogCluster [4]: LogCluster uses frequent pattern mining to automated log parsing. It is an offline method.
- LogMine [5]: LogMine is an offline log parsing method based on clustering algorithms. It generates event templates using a hierarchical clustering method.
- LenMa [15]: LenMa is an online clustering method. It parses logs in a streaming-like manner. New log messages are added to an existing cluster if a match is found. Otherwise, a new log cluster will be created.
- Spell [7]: Spell employs the longest common subsequence algorithm to parse logs in a streaming manner.
- Drain [6]: Drain applies a fixed-depth tree structure to parse log messages and extracts common templates.
- MoLFI [8]: MoLFI is a search-based method, where the log parsing problem is modeled as a multiple-objective optimization problem and solves it using evolutionary algorithms.

[1] https://github.com/logpai/loghub

TABLE II: Accuracy results on LogHub dataset.

Dataset	LogCluster	LogMine	LenMa	Spell	Drain	MOLFI	Paddy	Best	Average
HDFS	0.546	0.851	**0.998**	**1***	**0.998**	**0.998**	**0.940**	1	0.904
Hadoop	0.563	0.870	0.885	0.778	**0.948**	**0.957***	**0.952**	0.957	0.850
Spark	0.799	0.576	0.884	**0.905**	**0.920**	0.418	**0.980***	0.980	0.783
Zookeeper	0.732	0.688	0.841	**0.964**	**0.967**	0.839	**0.986***	0.986	0.860
OpenStack	0.696	0.743	0.743	0.764	0.733	0.213	0.839*	0.839	0.676
BGL	0.835	0.723	0.690	0.787	**0.963***	**0.960**	**0.963***	0.963	0.846
HPC	0.788	0.784	0.830	0.654	0.887	0.824	**0.904***	0.904	0.810
Thunderb.	0.599	**0.919**	**0.943**	0.844	**0.955***	0.646	**0.951**	0.955	0.837
Windows	0.713	**0.993**	0.566	**0.989**	**0.997***	0.406	**0.994**	0.997	0.808
Linux	0.629	0.612	0.701	0.605	0.690	0.284	0.859*	0.859	0.626
Mac	0.604	0.872*	0.698	0.757	0.787	0.636	0.851	0.872	0.744
Android	0.798	0.504	0.880	**0.919***	**0.911**	0.788	0.878	0.919	0.811
HealthApp	0.531	0.684	0.174	0.639	0.780*	0.44	0.755	0.780	0.572
Apache	0.709	**1***	**1***	**1***	**1***	**1***	**1***	1	0.958
OpenSSH	0.426	0.431	**0.925**	0.554	0.788	0.500	**0.938***	0.938	0.652
Proxifier	**0.951***	0.517	0.508	0.527	0.527	0.013	0.527	0.951	0.510
Average	0.665	0.694	0.721	0.751	0.865	0.605	0.895	*	*

For the comparison purpose, we use the same pre-processing rules for each log parser. Similar to the work in [12], we compute the accuracy of all log parsers based on over 10 experimental runs and report the best results to avoid bias from randomization. All the experiments were conducted on a server equip with 32 Intel(R) Xeon(R) 2.60GHz CPUs and Ubuntu 16.04.3 LTS.

C. Evaluation Metrics

We use three metrics, namely, *parsing accuracy*, *robustness* and *runtime efficiency* to evaluate the performance of the log parsers. Similar to the work of Zhu et. al. [12], we define the parsing accuracy as follows:

$$Accuracy = \frac{CN}{TN} \tag{4}$$

where CN is the correct number of parsed log messages and TN is the total number of log messages. After parsing, each log message has an event template. Each event template corresponds to a group of log messages. A log message is considered correctly parsed if and only if its event template corresponds to the same group of log messages as the ground truth does. For example, if a log sequence [E1, E2, E2, E3, E3] is parsed to [E1, E4, E5, E6, E6], we get $Accuracy = (1 + 2)/5 = 0.6$ because the 2nd and 3rd messages are not grouped while the 4th and 5th log messages are grouped. We use the metric parsing accuracy to quantify the effectiveness of log parsing methods.

D. Experimental Results

In this section, we evaluate the log parsers from three aspects: accuracy, robustness, and efficiency.

1) Accuracy of Log Parsers: In each LogHub dataset, $2,000$ log messages are manually labeled with corresponding event templates. We use these event templates as ground truth to evaluate the accuracy of the log parsers. The subset is randomly sampled from the original log dataset so that retains the key properties of the original data set such as event redundancy and event variety.

As illustrated in Table II, we compare the accuracy of Paddy with six other baseline log parsing methods on 16 log datasets. Each row lists the parsing accuracies of various log parsers on one dataset, which helps compare the parsing accuracy of different log parsers on one data set. Each column represents the parsing accuracies of one log parser over different datasets, which shows the robustness of a parser across different types of logs. The last two columns are the best accuracy and the average accuracy of each dataset. We highlight the high accuracy values greater than 0.9 in bold. We use an asterisk sign * to mark the best accuracy of each dataset. We also compute the average accuracy of each log parser over different datasets in the last row.

From Table II, we can observe that Paddy significantly outperforms other baseline methods on an average accuracy of 0.89, followed by Drain of 0.86. In addition, Paddy achieves a high accuracy (marked in bold) on 10 data sets out of 16, which is the highest among all parsers. The second best log parser Drain also attains a high accuracy on 9 data sets out of 16. In contrast, LogCluster only achieves high accuracy on one dataset. For data set Thunderb., Android, and HealthApp, Paddy also has the second-best accuracy.

Furthermore, the average accuracy of the HDFS and Apache datasets is more than 0.9 and some parsers can achieve 100% accuracy. This is because HDFS and Apache error logs have relatively simple event templates and are easy to identify. However, some log datasets such as HealthApp and Proxifier are not parsed accurately. This is because their log messages are short and lack adequate information (e.g., HealthApp). Paddy has the best accuracy overall for three reasons. First, it uses the dynamic dictionary to build an inverted index to ease the search of template candidates. Second, Paddy combines the Jaccard similarity and log message length to rank templates, which improves the precision of parsing. Third, Paddy allows users to set different thresholds on fitting scores for different data sets.

2) Robustness of Log Parsers: In this part, we evaluate the robustness of log parsers from three aspects: 1) robustness on different types of logs, 2) robustness from the perspective of

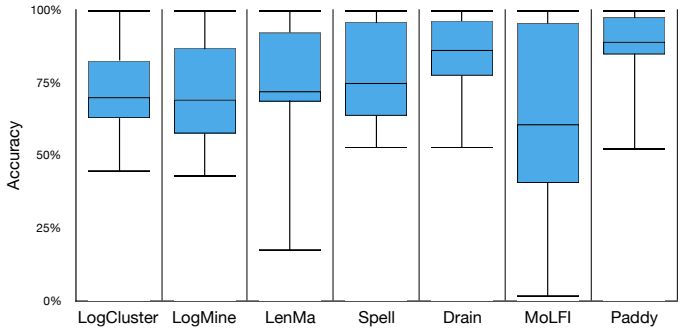


Fig. 3: Accuracy distribution of log parsers on different types of logs.

TABLE III: Standard deviation of log parsers.

LogCluster	LogMine	LenMa	Spell	Drain	MoLFI	Paddy
0.135	0.176	0.214	0.162	0.136	0.307	0.119

the length of the logs, and 3) robustness from the perspective of the number of log templates.

Figure 3 shows the accuracy of each log parser across the 16 log datasets in box and whisker plot. The horizontal lines from bottom to top of each box correspond to the minimum, 25-percentile, median, 75-percentile, and maximum accuracy values. We can see that the maximum accuracy values of most log parsers are near 100%. However, some of them have a large variance over different datasets, such as LenMa and MoLFI. From the perspective of 25-percentile and 75-percentile (blue box), Paddy performs the best among all the log parsers in the graph. It does not only have the highest accuracy on average but also has the smallest variance, which is the sign of robustness. We also list the standard deviation of each log parser in Table III. The standard deviation also indicates that our method Paddy has the smallest standard diviation which shows its robustness on different types of logs.

Meanwhile, we evaluate the robustness of log parsers from the perspective of the length of log messages. We select the top four log parsers to explore the relation between accuracy and the length of log messages. As mentioned in the previous section, the accuracy of log parsers is impacted by the length of log messages. We plot a line chart (Figure 4) to better understand such impact, where the x-axis represents the average length of log messages and the y-axis indicates the accuracy of log parsers. As shown in Figure 4, the accuracy has small fluctuations when the length is less than 13. We highlight this in the left dashed rectangle. Similarly we highlight the the portion with length over 14 in another dashed rectangle. As we can see the dashed rectangle on the right side, the accuracy is greatly impacted by the length of log messages. We can also see that all methods cannot parse log accurately when a log message is short. Therefore, further improvements should be made towards parsing extremely short log data.

In our next experiment, we evaluate the robustness of log parsers from the perspective of the number of log templates.

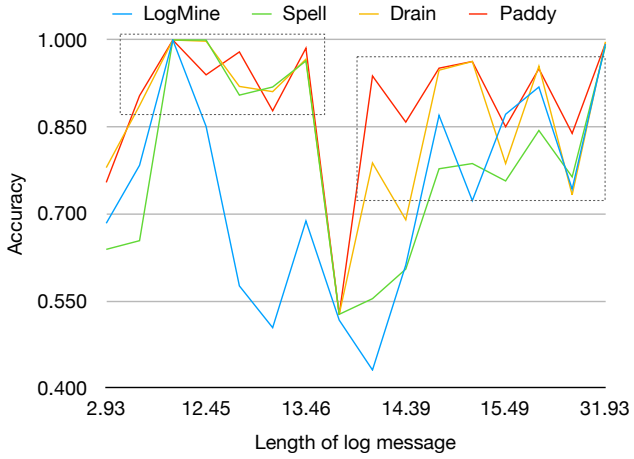


Fig. 4: Accuracy distribution of log parsers from the perspective of length of logs.

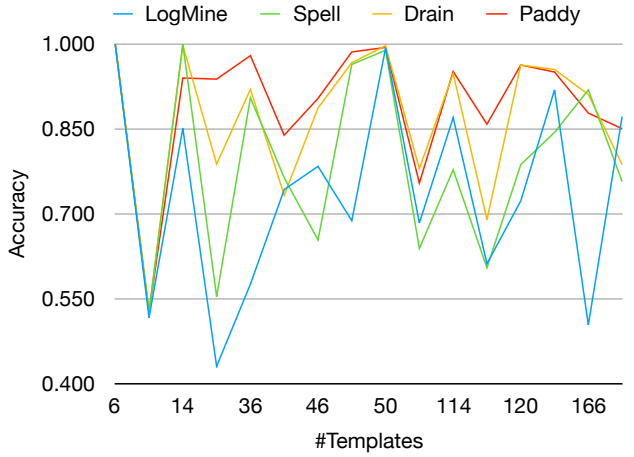
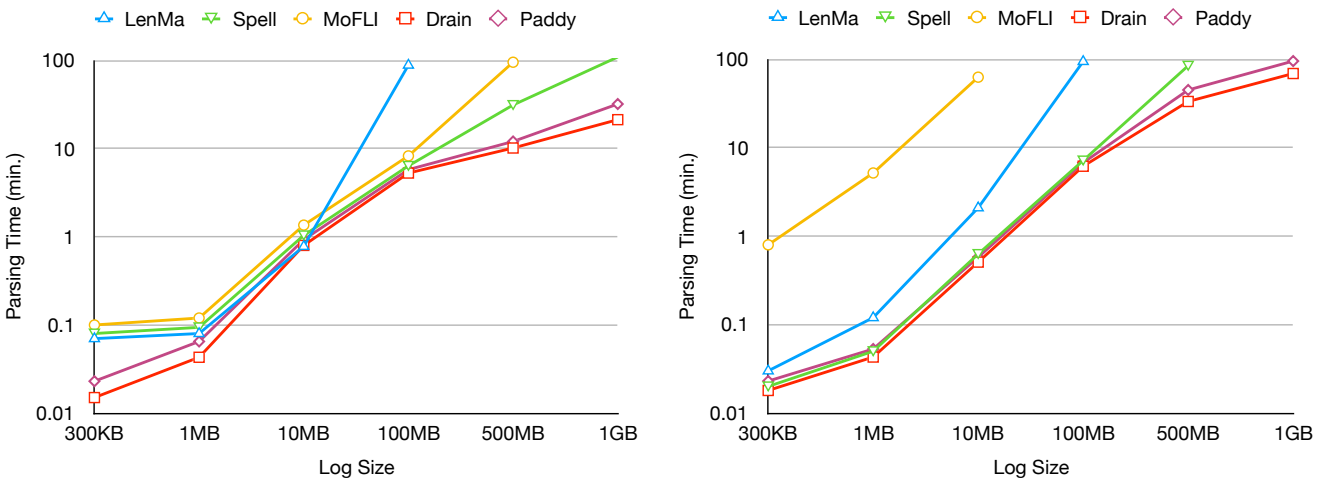


Fig. 5: Accuracy distribution of log parsers from the perspective of the number of log templates.

To some extent, more log templates means more complex structure. Excessive number of event templates increases log parsing complexity. Figure 5 shows the relation between accuracy (y-axis) and the number of log templates (x-axis). We can see that there is not an obvious linear relation between the accuracy and the number of templates. We conclude that the number of log templates is not the major influence factor of the performance. However, Figure 5 shows that our method Paddy achieves relatively stable accuracy compared to the other baseline methods.

3) Efficiency of Log Parsers: In this part, we evaluate the efficiency of Paddy. We select four log parsers, i.e., MoLFI, Spell, LenMa, and Drain as the baseline methods. They demonstrate high accuracy (over 90%) on more than four log datasets, as shown in Table II. We choose two large log datasets, i.e., BGL and Android from LogHub. The raw logs have a volume of over 1GB each and they belong to different types. BGL and Android have also been used as benchmarks datasets in the previous work [7, 12, 16].



(a) BGL (b) Android

Fig. 6: Running time of log parsing methods on data sets in different size.

For each log dataset, we vary the volume from 300 KB to 1 GB, while fixing the parameters of log parsers that were fine-tuned on 2k log samples. Specifically, 300KB is roughly the size of each 2k log sample. We truncate the raw log files to obtain samples of other volumes (e.g., 1GB).

The running time results are shown in Figure 6. We can observe that the parsing time increases with the raising of log size on both datasets. Drain method has a better efficiency, which scales linearly with the log size. Paddy results are comparable to Drain method on BGL and Android datasets. Both methods can finish parsing 1GB of logs within tens of minutes. Especially, LenMa and MoLFI cannot even finish parsing 1GB of BGL data or Android data within 100 minutes. The efficiency of a log parser also depends on the type of logs. We observe that the log parsing of Android dataset needs more time than the BGL dataset. It is because Android dataset contains more event templates. For example, Paddy method retrieves more templates from its inverted index, which increases log parsing time. Specifically, we keep at most one hundred templates as candidates. This strategy can help improve its efficiency with a large number of event templates.

The number of templates is an important metric to evaluate the stability of our model. We conduct our experiments on BGL and Android datasets. The number of templates results are shown in Figure 7. The x-axis is the number of templates and the y-axis is the number of parsed log. We can observe that Paddy arrives at a stable number of templates and the number of template will not explode.

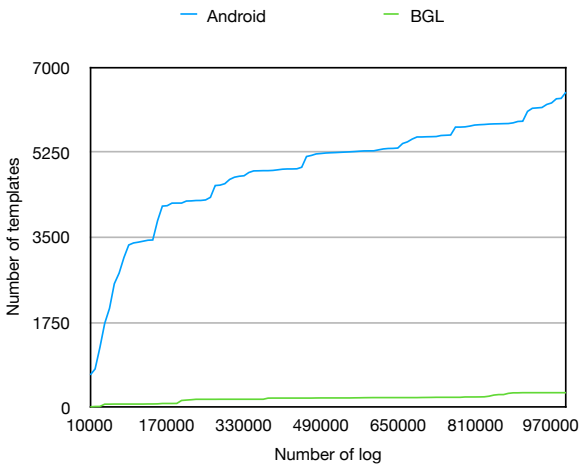


Fig. 7: The number of templates in different size.

IV. Related Work

Event log parsing plays an important role in system maintenance, which has been widely studied in recent years including rule-based, source code-based, and data-driven parsing [12]. In this work, we focus on data-driven log parsing approaches, which learn event log parsing without heavily relying on application-specific knowledge. Data-driven log parsing techniques can be categorized from three aspects.

1) Frequent Pattern Mining: Frequent pattern mining is to discover frequently occurring line patterns from event logs, which assume that each event is described by a single line in the event log, and each line pattern represents a group of similar events. To our best knowledge, SLCT [17] is the earliest frequent pattern mining algorithm for event log parsing, which identifies frequent words with a pass over the input data set first and then groups log messages into several clusters. LFA [18] leverages the token frequency distribution in each log message to improve its robustness. LogCluster [4] is an improved version of SLCT and is not sensitive to shifts in word positions. All the three log parsers are offline methods and they need to traverse over the whole input data first.

2) Clustering: Many previous work treat event log parsing as a log clustering problem and propose many clustering approaches to tackle this task. From this view, event log lines that share the same templates can be grouped into one cluster. LKE [19], LogSig [20], and LogMine [5] are offline clustering methods. LKE uses a k-means clustering algorithm based on

edit distance to extract log keys from free text messages. LogSig clusters similar frequency words in each line and abstracts it to event types. LogMine employs a hierarchical clustering way to cluster the messages into coherent groups, and identifies the most specific log pattern to represent each cluster.

SHISO [21] and LenMa [15] are both online clustering methods. For each newly coming log message, SHISO first computes its similarity to event templates of existing log clusters. The log message will be added to an existing cluster if it is successfully matched. Otherwise, a new log cluster will be created. SHISO creates a structured tree using the nodes generated from log messages to improve its performance.

3) Heuristics: Some works propose heuristics-based event log parsing methods, which leverage the unique characteristics of log messages. AEL [22] detects the dynamic and static parts in each log line and clusters the event log lines with the same static parts and the same structure of dynamic parts into the same execution events. IPLoM [16] proposes a 3-Step hierarchical iterative partitioning process IPLoM partitions log data into its respective clusters. Spell [7], which is an online streaming method Spell, utilizes the longest common sub-sequence based approach, to parse system event logs. Drain [6] designs a fixed depth tree structure to represent log messages and extracts common templates in leaf nodes. MoLFI [8] recasts the log message parsing problem as a multi-objective problem and uses an evolutionary approach to solve this problem. Different from treating event log as general text data, these heuristics make use of the characteristics of logs and perform quite well in many cases.

V. Conclusion

In this paper, we proposed Paddy, an online event log parsing approach with a dynamic dictionary. The dynamic dictionary helps search the template candidates effectively with a high rate of recall. Our method combines the Jaccard similarity and length feature to rank candidates in order to improve parsing precision. We conducted our experiments on 16 real-world log datasets. Our experimental results show that Paddy obtains the highest accuracy on eight data sets out of 16 datasets. Especially, the proposed method achieves the state-of-the-art on average accuracy. We also evaluate the robustness and efficiency of more than 1 GB raw log messages. One of the future directions of our work is to leverage log parsing into anomaly detection tasks.

Acknowledgment

This research is supported by National Key R&D Program, under the grant No. 2018YFB0204002; National Science Foundation of China, under the grant No. 61732002.

References

[1] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Communications of the ACM*, vol. 55, no. 2, pp. 55–61, 2012.

[2] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1245–1255, 2013.

[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 117–132.

[4] R. Vaarandi and M. Pihelgas, "Logcluster-a data clustering and pattern mining algorithm for event logs," in *2015 11th International Conference on Network and Service Management (CNSM)*. IEEE, 2015, pp. 1–7.

[5] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1573–1582.

[6] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 33–40.

[7] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 2016, pp. 859–864.

[8] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 167–177.

[9] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 654–661.

[10] J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted files versus signature files for text indexing," *ACM Transactions on Database Systems (TODS)*, vol. 23, no. 4, pp. 453–490, 1998.

[11] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," in *Proceedings of the international multiconference of engineers and computer scientists*, vol. 1, no. 6, 2013, pp. 380–384.

[12] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 2019, pp. 121–130.

[13] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 102–111.

[14] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 2007, pp. 575–584.

[15] K. Shima, "Length matters: Clustering system log messages using length of words," *arXiv preprint arXiv:1611.03213*, 2016.

[16] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 1255–1264.

[17] 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, 2003, pp. 119–126.

[18] M. Nagappan and M. 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)*. IEEE, 2010, pp. 114–117.

[19] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *2009 ninth IEEE international conference on data mining*. IEEE, 2009, pp. 149–158.

[20] L. Tang, T. Li, and C.-S. Perng, "Logsig: Generating system events from raw textual logs," in *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM, 2011, pp. 785–794.

[21] M. Mizutani, "Incremental mining of system log format," in *2013 IEEE International Conference on Services Computing*. IEEE, 2013, pp. 595–602.

[22] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting execution logs to execution events for enterprise applications (short paper)," in *2008 The Eighth International Conference on Quality Software*. IEEE, 2008, pp. 181–186.