HCLPars: A New Hierarchical Clustering Log Parsing Method

Arwa Bin Lashram
University of Jeddah, Saudi Arabia
arwa.alashram@yahoo.com (corresponding author)

Lobna Hsairi
University of Jeddah, Saudi Arabia
lalhabib@uj.edu.sa

Haneen Al Ahmadi
University of Jeddah, Saudi Arabia
hhalahamade@uj.edu.sa

Received: 5 May 2023 | Revised: 17 May 2023 | Accepted: 18 May 2023

 DOI: https://doi.org/10.48084/etasr.6013

ABSTRACT

Event logs are essential in many software systems' maintenance and development, as detailed runtime information is recorded in them, allowing support engineers and developers to monitor systems, understand behaviors, and identify errors. With the increasing size and complexity of modern software systems, parsing their logs by the traditional (manual) method is cumbersome and useless. For this reason, recent studies have focused on automatically parsing log files. This paper presents the Hierarchical Clustering Log Parsing method, called HCLPars, for automatically parsing log files, consisting of 3 steps: parameter removal according to acquired knowledge in order to avoid errors, grouping similar raw log messages, and getting the set of keys that make up the log. Experiments were run on 16 real system log data, and the performance of the proposed algorithm was compared with the one of other 14 algorithms. It was shown that the HCLPars outperformed the other log parsers in terms of accuracy, efficiency, and robustness.

Keywords-event log mining; system logs; log parsing; log analysis; log management; execution trace; HCLPars; agent

I. INTRODUCTION

Each system or application has its own log files containing detailed information about the operating time (execution traces). These execution traces play an important role in developing, maintaining, and sustaining software systems. They help developers and support engineers to understand the system behavior [1, 2] and track and diagnose errors and malfunctions that may arise [3, 4]. But despite the enormous information buried in the logs, finding ways to effectively analyze it remains a huge challenge [5], for two reasons: First, modern software systems routinely generate tons of records in seconds. This huge volume of logs makes it difficult to inspect log messages manually. Second, these log messages are unstructured in nature. To be able to analyze such files, the first and most important step is logging parsing, which is converting the raw log messages into a sequence of structured events [6-9]. As the example illustrated in Figure 1, each raw log message records a specific system event with a set of fields: timestamp, verbosity level (e.g. ERROR/INFO/DEBUG), component, and event. A raw log message has constant and variable parts. The constant part reveals the log key or event template for the log message, which remains the same for every event occurrence, and varies from event to event, while the variable part records runtime information (i.e. parameters and states), which may vary among various event occurrences. The goal of log parsing is to automatically separate the constant part and the variable part of a raw log message, or otherwise match each raw log message with a specified log key (constant part) [10]. So, we need to extract the log keys first, and then use it in the parsing process.

The traditional method of extracting log keys relies on handcrafted regular expressions [11]. Simple as it may seem, writing custom rules manually for a large volume of records is a time-consuming and error-prone method, and the logging code is frequently updated in modern software systems, leading to regular reviewing of these rules. To reduce the manual efforts in extracting log keys, some studies [12, 13], have suggested techniques for extracting log keys directly from the source code. These technologies are applicable in some cases, but in practice, the source code is not always accessible, and

often these technologies are limited to specific software or applications. Meanwhile, a static and generic analysis tool is needed for all programs across different programming languages, and to achieve this, several data-driven approaches have been proposed, including iterative mining (SLCT [14]), iterative segmentation (IPLoM [15]), and hierarchical clustering (LKE [16]). In contrast to handcrafted rules and source-based parsing, these methods can learn patterns from log data and automatically generate common log keys, but are unable to handle huge datasets, so a parallel log parsing method, called POP was proposed [17]. POP is able to handle large datasets with high accuracy, as it was implemented on top of Spark, and it used Resilient Distributed Datasets (RDD) abstraction, which is ineffective with data of small size because it consumes more time, while using it is expensive when the size of the data increases [18]. To address these issues, we propose the Hierarchical Clustering Log Parsing method (HCLPars), that works on top of Spark like POP [17], but with a different abstraction type, which is Data Frame (DF). HCLPars was evaluated on large-scale real-world data sets, and the results demonstrate its ability to achieve speed, accuracy, and efficiency. For example, HCLPars can parse an HDFS dataset in less than one minute, while POP requires 7 minutes and IPLoM 30 minutes and LenMa and LogSig fail to finish in a reasonable time.

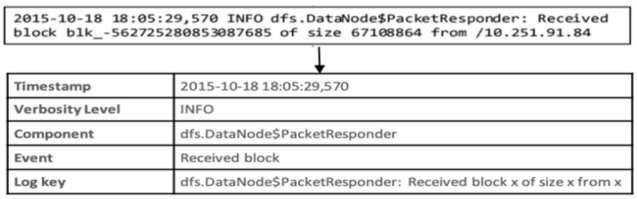

Fig. 1. Example of raw log message and log key.

II. RELATED WORK

Log key extraction has been studied extensively and has been categorized into three approaches: rule-based, source-code-based, and data-driven. Many researchers use rule-based methods [19, 20], which despite their accuracy, require domain expertise and are also limited to specific rather than general application scenarios. For example, authors in [19] a rule-based system for software failure analysis, taking advantage of artifacts that were produced at the time the system was designed and establishing a set of rules to formalize the placement of registration instructions within the source code. Authors in [21] proposed the novel Beehive system, which identifies potential security threats for a large volume of logs by unsupervised collecting of specific data features, then manually categorizing outliers. Source-code-based analysis has been used to extract a log key. But it is ineffective because the source code is often unavailable or incomplete to access. Meanwhile, most modern systems incorporate open-source software components written by hundreds of developers. For example, authors in [13] proposed a general methodology for log analysis based on source code analysis to discover large-scale system issues by extracting log events from console logs. Authors in [1] proposed an automated approach for log analysis to extract log keys directly from the source code and then produce an ordered list of all possible event occurrences. Also, several data-driven approaches have been proposed, which have the advantage that they do not require domain expertise and can learn patterns from log data and automatically generate shared registry key templates. For example, in [22] a new clustering algorithm (SLCT) that analyzes the log file using frequent pattern mining was presented, which helps discover frequent patterns and identify anomalous lines in log files. Authors in [16] presented a novel algorithm called IPLoM (Duplicate Partition Log Mining) to extract the log keys from event logs. It performs a 3-step hierarchical partitioning process of the log using unique log message properties. Authors in [17] proposed a technique for log analysis to detect anomalies. It first preprocesses the data using empirical rules and then performs hierarchical clustering of log messages using weighted edit distance, and finally, log keys are created from the resulting clusters. But although the overall accuracy of these log parsing approaches is high, they are not effective in datasets whose logs are growing at a large scale (for example, 100 million record messages), as these approaches fail to complete in a reasonable time (e.g. 1 hour), and most of them can't handle such data on a single computer. More recently, authors in [3] proposed a log parsing method through the parallelization on Spark called POP. POP handles logs with simple domain knowledge. Then, it uses iterative partitioning rules to divide the logs hierarchically into different groups. Then, the static parts are extracted to create the event log. Finally, similar groups are combined using hierarchical clustering to create the log keys. It is a basic system for processing large-scale data using the parallelization power of computer clusters. POP was implemented on top of Spark and used RDD abstraction. RDD cannot modify the system to work more efficiently and uses sequencing and garbage collection techniques, increasing the load on the system's memory and thus slowing the execution of operations.

Looking at the issues of the existing research, we suggest a new method to extract the log key. This method is based on a data-driven approach to analyze the execution log, which was built on top of Spark and used the DF abstraction. DF abstraction is characterized by its efficiency in analyzing files with high accuracy without the knowledge of the program, as well as its high speed in analyzing files of any size. We used HashCode and Equals method to find similarities between messages to ensure accuracy and speed instead of using the iterative partitioning used in [3], which consists of more than one steps, increasing execution time while it is not accurate enough. We used an intelligent Agent that does this process and we called it the Prepossessing Agent. A comparison between the reviewed papers is illustrated in Table I.

III. METHODOLOGY

A log message usually records a run-time behavior of the program, including events, state change, and interactions between components. It often contains two types of information: a free-form text string that is used to describe the semantic meaning of a recorded program behavior and a parameter that is used to express some important characteristics of the current task.

TABLE I. SUMMARY OF THE EXISTING RESEARCHES FOR LOG KEY EXTRACTION

Ref.	Approach	Data-driven approach	Use of Spark	Abstraction
[19]	Rule-based	-	No	-
[21]	Rule-based	-	No	-
[13]	Source-code-based	-	No	-
[1]	Source-code-based	-	No	-
[21]	Data-driven	Frequent pattern mining	No	-
[16]	Data-driven	Iterative partitioning	No	-
[17]	Data-driven	Hierarchical clusterin	No	-
[3]	Data-driven	Iterative partitioning + Hierarchical clustering	Yes	RDD
HCLPars	Data-driven	Hierarchical clustering	Yes	DF

In general, due to the various parameter values, the number of different types of log messages is massive or even infinite. Thus, the dimension problem of the direct consideration of log messages as a whole during log data mining may be troublesome. To resolve this problem, we replace every log message with its corresponding log key to perform analysis. A log key is defined as the common content of all log messages which are printed in the source code by the same log-print statement. The parameters are defined as a variable value printed by the log-print statement. In other terms, without any parameters, a log key equals the free-form text string of the log-print statement. For example, the key log message 1 and 3 is "INFO dfs.DataNode$PacketResponder: PacketResponder x for block x terminating" (shown in Figure 2). We analyze logs based on log keys for the following reasons:

- Different log-print statements often output different log text messages. Each specific log-print statement in the source code corresponds to a specific type of log key. So, a sequence of log keys will reveal the system's execution path and this will help us predict failures during implementation by identifying the normal execution path of the system.
- The number of key log message types is limited and much less than the number of raw log message types. It can help us avoid the dimension issue while extracting and analyzing data. It also provides a simplified view of all the events that occurred while the system was running for administrators.

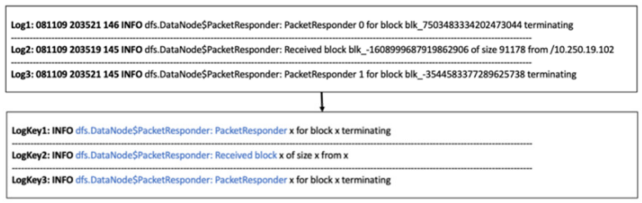

Fig. 2. Example 1 of raw log message and key log message.

The difficulty here is in identifying the log keys because we do not know which log messages are printed by the same statement print or the location of the parameters in the log messages. Generally, the log messages printed by the different log-print statements are often completely different, while the messages printed by the same statements are completely similar to each other. According to this observation, we can use clustering techniques to group the log messages printed by the same statement together and then find their common part as the log key. Parameters can cause some clustering mistakes because some of the different log messages contain a lot of matching parameter values, for example, raw log messages 1 and 2 have many similar parameters (shown in Figure 3). So, the Prepossessing Agent will remove the parameter values first, according to acquired knowledge, to avoid errors. Then, it groups the similar raw log messages and finds the common parts in each group to get the log keys.

A. Erasing Parameters via Acquired Knowledge

The parameters are mostly in the form of numbers, IP addresses, URIs, or follow special symbols like the colon or the equal sign. They are often included in parentheses or square brackets. It is easy to classify such content. Therefore, simple regular expression rules are often used to recognize and remove these parameters [23], for example, removing block ID in Figure 3 by "blk [-?[0-9]]+". For parameter removal (IP addresses, numbers, etc.) we used gained knowledge from previous research to define the types of parameters for each dataset (shown in the second block in Figure 5).

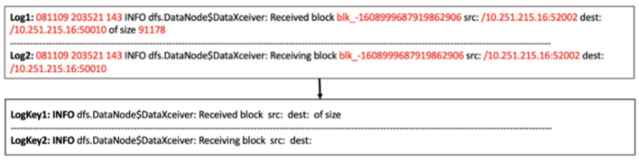

Fig. 3. Example 2 of raw log message and key log message.

B. Removing the Duplicate Raw Log Key

In this step, the Prepossessing Agent is partitioning the raw log keys into groups, so we need to find a proper metric to find the similarity between the raw log keys. We found that the HashCode and Equals method is the best metric for finding the similarity between raw log keys, because most programmers tend to write log keys firstly, and add the parameters afterward.

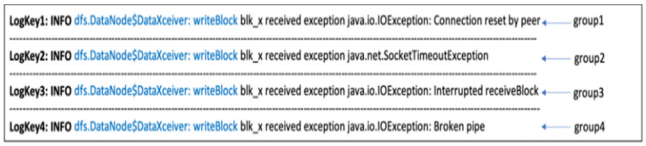

Fig. 4. Four log messages from the HDF dataset.

So, the log keys printed in the source code by the same log-print statement have the same HashCode. Based on this observation, the Prepossessing Agent is generating a hash code for all raw log messages. Then it puts the raw log messages that have the same hash code in the same group by using the Equals method to find matches between them. After selecting all the matching raw log messages, the Agent removes the duplicate results in each group, and it gets the log keys, while their number is limited. Examples of log keys are shown in the third block in Figure 5.

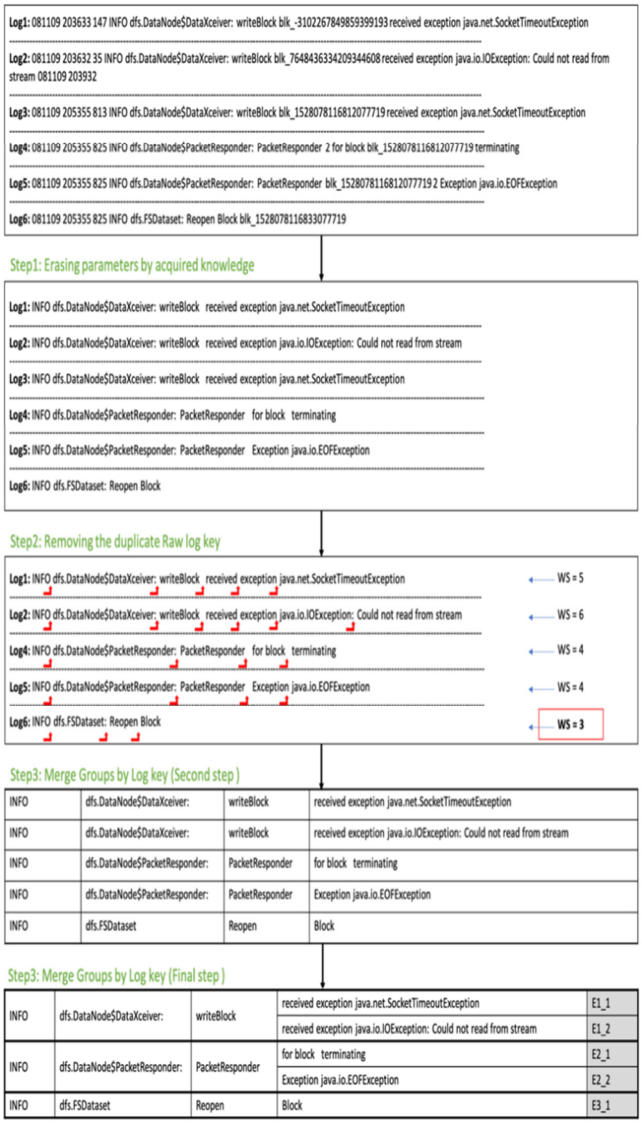

Fig. 5. Step of extracting log keys.

C. Merge Groups by Log Key

Each log key contains the component (server), log event, and message and most groups contain log keys that share the same component and log event but differ in the message. For example, Figure 4 contains 4 log messages from the HDFS dataset. The four log keys that contain the same component are dfs.DataNode$DataXceiver, and the same log event is writeBlock, but differ in the message. To improve parsing accuracy, the Prepossessing Agent clusters similar groups based on the component and their log events, through several steps.

1. The first step is to generate a regular expression that will be used to separate the log by using the whitespaces (WS) shared between all log keys in all groups as separators. Algorithm 1 provides the pseudo-code of Generating regular expressions. The Prepossessing Agent considers every log key from the previous step as a sentence and stores them as a list of sentences (line 1). The number of WS in each log key (sentence) is calculated, and the results are added into the LengthLogKeys list (lines 6-13). Then it finds the smallest common number of WS from LengthLogKeys list (line 14). For example, the third Block in Figure 5 contains 5 log messages from the HDFS dataset. These 5 log keys contain a different number of WS (6, 3, 4), whereas the smallest common number of WS is 3. The last step is generating a regular expression, where it replaces the number of WS from the previous step with the symbol ("\s"), for example, if the smallest common number of WS is 3, then the regular expression will be: ([^\s] +\s) ([^\s] +\s) ([^\s] +. *). Finally, the regular expression is returned (line 16).

2. The second step is to use a regular expression from step 1 to separate all the log keys in groups. In this step, every log key is transformed from sentence to columns, to find the common parts between the log keys in all groups. The fourth block of Figure 5 provides some examples of separated log keys.

```
Algorithm 1: pseudo code for generating a
regular expression
Input: a list of log keys from step B
(LogkeysL)
Output: regular expression for common
space between all log keys
1: ListLogKeys ← LogkeysL
2: LengthLogKeys ← => List() (initialize
with empty list)
3: RegEx ← NULL => (initialize with empty
string)
4: SmallestL ← NULL => (initialize with
empty integer)
5: LLog ← NULL => (initialize with empty
integer)
6: CurLog ← First log in ListLogKeys
7: while (Curlog has white-space) = true
do
8: LLog ← compute a white-space in CurLog
9: add LLog to LengthLogKeys
10: remove CurLog from ListLogKeys
11: CurLog ← next log in ListLogKeys
12: until ListLogKeys is empty
13: End while
14: SmallestL ← find smallest length in
LengthLogKeys
15: RegEx ←generation of regular
expression
16: return RegEx
```

3. In the final step, the Prepossessing Agent uses hierarchical clustering [22-25] to cluster similar groups based on components and their log events. The groups in the same cluster will be merged. This step assumes that if logs from different groups have the same component and log event type, then the texts of the component and the log events

that generated from these groups will be similar, so the the Hamming distance [25] is calculated between the texts (component and log event) of two logs to assess the similarity between them.

$$dist_A(a,b) = def \begin{cases} 0 & if \quad a = b \\ 1 & otherwise \end{cases} \qquad (1)$$

where a is the value of the component in the first log and b is the value of the component in the other log. If a and b are equal, it returns 0, otherwise 1. For example, we have 3 components, the first is "dfs.DataNode$DataXceiver:", the second is "dfs.FSDataset:", and the third is "dfs.DataNode$DataXceiver:". We calculated the Hamming distance between these and found that the first and the third are classified in the same group because they are exactly equal in value. We used the Hamming distance because it is a very practical metric for measuring the similarity and difference between data strings. Besides, the Hamming distance is intuitive, which makes parameter adjusting easier. After the above steps, we obtained the log key set (shown in the fifth block in Figure 5) from the training log messages in the training log files. The first part refers to the event (E1) and the second part refers to the message (1). With this step, we were able to know each component of the system, what events are issued by it, and what message types are issued for each event. So, this step provides the administrators with an overview of the system and what messages are issued from it.

IV. IMPLEMENTATION

The Prepossessing Agent uses Spark to make a large-scale analysis of records efficient [22, 26]. Spark is a platform for quickly processing data on a large scale and can also distribute data processing tasks across multiple computers, either alone or in conjunction with other distributed computing tools. Apache Spark offers three data abstractions: RDD, DF, and DS. In HCLPars, we use the DF API for several reasons: First, the DF resolves performance and measurement limitations that occur while using the RDD. Second, it uses input optimization engines, for exemplify, Catalyst optimizer, to process data efficiently. We can use the same engine for all Java, Python, R, and Scala DataFrame API. Third, it provides a schematic view of the data, meaning that the data has some meaning when it is stored, and this serves to provide a simplified view of the data for the administrators. Fourthly, DF optimally manages memory, it stores data outside the heap but still inside RAM (outside the main Java Heap), which in turn reduces garbage collection overload, while RDD stores data in memory (inside the main Java Heap). Lastly, it is characterized by flexibility and scalability. It supports various formats of data and can be combined with many other big data tools.

In our case, a DF can represent execution traces, where each message in execution traces is a row. Each step of the HCLPars requires specific tasks that are executed on every message. To speed up these tasks and execute them with high accuracy, we invoke Spark DF specially designed operations to work in parallel. Figure 6 illustrates the implementation of the Prepossessing Agent on Spark. The numbered arrows represent the interactions between the Spark cluster and the main program, where the main program works at Spark driver, which is responsible for allocating Spark tasks to workers in the Spark cluster [27]. For the Spark application, in Step 1, the Prepossessing Agent uses sqlContext.read.text() to read the text file (e.g. HDFS execution traces), converts every message or line into a single row at a single string column called value (DF), and loads the DF to the Spark cluster (arrow 1). Then, it uses withColumn() to preprocess all log messages (erasing parameters) (arrow 2). After preprocessing, it caches the preprocessed log messages as schema in off-heap memory and returns a DF as the reference (arrow 3). In step 2, it uses distinct() to drop duplicate rows (in the column) from the DF (arrow 4) and return them (arrow 5). In step 3, it generates regular expressions for all log messages (arrows 6, 7) as described above. Then, the driver program separates the column (value) into many columns based on the regular expression from the previous step and adds them into new DF (arrows 8, 9). When all the columns are separated, it runs hierarchical clustering on them, and then it uses groupby() to merge log message (row) based on the clustering result (arrow 10). Finally, the merged DF (log keys) are outputted as a CSV file by use coalesce(1).write() (arrow 11).

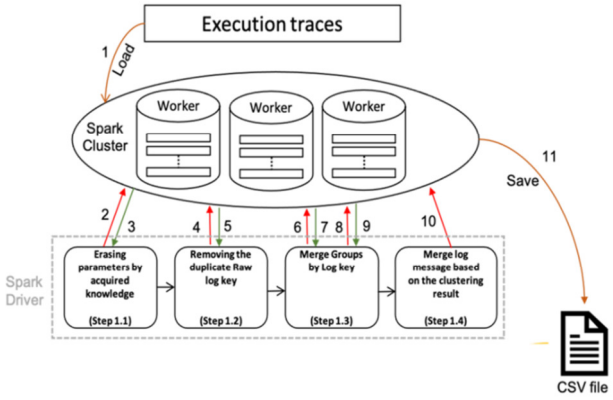

Fig. 6. Extracting log key steps.

V. DATA COLLECTION AND EVALUATION

In this section, we present the data sets that were used in the evaluation process and the evaluation methodology. The performance of HCLPars is evaluated in terms of its accuracy, efficiency, and effectiveness and the results are compared with those of existing log parsers.

A. Data Collection

The loghub dataset [6] were used during the training and test phases. Loghub is a large collection of logs from 16 real-world systems, including operating systems, mobile phone systems, distributed systems, supercomputers, standalone software, and server applications. All these logs are over 77 GB in size and contain 440 million log messages. Table II provides a summary of the dataset. The columns are marked with the symbol (#), as in [29].

B. Evaluation

The parameters of the log parsers are fine-tuned through over 8 runs and the best results are reported to avoid the randomization bias.

1) Evaluation Measures

Accuracy, robustness, and efficiency were considered for the evaluation of the results [29].

- Accuracy is a measure of the ability of a log parser to distinguish between fixed and variable parts. Therefore, we define the accuracy metric of parsing as the ratio of properly parsed log messages to the total number of log messages. A log message is parsed correctly if its event template matches one of the previously extracted log message templates.
- Robustness of a log parser is measured by the extent of its ability to work continuously within different datasets or different sizes with the same efficiency.
- Efficiency is measured by the amount of time it takes the parser to parse the data. The less time spent, the higher the efficiency.

TABLE II. SUMMARY OF THE LOGHUB DATASET

Dataset #	Description #	Log size #	Templates (total) #
HDFS	Hadoop distributed file system log	11,175,629	30
Spark	Spark job log	33,236,604	456
Hadoop	Hadoop mapreduce job log	394,308	298
ZooKeeper	ZooKeeper service log	74,380	95
OpenStack	OpenStack software log	207,820	51
Linux	Linux system log	25,567	488
Mac	Mac OS log	117,283	2,214
Thunderbird	Thunderbird supercomputer log	211,212,192	4,040
BGL	Blue Gene/L supercomputer log	4,747,963	619
HPC	High performance cluster log	433,489	104
Apache	Apache server error log	56,481	44
OpenSSH	OpenSSH server log	655,146	62
Proxifier	Proxifier software log	21,329	9
Android	Android framework log	30,348,042	76,923
Health app	Health app log	253,395	220

2) Evaluation Procedure

To evaluate HCLPars, we compared it with 14 log parsers by using 16 standard datasets. The log parser parameters were finely tuned through more than 8 runs, and the best results were recorded.

VI. RESULTS

A. Accuracy

In this part, we evaluate the accuracy of HCLPars, and compare it with the accuracy of 14 existing log parsers. To make the comparison fair, we performed accuracy experiments on subsets of the original log datasets, each containing 2,000 log messages.

Table III presents the accuracy results of the log parsers evaluated in 16 log datasets. Each column indicates the accuracy for 1 log parser across the datasets, helping define its robustness across different log types. Each row represents the accuracy of the parsing for different log parsers in a single dataset. For ease of observation, we marked the accuracy values greater than 0.9 in boldface, and highlighted the best accuracy with an asterisk (*). We can observe that most of the datasets were parsed accurately (more than 90%) by at least 2 log parsers. Totally, 12 out of the 15 log parsers provide the best accuracy on at least 3 log datasets. To measure the overall effectiveness of the log parsers, we calculated the average accuracy of each log parser across different datasets, as shown in the last row of Table III. We can observe that HCLPars is the most accurate on average with a score of 0.9605, achieving high accuracy (over 0.9) in 12 out of 16 datasets. It was followed by POP, which achieved high accuracy in 10 datasets.

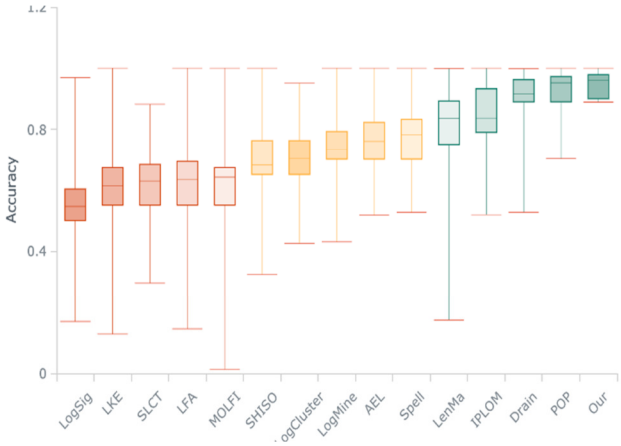

Fig. 7. Accuracy distribution of the log parsers in different types of logs.

B. Robustness

Robustness is an important measure of the practical use of a log parser. In this part, we evaluate the robustness of HCLPars and compare it with the existing log parsers from 2 aspects: across different types and sizes of logs.

Figure 7 (boxplot diagram) indicates the accuracy distribution of each log parser across the 16 log datasets. For each box, the highest point of the vertical line corresponds to the maximum accuracy values, while the lowest point corresponds to the minimum accuracy values. In Figure 7, from left to right, the log parsers are arranged in ascending order of average accuracy. It can be noted that HCLPars has the highest average accuracy. This means that it can efficiently parse different types of log data, as its minimum accuracy is 0.889. Additionally, we evaluated the robustness of HCLPars on different log sizes. We sampled 40 original real-world datasets, such as HDFS, BGL, Spark, Hadoop, ZooKeeper, Open-Stack, HPC, and Proxifier (Table III). HDFS, Spark, Hadoop, ZooKeeper, OpenStack are log files from distributed systems, while HDFS, BGL, HPC, ZooKeeper, and Proxifier were used in [17, 20]. For log dataset, we changed the size based on its total size. For example, HDFS has a total size of 1.47 GB. We changed its size to 300 KB, 1 MB, 10 MB, 100 MB, and 1 GB, successively. Table IV shows the number of raw log messages in the datasets. Each row presents 5 sample datasets created from a real dataset. We chose the log parsers that achieved high accuracy in more than 4 log datasets, i.e. AEL, IPLOM, LenMa, MoLFI, Spell, Drain, and POP and compared them with HCLPars.

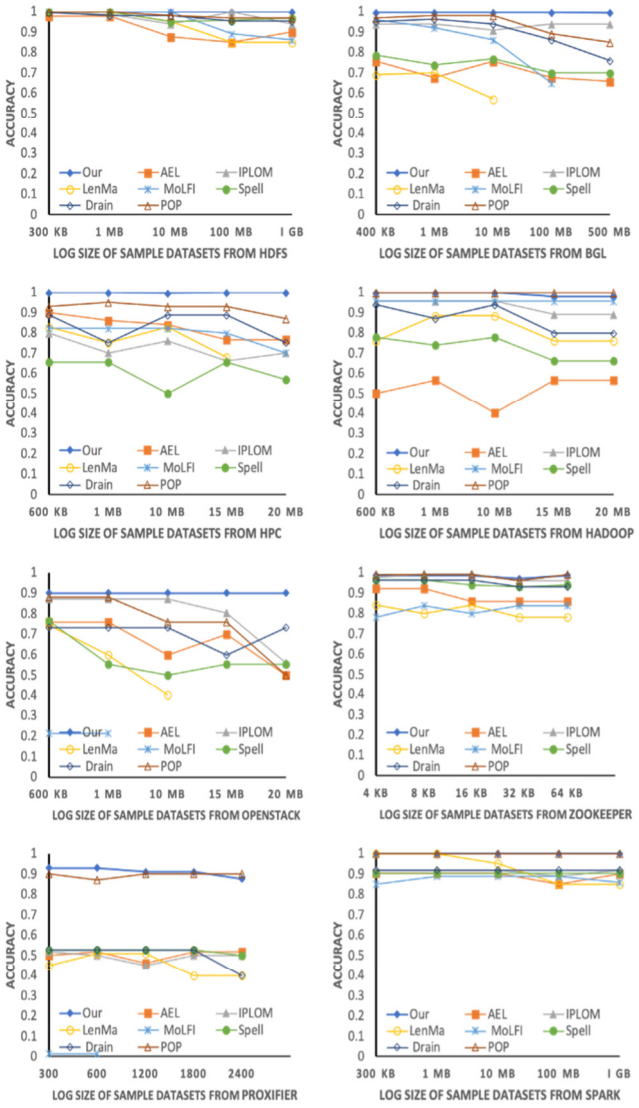

Fig. 8. Log parser accuracy on dataset log size.

Figure 8 shows the results of the parsing accuracy on different log dataset volumes. Note that some lines are incomplete in the Figure because some parsers, such as MoLFI and LenMa, cannot finish the parsing in a reasonable period (4 hours). The results show that POP works continuously in most cases except for the 0.24 drop in OpenStack, while the rest of the log parsers have clear drops in accuracy or clear fluctuations with increasing data volume in most datasets (except for HDFS, ZooKeeper, and Spark). The experimental results of HCLPars are shown in Table IV and Figure 8. Note that the accuracy of HCLPars is very consistent for all datasets. The accuracy on HDFS and Spark is 1 for all 5 samples. For BGL and Hadoop, the fluctuation of the accuracy is 0.001 at most. For Proxifier and ZooKeeper, the fluctuation of the accuracy is 0.02 at most. When compared to the other parsers, HCLPars is the only to obtain consistently high accuracy in all datasets.

C. Efficiency

Efficiency is an important aspect to consider when parsing log data. To evaluate the efficiency of the log parser, we record the runtime it takes to complete the entire parsing process. Like previous experiment settings, we evaluate the runtime of log parsers on 40 sampled datasets from original real-world datasets. The results can be seen in Figure 9. It is obvious that the size of the log is directly proportional to runtime, i.e. parsing time increases with log size. It is also obvious that the efficiency of the log parser depends on the number of event templates. The simpler the log data, containing a limited number of templates, the easier the parsing process. The efficiency of the log parser is shown when there are many log templates. Drain and IPLoM have better efficiency, which scales linearly with log size. POP has better efficiency with large data. AEL and Spell do not scale well with many event templates. LenMa and MoLFI do not scale well with large data.

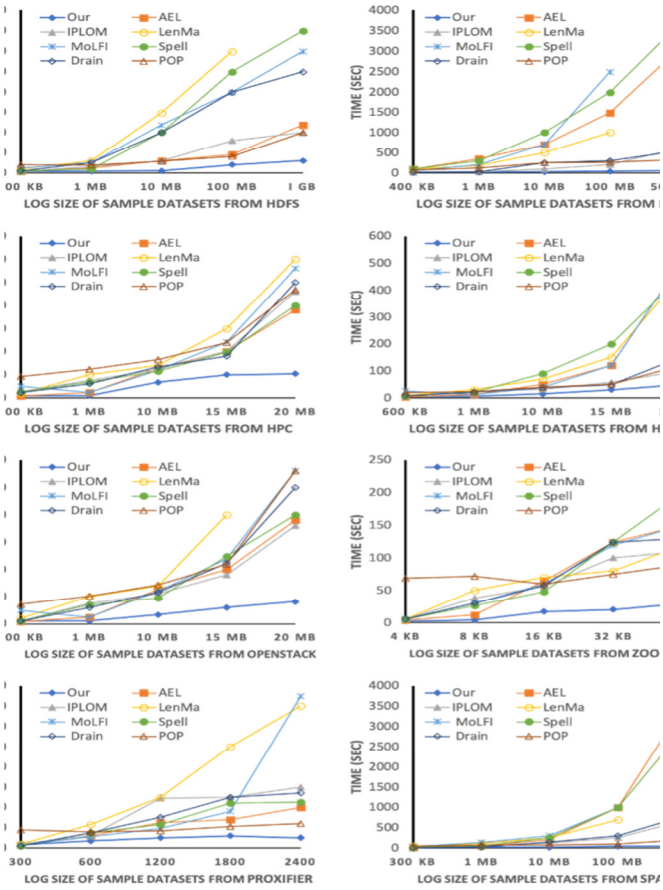

Fig. 9. Running time of log parsers on dataset log size.

For instance, BGL contains 619 event tem plates. POP can finish parsing within 5 min, while Drain and IPLoM take 10 min. AEL and Spell take a long time to complete parsing (1 hour), while LenMa and MoLFI cannot finish parsing 1 GB of BGL in 2, 4, or 6 hours, respectively. The experimental results of HCLPars are also shown in Table VI and Figure 9.

TABLE III. ACCURACY OF LOG PARSERS ACROSS DIFFERENT LOG TYPES

Dataset	SLCT	AEL	IPLOM	LKE	LFA	LogSig	SHISO	Log cluster	LenMa	Log mine	Spell	Drain	MoLFI	POP	HCLPars	Best
HDFS	0.455	**0.978**	**1***	**0.998**	0.875	0.800	**0.978**	0.546	**0.998**	0.851	**1***	**0.997**	**0.998**	**1***	**1***	**1**
Hadoop	0.432	0.567	**0.956**	0.700	**0.900**	0.654	0.865	0.566	0.885	0.867	0.778	**0.938**	**0.957**	**0.998**	**0.955***	**0.999**
Spark	0.685	0.905	**0.920**	0.634	**0.994**	0.544	**0.920**	0.795	0.887	0.576	**0.905**	**0.920**	0.418	**0.999**	**1***	**1**
Zookeeper	0.726	**0.921**	**0.992***	0.578	0.839	0.700	0.660	0.789	0.841	0.688	**0.964**	**0.967**	0.839	**0.990**	**0.987**	**0.992**
OpenStack	0.867	0.758	0.871	0.787	0.200	0.200	0.722	0.696	0.743	0.743	0.764	0.733	0.213	0.880	**0.900***	**0.900**
BGL	0.573	0.758	**0.939**	0.128	0.854	0.227	0.711	0.835	0.69	0.723	0.787	0.963	**0.960**	**0.990**	**0.996***	**0.996**
HPC	0.839	**0.900**	0.800	0.574	0.817	0.354	0.325	0.788	0.830	0.784	0.654	0.887	0.824	**0.950**	**1***	**1**
Thunderb	0.882	**0.941**	0.663	0.813	0.649	0.694	0.576	0.599	**0.943**	**0.919**	0.844	**0.955***	0.646	**0.955***	**0.955***	**0.955**
Mac	0.558	0.764	0.673	0.366	0.555	0.478	0.595	0.604	0.698	0.872	0.757	0.787	0.636	**0.889***	**0.889***	**0.889**
Windows	0.697	0.690	0.567	**0.990**	0.588	0.689	0.701	0.713	0.566	**0.993**	**0.989**	**0.997**	0.406	0.876	**1***	**1**
Linux	0.297	0.673	0.672	0.519	0.279	0.169	0.701	0.629	0.701	0.612	0.605	0.690	0.284	0.701	**0.894***	**0.894**
Android	0.882	0.682	0.712	**0.909**	0.616	0.548	0.585	0.798	0.880	0.504	**0.919***	**0.911**	0.788	0.876	**0.919***	**0.919**
HealthApp	0.331	0.568	0.872	0.592	0.549	0.235	0.397	0.531	0.174	0.684	0.639	0.780	0.440	0.772	**0.900***	**0.900**
Apache	0.731	**1***	**1***	**1***	**1***	0.582	**1***	0.709	**0.999**	**1***	**1***	**0.998**	**1***	**1***	**1***	**1**
OpenSSH	0.521	0.538	0.802	0.426	0.501	0.373	0.619	0.426	**0.925**	0.431	0.554	0.788	0.500	**0.998**	**0.999***	**0.999**
Proxifier	0.518	0.518	0.519	0.455	0.145	**0.969***	0.517	0.951	0.508	0.517	0.527	0.527	0.013	**0.900**	**0.930***	**0.969**
Average	0.624	0.760	0.809	0.614	0.647	0.513	0.679	0.702	0.835	0.735	0.782	0.864	0.640	0.923	**0.960**	

TABLE IV. LOG SIZE OF SAMPLE DATASETS

Dataset	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5
HDFS	300 KB	1 MB	10 MB	100 MB	1 GB
BGL	400 KB	1 MB	10 MB	100 MB	500 MB
Spark	300 KB	1 MB	10 MB	100 MB	1 GB
Hadoop	600 KB	1 MB	10 MB	15 MB	20 MB
ZooKeeper	4 KB	8 KB	16 KB	32 MB	64 KB
OpenStack	600 KB	1 MB	10 MB	15 MB	20 MB
HPC	600 KB	1 MB	10 MB	15 MB	20 MB
Proxifier	300 KB	600 KB	1200 KB	1800 KB	2400 KB

TABLE V. ACCURACY OF HCLPARS ON THE SAMPLE DATASETS OF TABLE III.

Dataset	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5
HDFS	1	1	1	1	1
BGL	0.996	0.996	0.996	0.996	0.995
Spark	1	1	1	1	1
Hadoop	0.999	0.999	0.999	0.998	0.999
ZooKeeper	0.987	0.987	0.987	0.980	0.987
OpenStack	0.900	0.900	0.900	0.900	0.900
HPC	1	1	0.996	1	1
Proxifier	0.93	0.93	0.91	0.91	0.93

TABLE VI. RUNNING TIME OF HCLPARS (sec) ON SAMPLE DATASETS IN TABLE III.

Dataset	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5
HDFS	4.67	4	5	10	30
BGL	15.98	20.98	34.76	40.15	60
Spark	8.403	18.7	20	44.67	50.89
Hadoop	7	7.98	15	30.38	50
ZooKeeper	2.120	4.203	18.23	23.78	31.67
OpenStack	5.21	6	17.95	30	45
HPC	4.98	5.45	34	50	52
Proxifier	3.09	7.56	10.45	12.33	10.94

Obviously, the runtime that HCLPars consumes is directly proportional to the log size, and the run time of HCLPars does not exceed 1 minute for the largest dataset size (1 GB). Compared to the other log parsers, HCLPars conducts the fastest parsing, as it can parse large logs in a record time that does not exceed 1 minute.

VII. CONCLUSION

This paper studied the automated parsing for large system event logs. Initially, a comprehensive study was conducted on the existing log parsing methods and the way they work. Based on the result, the Automatic Log Parsing (HCLPars) method using Spark was proposed, which consists of three steps: removing parameter values according to acquired knowledge, grouping of raw log messages based on similarity, and finding the common parts in each group to get the log keys. Many experiments were conducted on 16 sets of real-world data logs. The results from these experiments indicate that HCLPars is very effective, as it works accurately and efficiently on all types of data logs, regardless of their size. In the future, we hope to test this method on more data logs.

ACKNOWLEDGEMENT

This work was funded by the University of Jeddah, Jeddah, Saudi Arabia, under grant No. (UJ-20-123-DR). The authors, therefore, acknowledge with thanks the University of Jeddah technical and financial support.

REFERENCES

[1] J. Svacina *et al.*, "On Vulnerability and Security Log analysis: A Systematic Literature Review on Recent Trends," in *International Conference on Research in Adaptive and Convergent Systems*, Gwangju, Korea, Oct. 2020, pp. 175–180, https://doi.org/10.1145/3400286.3418261.

[2] J. Sun, B. Liu, and Y. Hong, "LogBug: Generating Adversarial System Logs in Real Time," in *29th ACM International Conference on Information & Knowledge Management*, New York, NY, USA, Oct. 2020, pp. 2229–2232, https://doi.org/10.1145/3340531.3412165.

[3] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: error diagnosis by connecting clues from run-time logs," in *Fifteenth International Conference on Architectural support for*

programming languages and operating systems, Pittsburgh, PA, USA, Mar. 2010, pp. 143–154, https://doi.org/10.1145/1736020.1736038.

[4] X. Xu, L. Zhu, I. Weber, L. Bass, and D. Sun, "POD-Diagnosis: Error Diagnosis of Sporadic Operations on Cloud Applications," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Atlanta, GA, USA, Jun. 2014, pp. 252–263, https://doi.org/10.1109/DSN.2014.94.

[5] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Communications of the ACM*, vol. 55, no. 2, pp. 55–61, Oct. 2012, https://doi.org/10.1145/2076450.2076466.

[6] X. Xie, Z. Wang, X. Xiao, Y. Lu, S. Huang, and T. Li, "A Confidence-Guided Evaluation for Log Parsers Inner Quality," *Mobile Networks and Applications*, vol. 26, no. 4, pp. 1638–1649, Aug. 2021, https://doi.org/10.1007/s11036-019-01501-6.

[7] H. Dai, "logram: efficient log paring using n-gram model," M.S. thesis, Concordia University, Montreal, QC, Canada, 2020.

[8] D. Aroussi, B. Aour, and A. S. Bouaziz, "A Comparative Study of 316L Stainless Steel and a Titanium Alloy in an Aggressive Biological Medium," *Engineering, Technology & Applied Science Research*, vol. 9, no. 6, pp. 5093–5098, Dec. 2019, https://doi.org/10.48084/etasr.3208.

[9] M. V. Japitana and M. E. C. Burce, "A Satellite-based Remote Sensing Technique for Surface Water Quality Estimation," *Engineering, Technology & Applied Science Research*, vol. 9, no. 2, pp. 3965–3970, Apr. 2019, https://doi.org/10.48084/etasr.2664.

[10] J. Zhu *et al.*, "Tools and Benchmarks for Automated Log Parsing," in *41st International Conference on Software Engineering: Software Engineering in Practice*, Montreal, QC, Canada, Dec. 2019, pp. 121–130, https://doi.org/10.1109/ICSE-SEIP.2019.00021.

[11] C. Gormley and Z. Tong, *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. Sebastopol, CA, USA: O'Reilly Media, 2015.

[12] M. Nagappan, K. Wu, and M. A. Vouk, "Efficiently Extracting Operational Profiles from Execution Logs Using Suffix Arrays," in *20th International Symposium on Software Reliability Engineering*, Mysuru, India, Nov. 2009, pp. 41–50, https://doi.org/10.1109/ISSRE.2009.23.

[13] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *22nd Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct. 2009, pp. 117–132, https://doi.org/10.1145/1629575.1629587.

[14] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *3rd IEEE Workshop on IP Operations & Management (IPOM 2003) (IEEE Cat. No.03EX764)*, Kansas City, MO, USA, Oct. 2003, pp. 119–126, https://doi.org/10.1109/IPOM.2003.1251233.

[15] A. A. O. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *15th ACM SIGKDD international conference on Knowledge discovery and data mining*, Paris, France, Jul. 2009, pp. 1255–1264, https://doi.org/10.1145/1557019.1557154.

[16] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A Lightweight Algorithm for Message Type Extraction in System Application Logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 11, pp. 1921–1936, Aug. 2012, https://doi.org/10.1109/TKDE.2011.138.

[17] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards Automated Log Parsing for Large-Scale Log Data Analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, Aug. 2018, https://doi.org/10.1109/TDSC.2017.2762673.

[18] Y. Ohno, S. Morishima, and H. Matsutani, "Accelerating Spark RDD Operations with Local and Remote GPU Devices," in *22nd International Conference on Parallel and Distributed Systems*, Wuhan, China, Dec. 2016, pp. 791–799, https://doi.org/10.1109/ICPADS.2016.0108.

[19] M. Cinque, D. Cotroneo, and A. Pecchia, "Event Logs for the Analysis of Software Failures: A Rule-Based Approach," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 806–821, Jun. 2013, https://doi.org/10.1109/TSE.2012.67.

[20] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in *ACM SIGSAC Conference on Computer and Communications Security*, Dallas, TX, USA, Nov. 2017, pp. 1285–1298, https://doi.org/10.1145/3133956.3134015.

[21] M. Zaharia *et al.*, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX conference on Networked Systems Design and Implementation*, Berkeley, CA, United States, Apr. 2012, pp. 1–14.

[22] T.-F. Yen *et al.*, "Beehive: large-scale log analysis for detecting suspicious activity in enterprise networks," in *29th Annual Computer Security Applications Conference*, New Orleans, LA, USA, Dec. 2013, pp. 199–208, https://doi.org/10.1145/2523649.2523670.

[23] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper)," in *The Eighth International Conference on Quality Software*, Oxford, UK, Aug. 2008, pp. 181–186, https://doi.org/10.1109/QSIC.2008.50.

[24] J. C. Gower and G. J. S. Ross, "Minimum Spanning Trees and Single Linkage Cluster Analysis," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 18, no. 1, pp. 54–64, 1969, https://doi.org/10.2307/2346439.

[25] E. F. Krause, "Taxicab Geometry," *The Mathematics Teacher*, vol. 66, no. 8, pp. 695–706, Dec. 1973, https://doi.org/10.5951/MT.66.8.0695.

[26] "Apache Spark™ - Unified Engine for large-scale data analytics," *Apache Spark*. https://spark.apache.org/.

[27] M. A. Biberci and M. B. Celik, "Dynamic Modeling and Simulation of a PEM Fuel Cell (PEMFC) during an Automotive Vehicle's Driving Cycle," *Engineering, Technology & Applied Science Research*, vol. 10, no. 3, pp. 5796–5802, Jun. 2020, https://doi.org/10.48084/etasr.3352.

[28] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics." arXiv, Aug. 14, 2020, https://doi.org/10.48550/arXiv.2008.06448.

[29] T.-K. Hu, T. Chen, H. Wang, and Z. Wang, "Triple Wins: Boosting Accuracy, Robustness and Efficiency Together by Enabling Input-Adaptive Inference." arXiv, Feb. 24, 2020, https://doi.org/10.48550/arXiv.2002.10025.

[30] W. Xu, "System Problem Detection by Mining Console Logs," Ph.D. dissertation, University of California, Berkeley, CA, USA, 2010.