Analysing the current log parsing techniques in large-scale software systems

Stefan Petrescu
TU Delft
Delft, Netherlands
petrescu-1@student.tudelft.nl

Abstract

Due to the complexity of nowadays software systems, the amount of logs generated is tremendous. As a result, it has become infeasible for humans to manually investigate this amount of data in reasonable time, thereby missing important information that could help making systems more efficient and more secure. Consequently, effectively utilizing logs requires automating the process of log analysis. By using log mining techniques, automated log analysis can tackle arbitrary system log sizes, thus complementing traditional log analysis methods. However, despite the unquestionable utility of log mining techniques, their effectiveness depends on the output quality of a process known as 'log parsing'. Thus, as this is of utmost importance, we examined the state-of-the-art log parsing approaches. For this, we conducted a thorough literature survey in which 34 log parsing methods were successfully identified. Furthermore, we evaluated the scalability and accuracy for the 14 most recognized in the literature. Our findings indicate that, out of the 14, only 4 are realistically scalable, namely Drain, IPLoM, LFA, LogCluster (these can handle log sizes of 20M lines in ~75 minutes, using ~25GiB of memory). With regards to their accuracy, the best results are given by NuLog (~95% over 9 datasets, for 2k log size datasets).

Keywords: log parsing, log abstraction, log signature extraction

1 Introduction

For large-scale software systems, logs record runtime information in order to provide an audit trail for monitoring, understanding the activity, or diagnosing problems [36]. For instance, in case of any incidents, system administrators can follow the logs and conduct investigations. Furthermore, in case of errors or anomalies, log messages can facilitate the process of generating alerts [40].

Unfortunately, the complexity of modern software ecosystems produces a deluge of log information that, due to the size and the low signal to noise ratio, often remains unanalyzed even though critical information could be destilled from the log content. For example, some systems can produce 30-50 gigabytes of logs per hour [20]. As a result of this, traditional log analysis approaches are no longer suitable. Thus, it has become infeasible for humans to manually investigate this amount of information in a reasonable amount of time. In order to realistically utilize system logs, automated log analysis techniques are required.

In their foremost step, the vast majority of automated log analysis methods require logs to undergo a specific type of transformation. In the literature, this process is known as 'log parsing'[1]. Log parsing transforms a raw log message (which is basically a logging statement in the source code) into structured information. More specifically, the goal is to separate between the static (parts of the initial logging statement) and the dynamic parts (information available at runtime) of a log message in order to create usable input for downstream tasks. Distinguishing between constants and variables means finding the static template of the raw log message, as displayed in Figure 1. Based on the specifics of the log dataset, conducting this type of transformation can be problematic, as it might be desirable to account for more than just accuracy (scalability can also be a very important aspect).

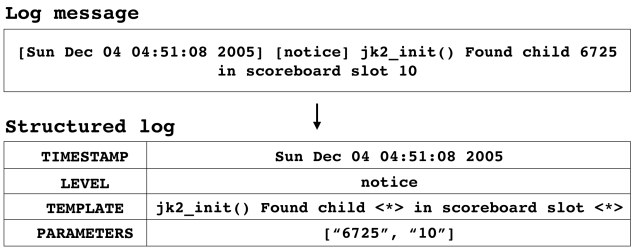

Figure 1. Log parsing example; a raw log message gets transformed into structured information. In other words, the main goal of running a log parser is to distinguish between constants and variables [12].

As the performance of automated log analysis techniques can be directly influenced by the output quality of the log parsing methods [10], it is crucial to find which of these are most appropriate. Hence, this study focuses on the state-of-the-art (SOTA) log parsing approaches in terms of their accuracy and scalability. More specifically, we tackle the following research questions:

[1]'Log parsing' is sometimes used interchangeably with 'event template extraction' or 'log abstraction'.

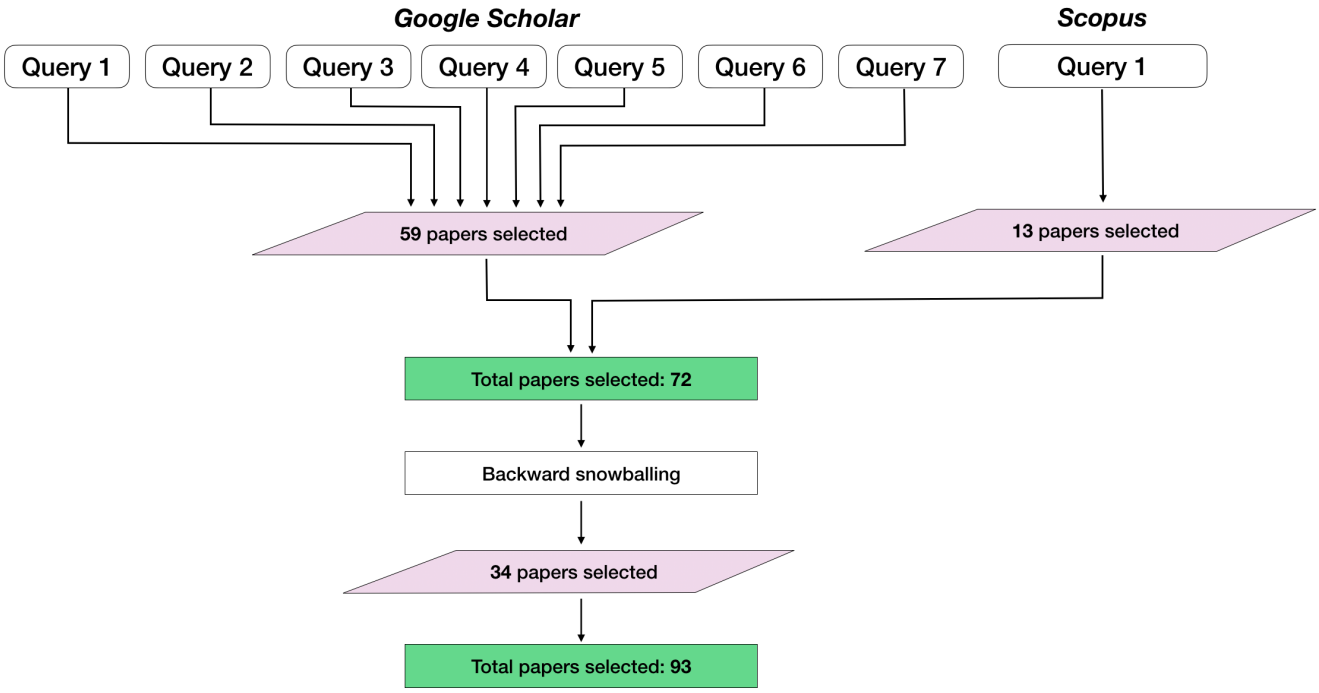

Figure 2. Selection process for the papers chosen for further investigation.

RQ1: How has log parsing been approached so far?
RQ2: How do log parsing methods perform in terms of their scalability?
RQ3: How do log parsing methods perform in terms of their accuracy?

By answering these questions, we hope to bring more structure and clarity to the discussion in order to guide future researchers in tackling log parsing.

In regards to the paper's outline, Section 2 presents the search methodology. Section 3 contains the answers to the research questions. Finally, the final section concludes this article and discusses the most important findings.

2 Method

Inspired by the methodology proposed in [16], we conducted the search for relevant papers. In Figure 2, an overview of the paper selection process can be found. Our goal here was to find all possible papers that contained information about log parsing. We considered a paper to be relevant (and thus, selected) either if, it proposed a log parsing method, had a log parser implemented in its pipeline, or had references to other log parsers/log parsing methods. This was decided by reading the title, abstract and, in case of uncertainty, by quickly reading the paper. Furthermore, we have to mention that we only selected papers that were about software logs (logfiles) and not (mathematical) logarithms. Lastly, we considered papers to be relevant only if they tackled software systems.

The selection process started by querying two well-known databases, Google Scholar[2] and Scopus[3]. For Google Scholar, we consulted the first 10 pages of results (first 100 results, any time, sort by relevance, any type). For Scopus, we consulted all of the 392 returned results. We tried to reduce unclarities as much as possible by thoroughly documenting the search and selection process. Thus, the queries used can be found in Appendix A. Additional information concerning the search method can be found in Section 3.

From a total of 93 selected papers, we investigated which of them actually proposed log parsing approaches. After this filtering process, we ended up with 34 papers, that were split in two main categories, namely online and offline approaches (details in Section 3). This process is visualised in Figure 3. Last but not least, during the writing of this document, we applied an additional filtering process, based on two aspects. Firstly, we excluded papers for which the authors explicitly mentioned that their method was preliminary (we only considered work that was feature complete). Secondly, we did not consider papers that had their technical description insufficient to reproduce.

[2] https://scholar.google.com
[3] https://www.scopus.com

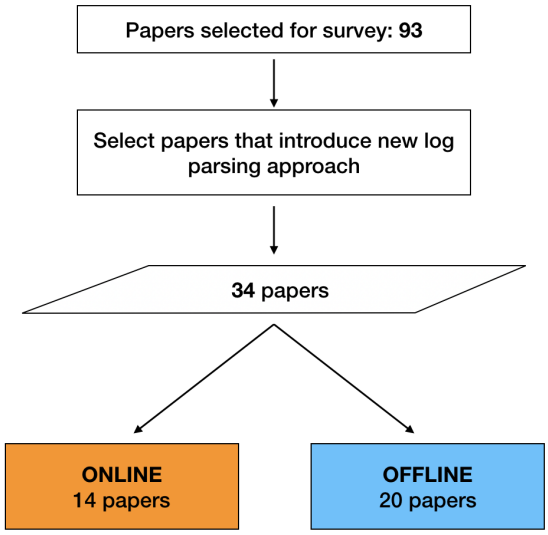

Figure 3. Final selection process of papers that proposed log parsing approaches.

3 Results

In this section we briefly present each log parsing approach found and mention its main contribution(s) and biggest disadvantage(s). Furthermore, two representations of how the methods are clustered are visualised. Each subsection tackles one corresponding research question.

3.1 Log parsing approaches

Inspired by similar works [7] [12], we distinguish between the methods by using their *mode*. This leads us into categorising the approaches in two main branches, *offline* and *online*. The distinction between these two modes is very important, and it has to do with the manner in which the log data is being processed.

Offline approaches process log data in batches, and are able to discover templates within the boundries of a given dataset. They require a training phase, during which the templates are discovered. After this, they are able to parse incoming logs by matching with the templates found previously, either in batch or stream [7]. As changes/updates in software can generate new log templates, one drawback of using this type of approach is that it requires the method to be re-run periodically in order to discover these.

Online approaches process log data item by item (in a streaming manner), without requiring (prior to the execution) an already available batch of data. More specifically, they are able to discover event templates without requiring an offline training phase. Furthermore, as event templates are being updated dynamically, such methods can be integrated seamlessly for downstream tasks [12]. Online parsers are recommended for use cases where the decision time for certain actions needs to be really short (e.g., trying to predict incidents in a software system) and logs need to be processed on the fly.

As a general overview, Table 1 contains a list of the selected offline log parsing approaches. In turn, Table 2 contains a list of the selected online log parsers.

Table 1. Overview of all offline log parsing approaches found in the literature.

Year	Method used
2003	SLCT [34]
2008	AEL [15]
2009	LKE [8]
2010	LFA [22]
2011	LogSig [31]
2012	IPLoM [18]
2014	NLP-LTG [17]
2013	HLAer [25]
2015	LogCluster [35]
2016	LogMine [9]
2017	NLM-FSE [33]
2017	POP [10]
2018	MoLFI [19]
2019	CLF [39]
2020	LPV [38]
2020	NuLog [24]
2020	ELA [30]
2021	AWSOM-LP [27]
2021	LogStamp [32]

3.1.1 Offline log parsing approaches. Figure 4 shows a clustering of the different offline methods based on their algorithmic approach. The connection between two methods is represented by an arrow. More specifically, we considered two methods connected, if either one of them evaluated against the other, or if one of them inherited its approach from the other. This way, we observe which types of algorithms test against which types – it can be noticed that most authors take into account others' methods when evaluating their own. Notable exceptions include the work on LPV, which renders the claimed benefits at least questionable. Furthermore, it can be seen that the biggest group is the one that uses clustering algorithms, whereas approaches that use heuristics or evolutionary algorithms have only one method present in their cluster. Last but not least, the figure suggests that the log parsing community is very active (indicated by the various connections between the methods/clusters).

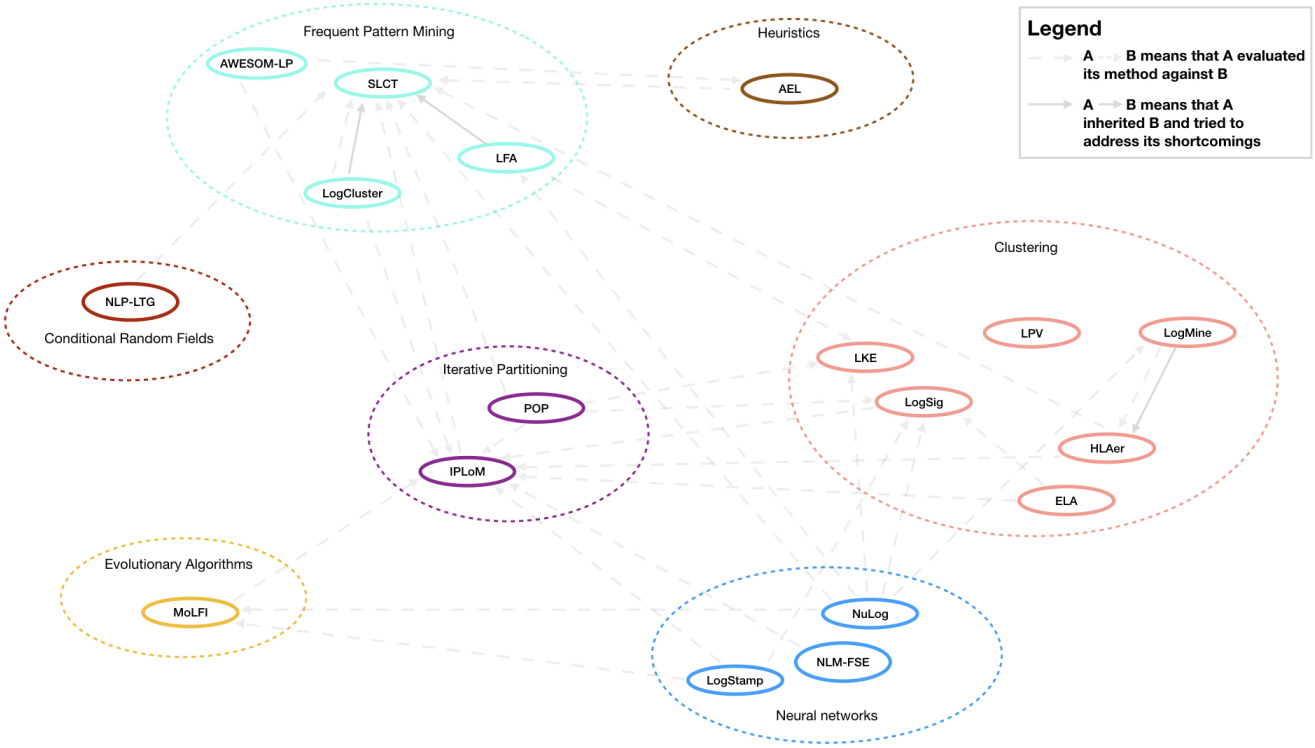

Figure 4. Graphical representation of how the offline methods are clustered together based on their algorithmic approach. The arrows represent connections between various methods. More specifically, an arrow means that the method from which it originates, evaluated against the method towards which it points.

SLCT (Simple Logfile Clustering Tool) [34] uses a data clustering algorithm to search for 'event types' (log templates). Finding these means discovering the constant parts of a log message – the same as log parsing. Practically, a frequent pattern mining algorithm is applied, consisting of 3 steps/data passes. During the first data pass, a table of frequent words is constructed (a word is considered to be frequent if it appears in the log data at least N times – a user specified threshold). For the second step, cluster candidates are formed. More specifically, log lines that contain one or more frequent words (found previously) are added to a candidate table (initially empty). The third step involves inspecting the candidate table previously generated, and, based on the user specified threshold N, clusters are reported – the log templates. We consider SLCT to be outdated, as it does not perform well neither in terms of accuracy nor scalability. For the former, its overall parsing accuracy is poor (having an average slightly below 70% over 9 datasets, for 2k logs). For the latter, although the authors of [7] claim the method to be scalable, our empirical results indicate otherwise (Section 3.2). More specifically, we found that SLCT is not robust to parsing different datasets of large sizes. Additionally, compared to other methods, its runtime is rather high (for example, parsing 20k logs for the Windows dataset requires ~25 minutes). More details can be found in Table 4.

AEL (Abstracting Execution Logs) [15] proposes a rule-based approach that consists of three steps. The first step requires the use of heuristics (rules tailored for the specifics of a particular log dataset) in order to replace the dynamic parts of a log message with generic tokens[4]. One of the two heurstics regards all "word=value" pairs present in a log message as containing dynamic information, thus replacing *value* with a generic token. For example, the log message: `"Data points amount to d=20"` gets transformed into `"Data points amount to d=$v"`. The second step clusters logs that are similar to each other in different groups (bins). For this step, the authors consider logs to be similar in terms of two aspects, namely the number of words and parameters (generic tokens) of a log line. The third step of the method iterates through all the previously created groups and returns the log templates. More specifically, by using a similarity metric, every log line within a specific group is compared against all the others. Logs that are similar to each other are considered to belong to the same template. Unfortunately, the authors do not specify how the similarity metric is computed. Finally, the log templates are returned. One of its biggest disadvantages is that, if the first step cannot be followed, the method cannot be used (as the similarity comparisons

[4]AEL replaces the dynamic parts of a log message with "$v".

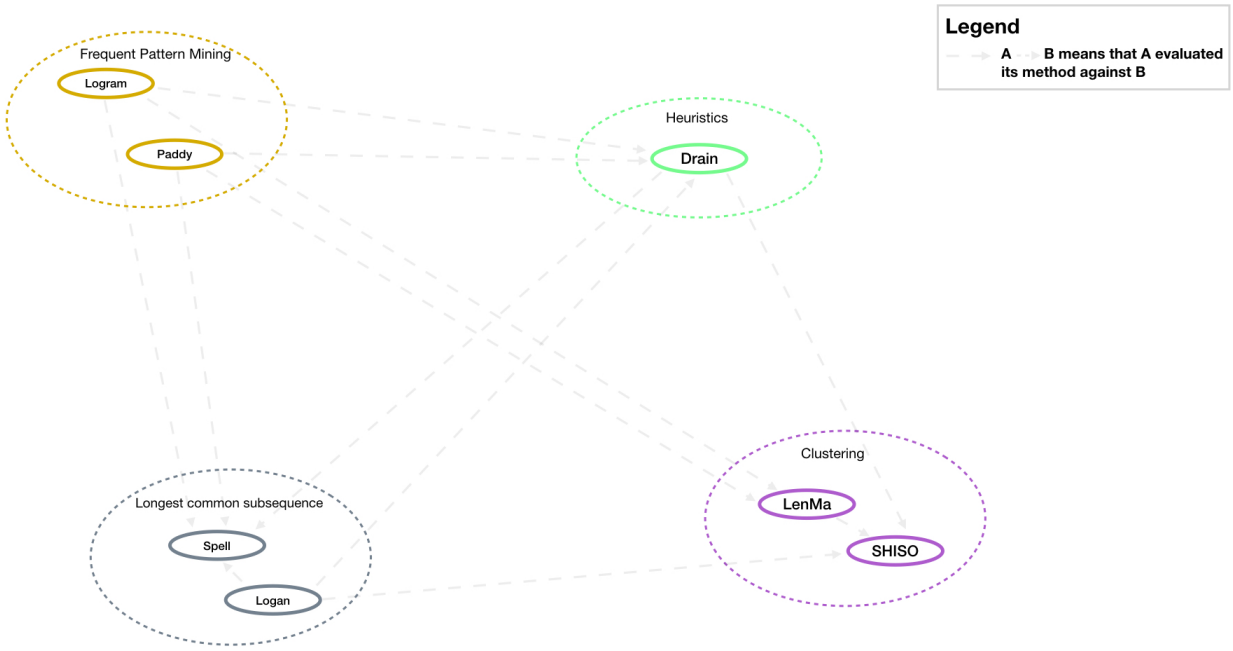

Figure 5. Graphical representation of how the online methods are clustered together based on their algorithmic approach. The arrows represent connections between various methods. More specifically, an arrow means that the method from which it originates, evaluated against the method towards which it points.

cannot be made anymore – the authors of LFA were unable to use the method due to being unable to follow AEL's first step). AEL's accuracy results looked very promising (having an average above 90% over 9 datasets, for 2k logs). However, it is still heavily dependent on heuristics, and might not be the best solution for log datasets of high diversity. In terms of scalability, compared to the other methods, AEL obtains good results – is able to parse 10M Thunderbird logs in ~80 minutes (Section 3.2).

HLAer (Heterogeneous Log Analyzer) [25] proposes a 3-step log parsing approach. During its first stage, logs are tokenized by using the white space delimiter. More specifically, all words and special symbols (except numbers) are separated by an empty space (other approaches consider existing whitespaces as already being the token delimiter). For example, the log message: `"GET /images/header/nav.gif"` gets transformed into `"GET / images / header / nav . gif"`. For the second step, the authors cluster logs using the *OPTICS*[5] [2] algorithm. During the third stage, for each of the previously found clusters, a pattern recognition algorithm is applied such that event templates are found and returned. Due to code unavailability, HLAer was not able to be reproduced.

IPLoM (Iterative Partitioning Log Mining) [18] tackles log parsing by proposing a novel 4-step hierachical partitioning algorithm. During the first stage, log lines are partitioned by their token[6] count. This step assumes that log lines that have the same template are more likely to have the same length in tokens. The second step involves partitioning by the token position. Here, the authors assume that, for a log that has a length of n tokens, the column with the least variability is most likely to contain static parts (parts of the template). The third step represents partitioning by "search for bijection" – a mapping between the set of unique tokens (suspected to be apart of the log template). Last but not least, during the fourth and final step, log templates are returned. In terms of accuracy, the results were not that promising – an average parsing accuracy of roughly 70% over 9 datasets (Section 3.3). In terms of scalability, we would recommend using IPLoM as it proved to be scalable – it is able to parse 10M logs in ~30 minutes (Section 3.2).

[5] Although the *OPTICS* algorithm has an $O(n^2)$ memory complexity [7], the authors HLAer still claim the method to be scalable as it can be parallelized.

[6] Any sequence of characters separated by whitespaces is considered to be a token – a word, a number, an IP address etc. For example, *"Connection from 120.0.0.1"* has 3 tokens.

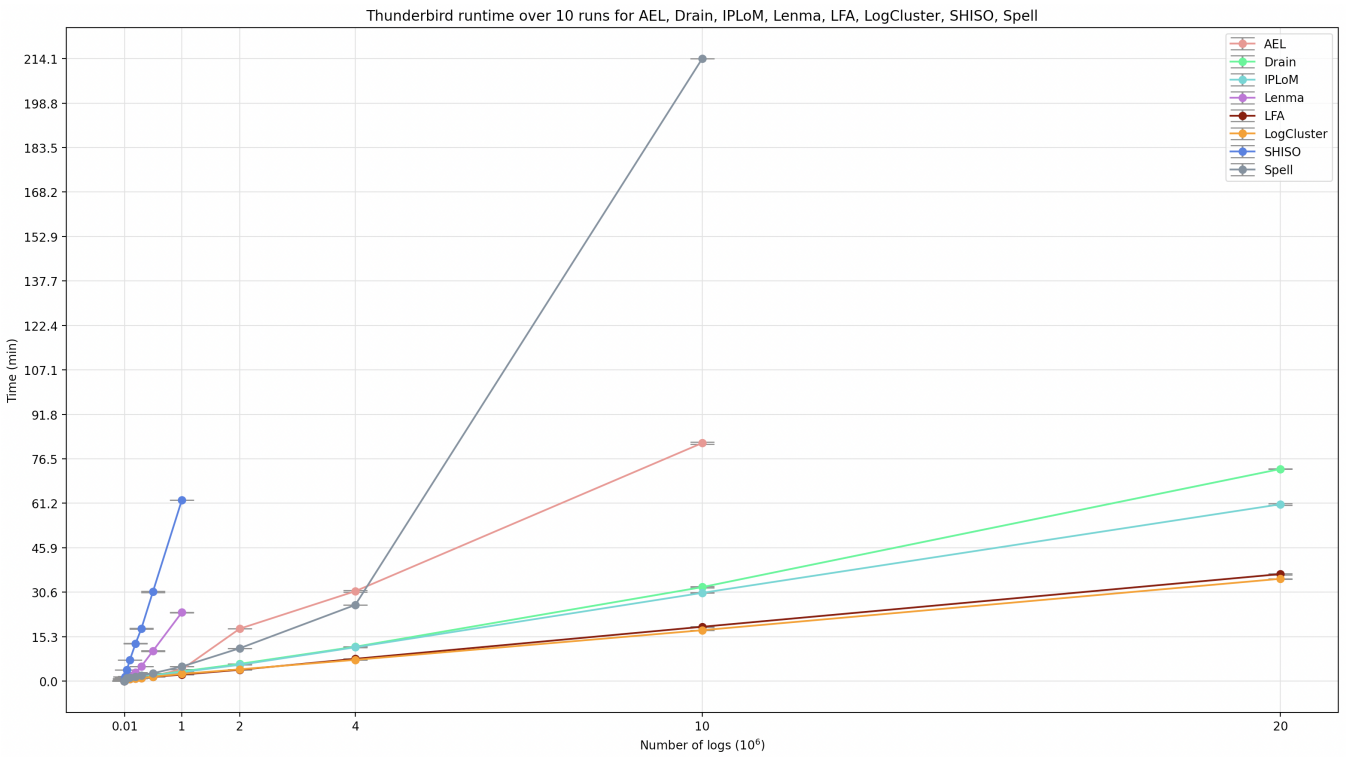

Figure 6. Runtime measurements using the Thunderbird dataset for Drain, Spell, LogMine, and SLCT.

LogCluster [35] tries to overcome the shortcomings of SLCT and IPLoM by introducing a frequent pattern mining algorithm, consisting of 3 steps[7]. During its first step, it discovers frequent words in the log dataset. During its second stage, it generates a set of candidate clusters. During its third step, it drops all cluster candidates that have the counter value lower that the defined support counter threshold, and subsequently reports the remaining candidates (log templates). An advantage of using this method is that it also considers log messages for which the parameter length is flexible (which proved to be a problem for SLCT and IPLoM). For example, "`user Fiona workerEnv in error state 7`" and "`user Peter Pan workerEnv in error state 9`" have the same event template "`user <*> workerEnv in error state <*>`", while the length of the parameter (i.e., user name) is flexible. An important observation is that if LogCluster is run with low support threshold values, the results can be similar to the ones returned by SLCT. In terms of accuracy, the results were not that promising – an average parsing accuracy of roughly 70% over 9 datasets (Section 3.3). As neither LogCluster nor SLCT were intended for log parsing, but rather for finding frequent patterns in logs, for better accuracy results, we would advise using other methods, like NuLog. In terms of scalability, we would recommend using LogCluster as it proved to be scalable – it is able to parse 20M logs in ~35 minutes (Section 3.2).

LogMine [9] introduces a new clustering approach for log parsing, consisting of 4 steps. The authors tackle the problem of log parsing by considering log messages as being automatically generated, rather than normal sentences (like the ones from a story book). The first step of the algorithm involves, tokenization and type detection[8] (for e.g., a type can be a date, timestamp, IP address, number etc.). As the second step, a clustering of the logs is done (by using a specific distance metric). During the third step, they run an algorithm to recognize patterns in logs (for each of the previously found clusters, a single pattern is representative of all other instances of a cluster) patterns. As their final step, the authors introduce an algorithm that generates a hierarchy of patterns, and returns log templates. In terms of accuracy, LogMine obtained roughly 75% over 9 datasets. However, the method was not able to parse the OpenSSH dataset (this dataset was not incuded in the experiments). Thus, we think there are some problems in terms of its robustness to parsing different datasets. Furthermore, the method showed poor scalability results, being unable to parse more than 20k logs (in ~10 minutes), regardless of the dataset used. Although

[7] In contrast to SLCT and IPLoM, LogCluster does not take the words' positions into account.

[8] During its first stage, LogMine replaces detected variables, such as numbers or dates with the name of the field. For example, after the first step, the log message "`session opened for user anna uid = 10`" becomes "`session opened for user name uid = number`"

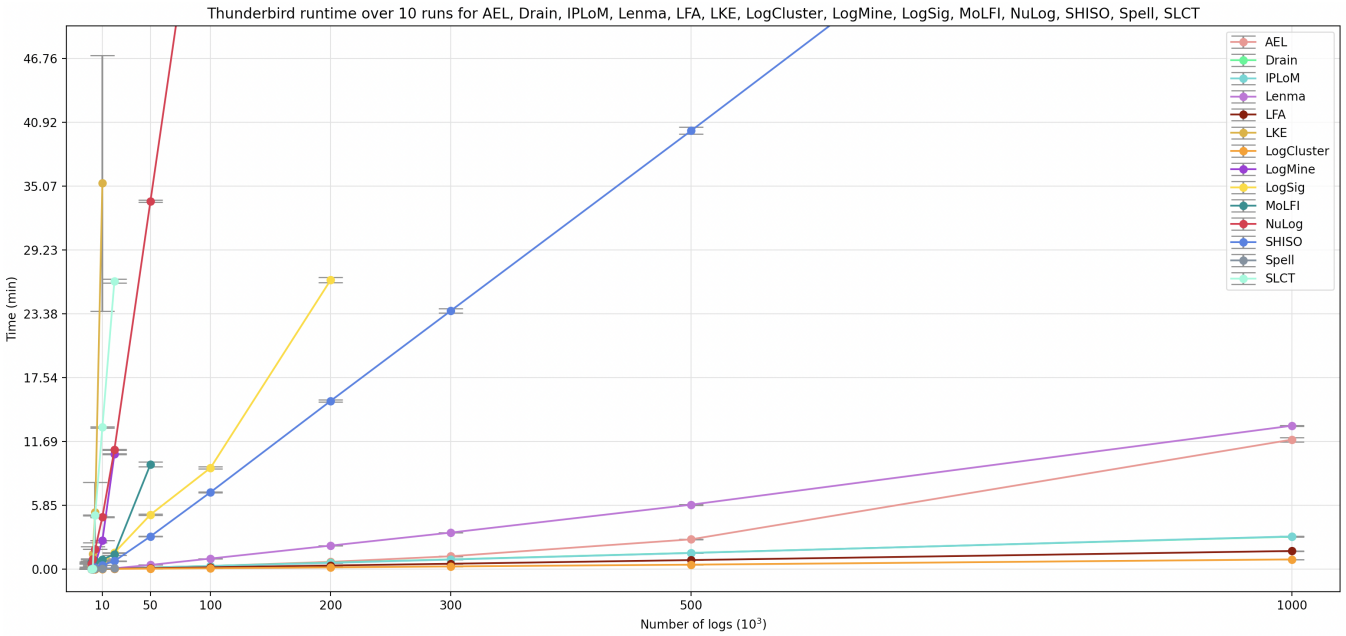

Figure 7. Scalability experiments up to 1M logs for all methods, namely Drain, IPLoM, Lenma, LogCluster, AEL, SHISO, LFA, Spell, SLCT, LogSig, NuLog, MoLFI, LogMine, LKE. Datasets used for each can be seen in Figure 8.

the method initially looked promising, its empirical results lead to a different conclusion.

NuLog (Neural Log) [24] formulates log parsing as a self-supervised[9] learning task. The method has two operation modes, namely offline and online. The former is used for training the model for log parsing, whereas the latter is used during the execution (i.e., when log templates are generated by forward passing log messages through the offline trained model). Training the model consists of 2 steps, namely tokenization and masking. For tokenization, each log message is transformed into a sequence of tokens, words being separated by using a specific delimiter (for e.g. whitespace). During masking, the authors use a general method from natural language processing called *Masked Language Modelling*. In practice, for each sequence of tokens (log message), a random token is replaced by the special `<MASK>` token. Furthermore, each token sequence is padded with two delimiter tokens, namely `<CLS>` and `<SPEC>`. Finally, after training, the model is able to return log templates. In terms of scalability, the method seems to scale poorly (~35 minutes for a 50k log dataset). However, it has proved to surpass all other methods in terms of parsing accuracy results (Section 3.3). Furthermore, it has proven to be robust to all the datasets – lowest accuracy being still above 80%. We highly recommend NuLog if the use case requires high parsing accuracy.

ELA (Event Log Abstraction) [30] uses a 5-step algorithm to parse logs. Compared to other methods such as Drain or AEL which rely heavily on heuristics, it is capable of parsing logs without any user input/hard-coded rules whatsoever. The first stage consists of an automatic preprocessing operation, namely running *nerlogparser* [29] and identifying all the unique log messages. More specifically, the first preprocessing procedure means splitting and labelling each field for each log entry (*nerlogparser*), whereas the second simply means extracting all the different (now-)preprocessed messages[10]. During its second stage, the authors group logs based on the word count – they assume that logs that contain the same number of words are likely to belong to the same log template. As the third step, the authors construct a graph model using the previously found count-based word groups, and, during its fourth stage, the authors cluster the log entries using an automatic approach (still, no user parameters are required). Last but not least, during the fifth and final stage, log templates are returned. The biggest advantage of ELA is that it does not require any hyperparameters, nor any knowledge about log datasets particularities.

AWSOM-LP [27] proposes a novel approach that tackles log parsing by using a frequency analysis technique. More specifically, it is composed of 3 steps. During its first stage, specific pre-processing is applied by using regular expressions (user input). The second step implies grouping (clustering) log messages based on a string similarity metric. During

[9]Self-supervised learning can be regarded as the intermediate between supervised and unsupervised machine learning [37]. More specifically, in supervised-learning, machine learning models make use of unlabeled data to yield labels.

[10]For ELA, during its first step, unique messages refer to messages that differ from all other previously parsed log lines.

the third and final stage, frequency analysis is applied in order to distinguish between constants and variables. In practice, this means counting the number of occurrences of each term for all log messages (that belong to a previously found cluster). Additionally, a post-processing operation is conducted (all numbers that are still present are considered to be variables). Finally, the log templates are returned.

LogStamp [32] introduced a novel approach that tackles log parsing as a sequence labelling task. More specifically, they train a model able to classify the tokens of a log message as either being constant or variables. This is achieved by training a classifier that serves as a tagger. The training data is obtained automatically, by using two different processes (both are aided by BERT [5], which is used for feature representations of log messages). The first process is responsible for embedding log messages at a coarse level[11], and then, by means of clustering, obtaining pseudo-labels for the input data. The second process, embeds log messages at a fine-grained level, which are then passed as input to the classifier. Thus, using the coarse and fine-grained level representations, a classifier capable of finding log templates is trained. Finally, the classifier is used during the online workflow (during execution), to predict log templates.

Table 2. Overview of all online log parsing approaches found in the literature.

Year	Paper
2013	SHISO [21]
2016	LenMa [28]
2017	Drain [11]
2019	Spell [6]
2019	Logan [1]
2020	Logram [4]
2020	Paddy [14]
2021	Prefix-Graph [3]

3.1.2 Online log parsing approaches. Figure 5 contains a clustering of the online methods based on their algorithmic approach. In comparison to the offline methods, as the number of online methods was smaller, naturally the number of formed clusters (& respective members) is smaller. Thus, it is premature to draw any conclusions in terms of which algorithmic approach is most preferred within the community. The connection between two methods represents the same concept as in Figure 4.

LenMa [28] proposes a 5-step clustering algorithm that considers the length of each word in a log message as a similarity metric between log messages. It first creates a word length vector, and a word vector of the message. Secondly, it calculates a similarity metric between the incoming messages and the clusters that have the same number of words. Based on a threshold value T_c, if an incoming log message is not similar enough, a new cluster gets created and returned. Last but not least, the most similar cluster is updated with the newly arrived message, and gets returned. In terms of scalability, the method can parse 1M logs in roughly ~25 minutes. In terms of accuracy, for some datasets it achieves good results, and for some others the results are not that great. An overview of this can be found in Table 3.

Drain [11] proposed a 5-step approach that relies heavily on heuristics. It can achieve high parsing accuracies for various datasets and it is recommended for use-cases where log data's diversity is rather low, as the approach is very sensitive to structural changes in log data (as mentioned previously, it is designed as a rule-based approach). During its first stage, logs are preprocessed by using domain knowledge – here, users have to provide regular expressions. During the second, third, and fourth step, the authors construct a 'parse-tree' – a tree structure that allows logs to be parsed using a number of heuristics (for e.g., log messages with the same number of tokens are more likely to have the same log template, the first token of a log message is always constant and apart of the log template, etc.). Last but not least, logs' tokens are compared using a similarity metric. Finally, during the final stage, the log template is returned and the parse-tree is updated. In terms of scalability and accuracy, we can say that Drain obtains promising results for both. For the former, Drain is able to parse 20M logs in ~25 minutes using ~25 GiB of memory (Section 3.2). For the latter, it is able to achieve an average accuracy of 95% over 9 different datasets (Section 3.3). We recommend Drain for use cases where log data is unlikely to change its intrinsic structure, as it is both scalable and accurate. However, if that is not the case, we do not recommend Drain, as constantly updating regular expressions can be a tedious and erroneous process.

Paddy (Parsing Approach with Dynamic Dictionary) [14] is a 4-step log parsing algorithm. During the first stage, log data is preprocessed using domain knowledge by means of regular expressions (provided by users). During the second stage, by means of using an inverted index (implemented using a dictionary structure), log template candidates are retrieved. During the third stage, the previous candidates are ranked using a similarity metric. More specifically, the

[11] Representing logs at a coarse level means extracting features that encompass information about the entire log message, rather than choosing a representation with too many details. Here, the coarse representation means extracting features that allow for distinguishing between logs that are very different from each other. For example, a coarse level representation would mean being able to tell that `"authentication failure; logname= uid=0"` is different from `"check pass; user unknown"`. However, a coarse level representation is not able to distinguish between messages that are similar, but apart of a different underlying log template.

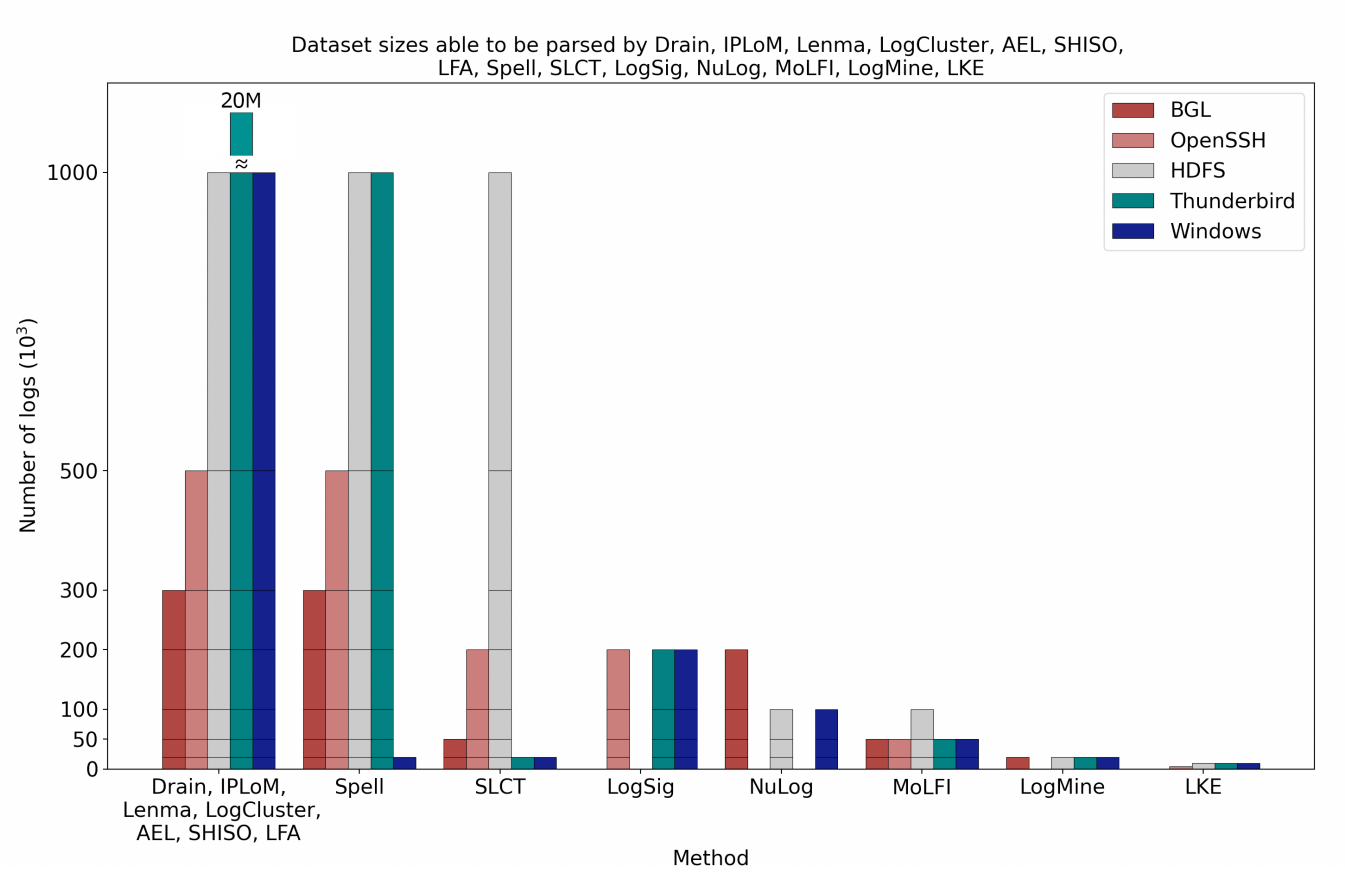

Figure 8. Dataset sizes used for the scalability experiments for Drain, IPLoM, Lenma, LogCluster, AEL, SHISO, LFA, Spell, SLCT, LogSig, NuLog, MoLFI, LogMine, LKE. If a method parsed a dataset of size *N*, all smaller splits were also parsed (for example if Drain parsed the 1M Windows dataset, it also parsed 1k, 2k, 4k, 10k, 20k, 50k, 100k, 200k, 300k, 500k log lines during the experiments)

Table 3. Parsing accuracy results after running each method 10 times (for each dataset). For the experiments, datasets comprised of 2k logs were chosen. The results are averaged over 10 runs.

Dataset	AEL	Drain	IPLoM	Lenma	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
BGL	0.957	0.962	0.939	0.689	0.854	0.127	0.835	0.723	0.226	0.948	**0.980**	0.711	0.572	0.786
OpenStack	0.757	0.732	0.341	0.742	0.200	0.787	0.695	0.743	0.866	0.213	**0.990**	0.721	0.867	0.764
HDFS	0.997	0.997	**1.000**	0.997	0.885	**1.000**	0.546	0.850	0.849	0.997	0.998	0.997	0.545	**1.000**
Apache	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.708	**1.000**	0.582	**1.000**	**1.000**	**1.000**	0.730	**1.000**
HPC	0.903	0.887	0.829	0.829	0.816	0.845	0.787	0.784	0.354	0.824	**0.945**	0.324	0.838	0.654
Windows	0.689	0.997	0.566	0.565	0.588	0.989	0.713	0.992	0.689	0.679	**0.998**	0.700	0.696	0.988
HealthApp	0.567	0.780	0.821	0.174	0.548	0.474	0.530	0.684	0.235	0.382	**0.875**	0.397	0.331	0.639
Mac	0.763	0.786	0.000	0.698	0.599	0.221	0.603	0.872	0.477	0.650	**0.821**	0.595	0.557	0.756
Spark	0.905	0.920	0.920	0.883	0.993	0.380	0.798	0.575	0.543	0.650	**1.000**	0.906	0.685	0.905

similarity metric is a weighted sum (coefficients as hyper-parameters) between *Similarity* (Jaccard similarity [26]) and *LengthFeature* (the length of a log message in terms of number of tokens). During the fourth and final step, the log template is returned and the inverted index is updated.

3.2 Scalability of log parsing approaches

In order to test scalability, we measured the runtime and the memory consumption of each of the methods. Using the source code provided by the authors of [13], we modified

the benchmarks' code in order to account for both metrics[12]. We have to mention that no algorithmic changes were done, but rather changes that were related to collecting the experiments' results automatically. In order to reduce the measurements noise, we ran each of the experiments 10 times, for each dataset, and for its respective size (Table 4). Last but not least, experiments were run on a dual socket AMD Epyc2 machine with 64 cores in total (with a dual Nvidia RTX 2080Ti graphics card). In the following subsections, we discuss how we constructed the datasets used for the experiments, and discuss the empirical results obtained.

3.2.1 Constructing the datasets. Using the log data provided by the authors of [13], we created 6 different datasets. We selected datasets that had at least 300k log lines, namely Android, BGL, OpenSSH, HDFS, Thunderbird, and Windows. The way in which we constructed the datasets was to select the first k lines from the original data (for example, if a dataset had 330k log lines in total, we selected only the first 300k log lines; if a dataset had 2.5 million log lines in total, we selected only the first million log lines). In order to see how the methods scale, the datasets were split in a number of different sizes, namely 1k, 2k, 4k, 10k, 20k, 50k, 100k, 200k, 300k, 500k, 1M log lines. An additional dataset comprised of 20M logs was created (Thunderbrid). A visualisation of the constructed datasets sizes can be found in Figure 8.

3.2.2 Scalability experiments. We tested the scalability of the methods from an empirical perspective. Thus, we defined the scalability of a method by taking into account two aspects, namely the ability to parse various datasets and dataset sizes, and the runtime and memory consumption measurements. Unfortunately, not all methods (found in the literature) were able to be tested (either due to the lack of source code, or insufficient technical implementation). However, the ones that were able to be tested can be found in Table 4.

In [7], the authors claim that SLCT, POP, LogMine, Spell, Drain are scalable. Also, they claim that IPLoM and HLAer are potentially scalable. Thus, we were particularly interested in obtaining the results for these. However, the code for POP and HLAer was not available. Consequently, we were unable to run any experiments for these two. We have to mention that most of the algorithms had problems parsing the Android dataset, which in turn made comparisons between methods impossible for this specific dataset. On the other hand, there were no issues running the experiments for the rest of the datasets (and comparisons were thus feasible).

Figure 7 shows a visualisation of the runtime measurements of the different methods that were able to be tested (up to 1M log lines). We observe a significant difference between LogCluster, LFA, IPloM, Lenma and the rest of the methods. A visualisation of the memory consumption for these 4 methods can be seen in Figure **??**. We observe that the memory difference is not that significant, compared to the previous (runtime) scenario. Last but not least, the figure provides an estimate of runtime, that can aid practitioners in choosing the appropriate method.

Figure 6 shows a visualisation of the runtime measurements of the different methods that were able to be tested (up to 20M log lines). We observe that only 4 methods are capable of parsing 20M logs, namely LogCluster, LFA, IPLoM and Drain. We also observe a significant difference between these 4 methods and the others. Up to 4M, it could have been considered that Spell scales similarly to AEL, however the results indicate otherwise (significant difference between runtime from 4M to 10M). Last but not least, the figure provides an estimate of runtime, that can aid practitioners in choosing the appropriate method. Additional visualisations can be found in Appendix B.

3.3 Accuracy of log parsing approaches

In order to test the methods' accuracy, we reproduced the results[13] of the experiments performed in [23]. Overall, the methods have different parsing accuracies for different datasets. A visualisation of this can be seen in Figure **??**. For a list of all results, please consult Table 3. Since the determination of the accuracy requires the labels to be already present in order to compare against a ground truth, this part of our study is limited to a 2k dataset. While we do not consider this to impact the generality of our finding significantly[14], we hope to produce larger labeled datasets in future work to expand this part of our study.

4 Discussion & Conclusion

In this paper we analyed the SOTA log parsing methods. The aim of this survey was directed towards three aspects, namely mapping out SOTA log parsing approaches, evaluating the scalability and the accuracy of these methods. Regarding the first aspect, log parsing approaches are divided in two main categories, offline and online approaches. For offline log parsing approaches, 14 were successfully identified, whereas, for online log parsing approaches, 20 approaches were successfully identified. The answer to the second research question is that it seems like there are only a few scalable methods (out of the ones that were available to be tested) namely, Drain, IPLoM, LFA, LogCluster (these can run their methods in a matter of minutes, and seem robust to

[12]Code available at: https://github.com/spetrescu/literature-survey-log-parsing

[13]These can be found at Section 2.2

[14]For example, for datasets sizes 25 times greater than 2k log datasets, the runtime is ~5 minutes, which is still a reasonable amount of time in order to obtain other parsing accuracy estimates.

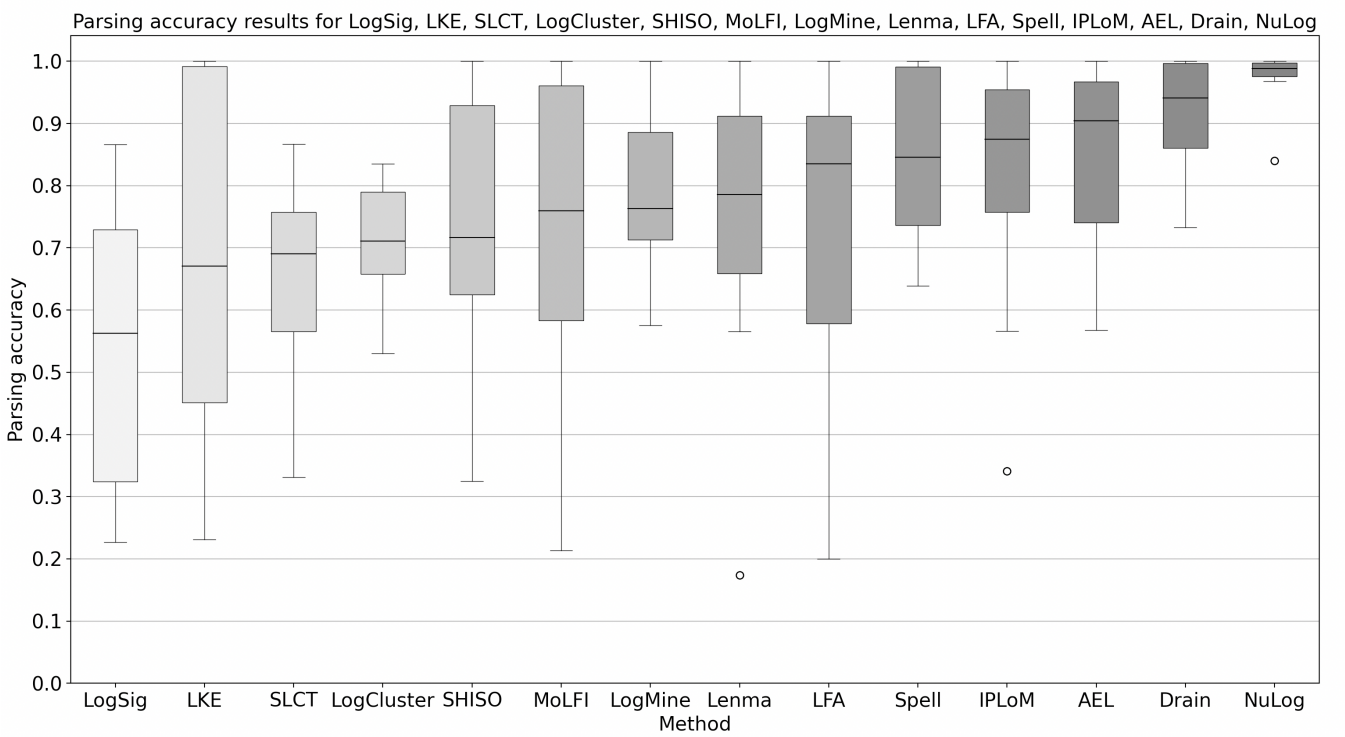

Figure 9. Parsing accuracy results for LogSig, LKE, SLCT, LogCluster, SHISO, MoLFI, LogMine, Lenma, LFA, Spell, IPLoM, AEL, Drain, NuLog.

various log formats). However, out of the 4, IPLoM consumes significantly more memory than the rest. In regards to the third reasearch question, the methods have various accuracy, and they mainly depend on the specifics of a dataset. However, NuLog proved to be robust to datasets' particularities, and obtained the best parsing accuracy results (over 95% over 9 datasets).

References

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

[2] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/304182.304187

[3] Guojun Chu, Jingyu Wang, Qi Qi, Haifeng Sun, Shimin Tao, and Jianxin Liao. 2021. Prefix-Graph: A Versatile Log Parsing Approach Merging Prefix Tree with Probabilistic Graph. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 2411–2422. https://doi.org/10.1109/ICDE51399.2021.00274

[4] Hetong Dai, Heng Li, Weiyi Shang, Tse-Hsun Chen, and Che-Shao Chen. 2020. Logram: Efficient Log Parsing Using n-Gram Dictionaries. arXiv:2001.03038 [cs.SE]

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[6] Min Du and Feifei Li. 2019. Spell: Online Streaming Parsing of Large Unstructured System Logs. *IEEE Transactions on Knowledge and Data Engineering* 31, 11 (2019), 2213–2227. https://doi.org/10.1109/TKDE.2018.2875442

[7] Diana El-Masri, Fabio Petrillo, Yann-Gaël Guéhéneuc, Abdelwahab Hamou-Lhadj, and Anas Bouziane. 2020. A systematic literature review on automated log abstraction techniques. *Information and Software Technology* 122 (2020), 106276. https://doi.org/10.1016/j.infsof.2020.106276

[8] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *2009 Ninth IEEE International Conference on Data Mining*. 149–158. https://doi.org/10.1109/ICDM.2009.60

[9] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: Fast Pattern Recognition for Log Analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (Indianapolis, Indiana, USA) *(CIKM '16)*. Association for Computing Machinery, New York, NY, USA, 1573–1582. https://doi.org/10.1145/2983323.2983358

[10] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. 2018. Towards Automated Log Parsing for Large-Scale Log Data Analysis. *IEEE Transactions on Dependable and Secure Computing* 15, 6 (2018), 931–944. https://doi.org/10.1109/TDSC.2017.2762673

[11] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *2017 IEEE International Conference on Web Services (ICWS)*. 33–40. https://doi.org/10.1109/ICWS.2017.13

[12] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. 2021. A Survey on Automated Log Analysis for Reliability Engineering. *ACM Comput. Surv.* 54, 6, Article 130 (July 2021), 37 pages. https://doi.org/10.1145/3460345

[13] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2020. Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics. arXiv:2008.06448 [cs.SE]

[14] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. 2020. Paddy: An Event Log Parsing Approach using Dynamic Dictionary. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 1–8. https://doi.org/10.1109/NOMS47738.2020.9110435

[15] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper). In *2008 The Eighth International Conference on Quality Software*. 181–186. https://doi.org/10.1109/QSIC.2008.50

[16] Staffs Keele et al. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Citeseer.

[17] Satoru Kobayashi, Kensuke Fukuda, and Hiroshi Esaki. 2014. Towards an NLP-Based Log Template Generation Algorithm for System Log Analysis. In *Proceedings of The Ninth International Conference on Future Internet Technologies* (Tokyo, Japan) *(CFI '14)*. Association for Computing Machinery, New York, NY, USA, Article 11, 4 pages. https://doi.org/10.1145/2619287.2619290

[18] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. 2012. A Lightweight Algorithm for Message Type Extraction in System Application Logs. *IEEE Transactions on Knowledge and Data Engineering* 24, 11 (2012), 1921–1936. https://doi.org/10.1109/TKDE.2011.138

[19] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A Search-Based Approach for Accurate Identification of Log Message Formats. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 167–177. https://doi.org/10.1145/3196321.3196340

[20] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. 2013. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255. https://doi.org/10.1109/TPDS.2013.21

[21] Masayoshi Mizutani. 2013. Incremental Mining of System Log Format. In *2013 IEEE International Conference on Services Computing*. 595–602. https://doi.org/10.1109/SCC.2013.73

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

[23] Sasho Nedelkoski. 2021. Deep anomaly detection in distributed software systems. (2021).

[24] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2020. Self-Supervised Log Parsing. *CoRR* abs/2003.07905 (2020). arXiv:2003.07905 https://arxiv.org/abs/2003.07905

[25] Xia Ning, Geoff Jiang, Haifeng Chen, and Kenji Yoshihira. 2013. HLAer : a System for Heterogeneous Log Analysis.

[26] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. [n. d.]. Using of Jaccard Coefficient for Keywords Similarity.

[27] Issam Sedki, Abdelwahab Hamou-Lhadj, and Otmane Aït Mohamed. 2021. AWSOM-LP: An Effective Log Parsing Technique Using Pattern Recognition and Frequency Analysis. *CoRR* abs/2110.15473 (2021). arXiv:2110.15473 https://arxiv.org/abs/2110.15473

[28] Keiichi Shima. 2016. Length Matters: Clustering System Log Messages using Length of Words. arXiv:1611.03213 [cs.OH]

[29] H. Studiawan, F. Sohel, and C. Payne. 2018. Automatic log parser to support forensic analysis. In *16th Australian Digital Forensics Conference*. https://researchrepository.murdoch.edu.au/id/eprint/42841/

[30] Hudan Studiawan, Ferdous Sohel, and Christian Payne. 2020. Automatic Event Log Abstraction to Support Forensic Investigation. In *Proceedings of the Australasian Computer Science Week Multiconference* (Melbourne, VIC, Australia) *(ACSW '20)*. Association for Computing Machinery, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/3373017.3373018

[31] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating System Events from Raw Textual Logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (Glasgow, Scotland, UK) *(CIKM '11)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2063576.2063690

[32] Shimin Tao, Weibin Meng, Yimeng Chen, Yichen Zhu, Ying Liu, Chunning Du, Tao Han, Yongpeng Zhao, Xiangguang Wang, and Hao Yang. 2021. LogStamp: Automatic Online Log Parsing Based on Sequence Labelling. *Interface* 19, 03 (2021), 03.

[33] Stefan Thaler, Vlado Menkonvski, and Milan Petkovic. 2017. Towards a neural language model for signature extraction from forensic logs. In *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*. 1–6. https://doi.org/10.1109/ISDFS.2017.7916497

[34] R. Vaarandi. 2003. 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)*. 119–126. https://doi.org/10.1109/IPOM.2003.1251233

[35] Risto Vaarandi and Mauno Pihelgas. 2015. LogCluster - A data clustering and pattern mining algorithm for event logs. In *2015 11th International Conference on Network and Service Management (CNSM)*. 1–7. https://doi.org/10.1109/CNSM.2015.7367331

[36] Wikipedia contributors. 2021. Logging (software) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Logging_(software)&oldid=1042915159 [Online; accessed 22-October-2021].

[37] Wikipedia contributors. 2021. Self-supervised learning — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Self-supervised_learning&oldid=1054652741 [Online; accessed 19-January-2022].

[38] Tong Xiao, Zhe Quan, Zhi-Jie Wang, Kaiqi Zhao, and Xiangke Liao. 2020. LPV: A Log Parser Based on Vectorization for Offline and Online Log Parsing. In *2020 IEEE International Conference on Data Mining (ICDM)*. 1346–1351. https://doi.org/10.1109/ICDM50108.2020.00175

[39] Lin Zhang, Xueshuo Xie, Kunpeng Xie, Zhi Wang, Ye Lu, and Yujun Zhang. 2019. An Efficient Log Parsing Algorithm Based on Heuristic Rules. In *Advanced Parallel Processing Technologies*, Pen-Chung Yew, Per Stenström, Junjie Wu, Xiaoli Gong, and Tao Li (Eds.). Springer International Publishing, Cham, 123–134.

[40] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2019. Tools and Benchmarks for Automated Log Parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 121–130. https://doi.org/10.1109/ICSE-SEIP.2019.00021

A Queries

In this section you can find the papers that were selected for the survey. Each subsection contains the number of papers selected for the respective query.

A.1 Google Scholar

These were the queries used in Google Scholar.

Query 1: "log parsing"

From the first 100 results, 27 were selected. 29 additional papers were selected during the process of backward snowballing.

Query 2: "log parsing survey"

From the first 100 results, 2 were selected. 2 additional papers were selected during the process of backward snowballing.

Query 3: "log abstraction"

From the first 100 results, 7 were selected. 2 additional papers were selected during the process of backward snowballing.

Query 4: "log abstraction survey"

From the first 100 results, 0 were selected.

Query 5: "event log parsing"

From the first 100 results, 6 were selected. 1 additional paper was selected during the process of backward snowballing.

Query 6: "log signature extraction"

From the first 100 results, 6 were selected. 0 additional papers were selected during the process of backward snowballing.

Query 7: "event log signature extraction"

From the first 100 results, 0 were selected.

A.2 Scopus

This query was used in Scopus.

Query 1: "TITLE-ABS-KEY(log AND parsing) OR ((logs OR log OR logging OR events OR "event log" OR "event logs" OR "event logs templates" OR "event log signatures") AND (abstractionOR parsing))"

From all 392 results, 13 were selected. 0 additional papers were selected during the process of backward snowballing.

B Experiments visualisations

This section contains visualisations of runtime and memory consumption measurements for IPLoM, AEL, Lenma, MoLFI, SHISO, LogCluster, LogSig, and LKE.

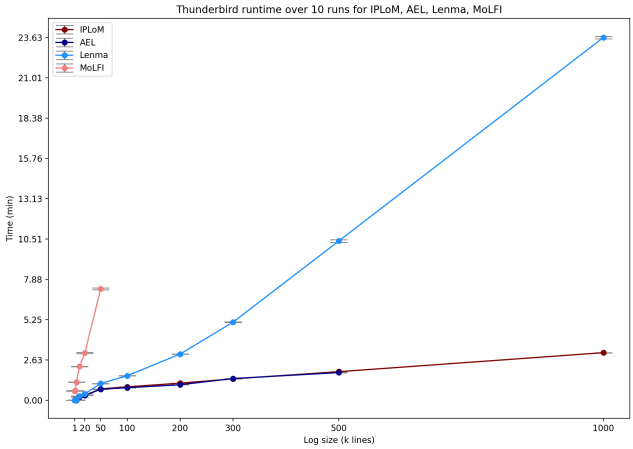

Figure 10. Runtime measurements using the Windows dataset for IPLoM, AEL, Lenma, and MoLFI.

In Figure 10 we can observe that there is a significant difference between IPLoM & AEL and Lenma. Compared to the first two that parse the dataset in about 1.5 minutes (for 500k log lines), Lenma parses 500k in 10 minutes. Also, in Figure 11 we notice a significant difference between the aforementioned methods and MoLFI. The first three methods parse 50k logs in roughly 1 minute, compared to MoLFI which takes 8.

We plot the same measurements for the Windows dataset.

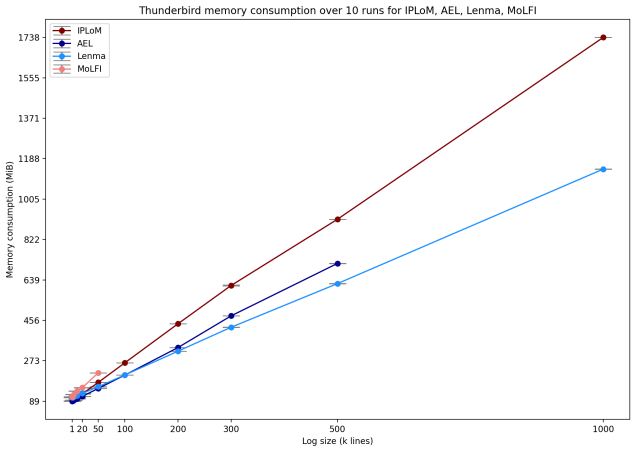

Figure 11. Memory consumption measurements using the Thunderbird dataset for IPLoM, AEL, Lenma, and MoLFI.

These can be found in Figure 12 and Figure 13 respectively.

It can be see that the methods perform similarly to the Thunderbird dataset. However, we notice some differences. For the Windows dataset, the runtime of Lenma reduces significantly compared to the previous scenario. Also. MoLFI runtime seems to increase, compared to the experiments performed on Thunderbird.

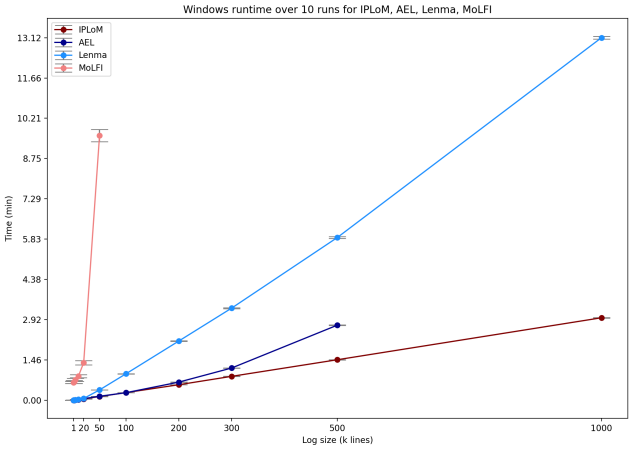

Figure 12. Runtime measurements using the Windows dataset for IPLoM, AEL, Lenma, and MoLFI.

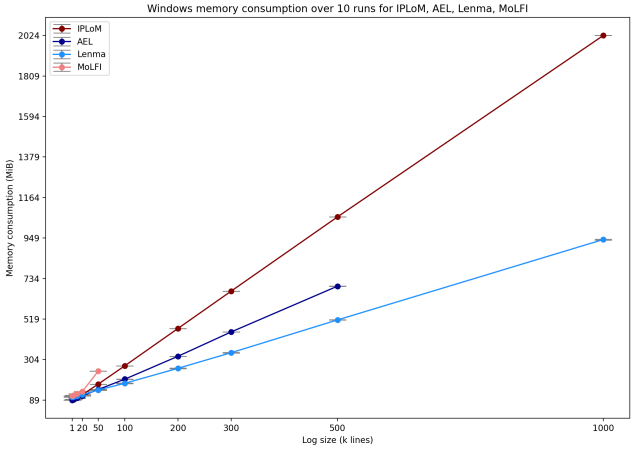

Figure 13. Memory consumption measurements using the Windows dataset for IPLoM, AEL, Lenma, and MoLFI.

Table 4. Scalability experiments successfully run for each dataset and method.

Dataset	AEL	Drain	IPLoM	Lenma	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
BGL 1k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 2k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 4k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 10k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 20k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
BGL 50k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓
BGL 100k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓
BGL 200k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓
BGL 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
HDFS 1k	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 2k	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 4k	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 10k	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 20k	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓
HDFS 50k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓
HDFS 100k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓
HDFS 200k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
HDFS 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
HDFS 500k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
HDFS 1M	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
OpenSSH 1k	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 2k	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 4k	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 10k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 20k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 50k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✓	✓
OpenSSH 100k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✓	✓
OpenSSH 200k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✓	✓
OpenSSH 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
OpenSSH 500k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
Thunderbird 1k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 2k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 4k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 10k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 20k	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓
Thunderbird 50k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓
Thunderbird 100k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✗	✓
Thunderbird 200k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✗	✓
Thunderbird 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
Thunderbird 500k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
Thunderbird 1M	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓
Windows 1k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows 2k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows 4k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows 10k	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows 20k	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
Windows 50k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✗	✗
Windows 100k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✓	✗	✗
Windows 200k	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✗	✗
Windows 300k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗
Windows 500k	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗
Windows 1M	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗

Table 4. Scalability experiments successfully run for each dataset and method. (Continued)

Dataset	AEL	Drain	IPLoM	Lenma	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
Thunderbird 2M	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓
Thunderbird 4M	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓
Thunderbird 20M	✗	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
Thunderbird 50M	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Thunderbird 100M	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Table 5. Parsing accuracy experiments able to be run on various datasets by the selected methods.

Dataset	AEL	Drain	IPLoM	Lenma	LFA	LKE	LogCluster	LogMine	LogSig	MoLFI	NuLog	SHISO	SLCT	Spell
BGL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
OpenStack	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HDFS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Apache	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HPC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windows	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HealthApp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Mac	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Spark	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓