International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-10 Issue-5, June 2021

Survey on Online Log Parsers

Tejaswini S, Azra Nasreen

***Abstract** Technological dependence is growing in leaps and bounds as days progress. As a result, software applications are required to be up and running at all times without fail. The health and safety of these applications need to be monitored regularly by theuse of constant logging of any faults that occur at their runtime executions. Log analysis techniques are applied to recorded logs to obtain a better overview of how to handle failures and health deterioration. Before these algorithms can be utilized in practice, the raw unstructured logs need to be converted into structured log events. This process is performed by log parsers, which are accessible in two different modes – offline and online. While offline log parsers have a pre-defined knowledge base containing templates and conversion rules, online log parsers learn new templates on the job. This paper focuses on surveying and creating a comparative study on online log parses by analysing the type of technique used, efficiency and accuracy of the parser on a given dataset, time complexity, and their effectiveness in motivating applications.*

***Keywords**: Log analysis techniques are applied to recorded logs to obtain a better overview of how to handle failures and health deterioration.*

I. INTRODUCTION

In a computing context, logs can be defined as pieces of information that provide insight into various events that occur throughout the run-time of a computer application. Real-time systems generate log files that are massive in size that find use in many fields such as security[37],[24], health monitoring[15],[13], failure handling[31],[25] and anomaly detection[22],[30]. There are many open source and freely available frameworks that ease the process of logging in applications. For example, java applications use a logging fac¸ade like SLF4J, which bind with frameworks like Logback, Log4j and so on. Figure 1 shows an example how logging works using SLF4J. Figure 2 shows the entries logged in the file corresponding to the logging statements.

```
public void example()
{
   //statements
   logger.info("This is an info message");
   //statements
   logger.debug("This is a debug message");
}
```

Fig. 1. Log statements using slf4j facade

Manuscript received on June 12, 2021.
Revised Manuscript received on June 16, 2021.
Manuscript published on June 30, 2021.
* Correspondence Author

Tejaswini S*, Department of Computer Science and Engineering, Rashtreeya Vidyalaya College of Engineering, Bangalore (Karnataka), India.

Azra Nasreen, Assistant Professor, Department of Computer Science and Engineering, Rashtreeya Vidyalaya College of Engineering, Bangalore (Karnataka), India.

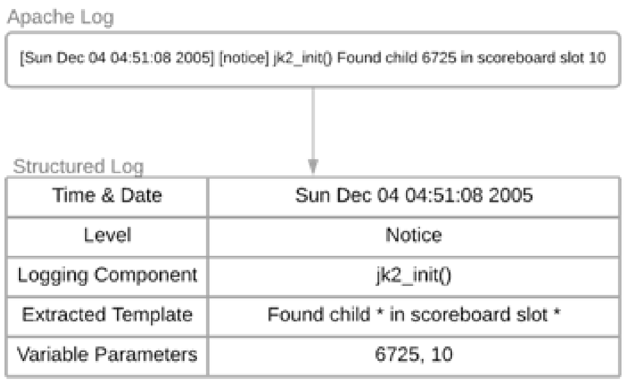
While logging of warnings, errors, debugs, information and traces may seem like a trivial task, there are many surveys[17],[18], that stress the importance of following logging practices. Three major aspects – how-to-log, where-to-log, and what-to-log are addressed to ensure that logging is not dependent purely on human or developer expertise[26]. How-to-log outlines "anti-patterns" that are undesirable designs to follow while adding logging statements in the source code[43]. Some papers [27],[26] also emphasize on the characterization and prioritization of the maintenance of logging statements. The position of logging statements in the feature code determines the where-to-log aspect[17]. Logging unnecessary information

```
1756 [main] INFO com.example.Logging - This is an info message
1825 [main] DEBUG com.example.Logging - This is an debug message
```

Fig. 2. Log file entries

can degrade performance and add on as avoidable overhead costs. Once the decision to log is made, the contents of the log
– message, parameter, thread information, level, and so on – need to be chosen, which constitutes what-to-log[39]. Both too much and too little information as well as the structure of the log have impact on how it is perceived by the user or othermining applications in later stages.

The generated log files are raw and unformatted that need to be converted to a suitable form for any kind of analysis to be performed. The process of reshaping unstructured entries into structured logs as shown in Figure 3 is called log parsing. Figure 3 uses an Apache log entry taken from the Loghub dataset[44].



[Sun Dec 04 04:51:08 2005] [notice] jk2_init() Found child 6725 in scoreboard slot 10

Time & Date	Sun Dec 04 04:51:08 2005
Level	Notice
Logging Component	jk2_init()
Extracted Template	Found child * in scoreboard slot *
Variable Parameters	6725, 10

Fig. 3. Example of a parsed Apache Log

A common brute-force way of performing log parsing is to use regular expressions[11] to match the template of the log. The drawback of using regular expressions is that it needs prior knowledge of all possible structures and cannot learn new templates without explicit developer intervention.

Another approach is to use static analysis[6],[5], which extracts the structure directly from the source code. More recent studies have shown promise in using automated log parsers. Some techniques employed are frequent pattern mining[1],[7], heuristics[29],[33], iterative partitioning[4],[30], clustering[3],[10],[20] and longest common subsequence[19],[32]. The obvious advantage of using automated log parsers like Drain[29],[33], Spell[19],[32], LenMa[23], IPLoM[4] and LogSig[10], is that they have the ability to learn templates for logs without any intervention. Automated log parsers are currently available to operate in two modes, namely online and offline. Offline log parsers like LogSig[10], IPLoM[4], LogCluster[16] and MoLFI[35], require all logs beforehand and process log entries in batches, whereas online log parsers like Spell[19],[32], Drain[29],[33], SHISO[12] and LenMa[23], can process logs in a streaming manner one after the other. This paper surveys existing implementations of online log parsers, by taking a deep dive into features like technique employed, efficiency and accuracy over a given open source dataset, effect of volume on efficiency, effect of the output of log parsers on log mining and further steps in log analysis pipeline.

II. IMPLEMENTED METHODOLOGY

A. Fixed Depth Parsing Tree/ Directed Acyclic Graph Approach

Drain[29] is a type of online log parser that first showed a parsing tree with fixed depth approach and improved on the same idea in a future paper with a DAG approach, by providing a new log parsing method and automating the parameter tuning mechanism.

In the first approach, it constructs a parse tree of fixed depth which prevents the tree from being unbalanced and lessens the traversals that can occur in very deep trees. It proposes a 5-step approach – Pre-processing by Domain Knowledge, Search by Log Message Length, Search by Preceding Tokens, Search by Token Similarity and Update the Parse Tree. In the Pre-processing stage, Drain uses regular expressions that are user-provided based on prior understanding of the logs domain like Apache, Hadoop, etc. These simple regexes are used to filter out frequently used variables like IP Addresses and other identification variables, and replace them with a generic constant. In the second stage, the message length ‘X’ which is equal to the number of tokens is calculated, and the tree is traversed from the root node to the appropriate 1st layer length node, such that all messages of the same length reach the same 1st layer node. In the third stage, (*depth* -2) number of traversals within the internal nodes occurs, where *depth* is the fixed depth of the parse tree. This stage also ensures avoidance of branch explosions by checking if any token has digits, which by experience[21] is found to be a variable, also called a wildcard. Any token containing digits is replaced by a generic “*” and the parsing in further stages is based on the “*”. A limit on the number of children that a node can have is also placed. By the end of this stage, the traversal arrives at the leaf node of the tree. In the fourth stage, the leaf nodes contain many log groups corresponding to the given length and token. Every log group has a log event that could be a possible template for a log entry and log IDs of all the previous instances that matched it. The similarity between each log event and message is calculated by checking if the tokens at the same positions match each other.

Once the biggest similarity value *simSeq* is found, it is compared with a threshold value *st*, such that *simSeq* is greater than or equal to *st*. At the end of the fourth stage, the algorithm would have either found the closest matching log group or find that no match occurs. In the fifth stage, if a log group has been matched, the log event is updated by comparing the log event and message, token by token, and replacing the unmatching token with “*” or a wildcard. If no log group has been matched, a new path of fixed depth is formed from the root node to a new leaf node containing a log group in which the log event is the inputted unmatched log message and its ID. Through these 5 stages, Drain[29] is able to parse a log in a streaming manner. In the updated paper on Drain[33], the basic premise of dynamic creation of a directed acyclic graph at runtime remains the same from its previous version[29]. The methodology followed in the first and second stages also remain the same. However, in the third stage, now called Token Layer, the token for traversal or the split token can be taken as the first token, last token or it can remain empty. The deciding factor for choosing the token is that it must contain digits. If neither has digits, the one that does not have punctuation marks is chosen as the split token. If both the first and last token cannot satisfy the two previous conditions, the split token remains empty. Traversal to the next similarity layer node is based on the chosen token or “*” in case it is empty. In the fourth stage, now called Similarity Layer, the similarity equation is modified to not take into account wildcards.

The updated paper[33] also provides a cache mechanism implemented by maintaining a pointer to the most recent path followed and comparing the new incoming message to that path first and foremost. The major improvements proposed occur in the similarity threshold initialization and updating. Unlike other online log parsers, namely Spell[19] and SHISO[12] which require 1 and 4 parameters respectively for fine-tuning the threshold, Drain uses a dynamic self-updating threshold. Each log group has its own threshold which depends on the message length of the log event and the number of tokens in the log event that contain digits. Whenever the log event in a log group is updated, so is the threshold by taking into account the current threshold and the number of tokens in the incoming log event that contain digits, while keeping the threshold at a minimum value of 1. The significance of this process is that the threshold gets bigger every time the number of variables increases, thus making it more difficult to match to that log group. In case of over-parsing of logs, this paper[33] also proposes a way to merge log groups that are very similar by comparing their log events.

B. Longest Common Subsequence

Spell[19],[32], utilizes Longest Common Subsequence algorithm (or LCS) to perform parsing of logs. The simple idea behind LCS is that given two strings, the substring longest in length that is present in both of the given strings is found.

Unlike Drain[29],[33], Spell[19],[32], does not require any kind of pre-processing or prior domain knowledge to begin parsing logs, adhering to the true meaning of being an online stream parser. The implementation is also relatively simple compared to thetree approach previously discussed.

Every incoming log entry is converted into a sequence of tokens, in a manner where each word is considered as a token. Each log entry is also assigned a unique identifier, which is a simple integer beginning from 0 and incremented for every new entry. The parser keeps in its memory an LCSMap which contains LCSObjects. LCSObjects contain an LCSseq, which is a sequence of tokens corresponding to the log message template, and the unique identifier of the log entry that matches with the sequence. After tokenization, the token sequence is compared with the LCSseq of every LCSObject to obtain the length of the longest common subsequence of each comparison. The largest length *l* of all comparisons, is then compared to a threshold *t=—s—/2* where *s* is the length of the tokenized log message. If *l* ¿= *t*, it is considered to be matched to that template. Compared to the threshold value in Drain[29],[33], threshold in Spell[19],[32], is extremely intuitive and straightforward. Once the threshold criterion is satisfied, the algorithm backtracks to create a new template that agrees with both the new accepted sequence and the LCSseq. If the threshold criterion is not satisfied, then a new LCSObject is initialized with the LCSseq as the new log message and its unique identifier. Logan[36] is a distributed online log parser, which is also based on the Longest Common Subsequence algorithm. While it does perform some pre-processing on the data and its most salient feature is that it leverages the fact that searching for LCS can be performed parallelly. It distributes the sharded inputs among multiple independent tasks and collates the results of each.

C. Clustering

LenMa(Length Matters)[23] is an older online parser that is based on clustering algorithm. The basic premise of clustering is to create groups of similar log messages and compare new log messages with each cluster to see where it belongs.

A rudimentary way to cluster is by using the number of words in a log as the parameter. LenMa[23] improvises on the same, by parametrizing the length of words in the message to decide its cluster belongingness. The incoming log is first processed into a vector containing lengths of each word and a vector containing individual separated words of the log. Similarly, each cluster has its own length vector and word vector. Cosine similarity between the message vector and every cluster of the same vector length is calculated by taking the cosine of the two vectors. A positional similarity is also calculated which compares the word vectors and returns the number of common words. If both the similarities are greater than the developer-defined threshold values, the new log is accepted into the cluster and the length vector is updated with that of the new message, while the word vector is updated with "*" wherever a mismatch occurs. Otherwise, a new cluster with the length and word vector of the new log is created. Like Spell[19],[32], LenMa[23] also does not require any kind of pre-processing of the data. However, the threshold used in LenMa[23] is of static nature and developer-defined, unlike other discussed parsers.

One of the earliest implementations of online parsers, SHISO[12] creates a tree-like structure that can be parsed to identify the log format, where the child nodes of each parent node can be considered as a cluster. This is done in two phases- the search phase and the adjustment phase. Starting with the root node as the parent node, the search phase looks for a format node in the tree which is similar to the tokenized log message using Euclidean distance measurement. If none of the measured distance crosses the threshold which is a user-chosen value between 0 and 1, each child node is considered as the parent node and the rest of the tree is parsed. All child nodes of the parent node must have the same message length and also have a maximum limit on the number of child nodes. Once a node passes the set threshold, the adjustment phase begins. This phase updates the format of the chosen node by creating generic templates by replacing variables with wildcards and holding static words in place. To perform the adjustment phase, SHISO[12] uses n-grams to find the static and variable parts of the message. Compared to other parsers, SHISO[12] does not require any explicit pre-processing. The threshold is static and requires developer knowledge to choose the right value. The parser also suffers from the drawback that it can create imbalances in the tree, allowing certainparts to grow much deeper than others.

D. N-gram Dictionaries

Logram[42] was initially developed as an n-gram offline batch parser, but it is shown how it can easily be modified into an online parser. N-gram, where 'N' is an integer greater than 0, is a substring of 'N' words derived from the log message. For example, consider the sentence "It is raining dogs and cats", which forms five 2-grams – "It is", "is raining", " raining dogs"," dogs and" and," and cats".

Like Drain[29],[33], Logram[42] has a pre-processing step where it extracts only the message content by removing all additional information like thread name, level, time and date. It also tokenizes the extracted message. Logram[42] uses a 2-gram and a 3-gram dictionary to aid the parsing procedure. On the arrival of the first message, the dictionary is empty and thus the message is divided into its n-grams that are directly used as the dictionary entries along with their number of occurrences. As newer messages come in, the n-grams are compared with the dictionary, and if the number of occurrences for an n-gram is below a threshold, the n-1 gram is taken into consideration. If this comparison is also below the threshold, the n-gram is considered to be a dynamic n-gram, which means it contains a variable. The n-gram entries and the number of occurrences is updated with every new message. The threshold is calculated by using a smoothing function called *loess*[50], and a one-dimensional clustering method called Ckmeans[49], which finds a breaking point between groups of static n-grams and dynamic n-grams.

III. EXPERIMENTAL DATASET AND ANALYSIS

The availability of log datasets is scarce since they contain sensitive information that most companies would not like to make public due to issues of security. Zhu et al[41] conducted a tools and benchmark study on 13 parsers – SLCT[1], AEL[2], IPLoM[4], LKE[3], LFA[7], LogSig[10], SHISO[12], LogCluster[16], LenMa[23], LogMine[20], Spell[19], Drain[29] and MoLFI[35]. The parsers were tested on logs from 16 different sources – HDFS, Hadoop, Spark, ZooKeeper, OpenStack, BGL, HPC, Thunderbird, Windows, Linux, Mac, Android, HealthApp, Apache, OpenSSH, and Proxifier. These range from distributed systems, supercomputers, operating systems, mobile systems, server applications, and standalone software. The study made the whole repository of logs data public for open usage in future research, which is now called Loghub[44]. It amounts to about 77GB of data, with 440 million log entries[41]. The logs have been collected from both industrial applications and academic implementations. The metrics – efficiency, accuracy, and time complexity discussed further in this paper will be based on the Loghub[44] dataset unless otherwise specified.

A. Efficiency

EEfficiency is defined as the time taken by a parser to parse a given set of logs in a dataset[41].

Drain[33] applies its parsing algorithm on 11 datasets – BGL(4m le), HPC(375k le), Thunderbird(2k le), HDFS(10m le), Zookeeper(64 le), Hadoop(2k le), Spark(2k le), Windows(2k le), Linux(2k le), Apache(2k le) and Proxifier(9600 le)[45]. It compares its performance with 3 offline parsers – LogSig[10], LKE[3], IPLoM[4] and 2 online parsers- SHISO[12] and Spell[19]. It outperforms all the other parsers on every dataset by taking the least amount of time, followed by Spell and SHISO in that order. The highest improvement of 97.14% is noticed on the Thunderbird dataset, while the least improvement of 37.15% is noticed on the HDFS dataset. The average efficiency improvement is 65.54%. Notably, the running time of Drain is only 2 min for 4m BGL log messages and 7 min for 11m HDFS log messages.

Spell[32] uses 4 non-loghub datasets – Los Alamos HPC(433k le), Blue Gene(4m le), HDFS log(100k le) and OpenStack Cloud Log(106k le), and compares its performance with IPLoM[4] and Drain[29]. It also tests the efficiency on 4 implemen- tations of Spell – na¨ıve, with pre-filtering, with split, and with split and merge. On the HPC dataset, Spell(with pre-filtering) shows the best results with a runtime of 5seconds compared to the 75s by that of Spell(na¨ıve). Overall, IPLoM and Drain perform slightly better than Spell(with pre-filtering).On the Blue Gene dataset, Spell(with pre-filtering) outperforms IPLoM. On the HDFS and OpenStack logs, Spell(with pre-filtering) performs just as efficiently as the other two. Implementing Spell parallelly can reduce the run time considerably, enabling it to scale with large volumes.

Logan[36] measures its efficiency on 8 non-loghub datasets- Zookeeper(74k le), HPC(433k le), BGL(4.7m le), Presto-5.2M (5.2m le), HDFS(11m le), Spark-13M(13m le), Presto(16.2m le) and Spark(96.5m le), and compares its performance with

Table I Summary of Online Log Parsers

Parser	Technique	Scalable	Time Complexity	Anomaly Detectors	Log Compression	Parallel Implementation
Drain	Directed Acyclic Graph	Yes	O(n)	LogRobust	LogReducer,LogZip	Possible
Spell	Longest Common Subsequence	Yes	O(n)	Deeplog	None	Possible
Logan	Longest Common Subsequence	Yes	O(n)	None	None	Yes
LenMa	Clustering	No	O(c)	Xu et al Kimura et al Kimura et al Tongqing et al	None	N/A
SHISO	Clustering	No	O(n)	None	None	N/A
Logram	N-gram Dictionary	Yes	O(n)	None	None	Possible

Drain[33], Spell[19], SHISO[12] and IPLoM[4]. It performs the best for all datasets – with or without distributing the process. Its performance on the HDFS dataset is significantly better as it takes 28.767s (distributed) and 53.390(single node), while Drain and IPLoM take 559.679s and 379.038s respectively.

LenMa[23] tests its implementations in comparison with SHISO[12] on the Chuvakin[47] dataset, WIDE project dataset and academically collected logs dataset. No real data on the efficiency is mentioned in the paper. However, Zhu *et al.*[41] performs a comparison of efficiency of LenMa on the HDFS, BGL and Android Loghub datasets with MoLFI[35], Spell[19], Drain[29], IPLoM[4] and AEL[2]. It is observed that LenMa performs only slightly worse than Spell and Drain on the HDFS dataset, but as the volume and number of templates increases in BGL and Android, it performs significantly worse than other online parsers. Thus proving it is not the most suitable for scaling purposes.

SHISO[12] collected data from the Chuvakin Public Security Log Sharing Site[48] and split the data into two datasets. It compares its processing time with IPLoM[4] and SLCT[1], where it outperforms both with an average of 19.660s comparedto 49.733 s and 22.735s respectively.

Logram[42] compares it efficiency with some of the best performing parsers- IPLoM[4], Drain[29], Spell[19], AEL[2] and LenMa[23] on 5 datasets of the Loghub[44] repository, namely Android, BGL, HDFS, Windows and Spark.

It is observed that Logram outperforms all 5 parsers in all 5 datasets by a decent margin. Its efficiency was observed to be 1.8 to 5.1 times better than others. It also is scalable as the efficiency remains steady with increase in the volume of logs and templates.

B. Accuracy

Accuracy is defined as the number of logs parsed correctly to the log template[41]. Drain[33] showed an accuracy of 92% on the 11 datasets it parses. Compared to SHISO[12] and Spell[19], its accuracy ranked highest in every dataset. It attributes such high accuracy to the presence of its token and length layer, merging log groups in case of over-parsing, and dynamic updates to the threshold parameters. Zhu et al.[41] studied the accuracy of Drain compared to 12 other online and offline parsers by considering the same volume (2000 samples) for each of the 16 datasets. Its average accuracy ranked the highest of all at 86.5%. It was the best performing parser on the Zookeeper, BGL, Thunderbird, Windows, and Apache dataset. Its worst performance was on the Proxifier dataset at 52.7%.

Spell[32], the version with slit and merge, showed the highest accuracy of 99.94% on the HDFS dataset, greater than its comparatives IPLoM and Drain. Its basic version showed an accuracy of 98.86% on the OpenStack dataset, which was significantly higher than the rest. Zhu et al.[41] accuracy comparison ranked Spell at 4th position with an average value of 75.1%. It was the best performing parser on HDFS and Apache datasets. Its worst performance was on the Proxifier dataset at 52.7%.

Logan[36] does not provide the accuracy metric on the dataset it has used. However, it does provide information on the number of templates correctly identified by the parser. It is observed that Spell[19] and Drain[32] identify more templates than Logan in its single or distributed mode. LenMa[23] ranked a close 5th position in the Zhu et al.[41] study with accuracy 72.1%, only 3% behind Spell. It was the best performing parser on the Apache and OpenSSH datasets. Its worst performance was on the HealthApp dataset at 17.4%. SHISO[12], with an average accuracy of 66.9%, ranked at 7th position with performing its best on the Apache Dataset. Its worst performance was on the HPC dataset at 32.5%.

Logram[42] used a more refined definition of accuracy where the message is considered to be correctly parsed if the static and variable parts were correctly identified. By that definition, Logram showed an average accuracy of 82.5% on the Loghub[44] dataset, compared to the 74.8% of Drain[29] and 66.9% of Spell[19]. It was the best performing parser on 6 datasets – Apache, Hadoop, HealthApp, Linux, Mac, and OpenSSH. Its worst accuracy of 46% was observed on the Linux dataset.

C. Time Complexity

The time complexity of Drain[33] was O(n) where n is the number of log messages to be parsed. Its linear complexity supported it in being one of the fastest parsers. Drain also exploited the cache mechanism which allows it to further reduce its parsing time. The time complexity of the na¨ıve version of Spell[32] was O(m.n2) where m is the number of templates currently in the LCSMap and n is the length of the new log message. However, other implementations like Spell(with pre-filtering) had a time complexity of O(n).

While Logan[36] was implemented as a distributed parser, the time complexity of its underlying principle of LCS search in the existing templates was O(n) where n is the number of templates. The time complexity of LenMa[23] was O(c) where c is the number of clusters as the algorithm compares with each cluster centroid. The time complexity of SHISO[12] was O(n)[45] where n is the number of log messages. Logram[42] had a time complexity that depended on the construction of the n-gram dictionary and the cost of querying the dictionary, which were O(n) and O(1) respectively.

D. Effectiveness in Motivating Applications

All the discussed online parsers have proved that they can parse logs in a streaming fashion, with the ability to parse multiple types of log entries. The true test of log parsers is how effective they are when used in the log analysis and management pipeline[33]. Even if log parsers achieve high accuracy and efficiency, the usefulness of their outputs may vary highly in applications of anomaly detection, usage analysis, failure diagnosis, and performance modelling[41]. He et al.[33]analyzed the performance of Principal Component Analysis (PCA) based anomaly detection algorithm[6] on the parsed outputs of Drain, Spell and SHISO were compared on HDFS dataset of 575,061 blocks with a total of 29 log event types. The parsing accuracies of Drain, Spell, and SHISO were 99%,87% and 93% respectively. Out of the 16,838 marked anomalies, the PCA algorithm correctly detected 63% of the cases while using Drain and Spell, and 66% while using SHISO. However, 14.6% of the reported anomalies while using SHISO were false, which is much higher than 2.5% and 2.6% reported while using Spell and Drain respectively. Both Drain and Spell proved that their usage in log analysis is very effective.

Anomaly detector LogRobust[40] proposed by Xu et al. employed Drain[29] as their log parser. LenMa proposes that its parsing method can be used in the anomaly detection techniques proposed in [6],[34],[14] and [8]. Deeplog[28], a deep learning-based anomaly detector, suggests the use of Spell[19] for its log parsing step. Any real-time application produces TBs and PBs[9] worth of logs, causing companies to invest in the resources for the strenuous memory requirements[43]. Thus, the logs are compressed to reduce their sizes which gives rise to the field of log compression techniques. Popular log compressors like LogZip[38] and LogReducer[46] both use Drain[29] in their implementation.

IV. CONCLUSION

Extensive research is being carried out in the field of log parsing tools and their implementations. Parsers are needed in many applications to enable and aid the process of log analysis and management. With the discussed online parsers as seen summarized in Table 1, support for real-time log analysis has increased manifold.

Based on the metrics inspected in this paper, Drain and Logram are the front-runners in comparison to others. Drain has also been used in log analysis, management, and compression techniques. As the field evolves, more applications can find direct usage of these parsers instead of implementing their own.

REFERENCES

1. Risto Vaarandi. "A data clustering algorithm for mining patterns from event logs". In: *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. Ieee. 2003, pp. 119–126.
2. Zhen Ming Jiang et al. "Abstracting execution logs to execution events for enterprise applications (short paper)". In: *2008 The Eighth International Conference on Quality Software*. IEEE. 2008, pp. 181–186.
3. Qiang Fu et al. "Execution anomaly detection in distributed systems through unstructured log analysis". In: *2009 ninth IEEE international conference on data mining*. IEEE. 2009, pp. 149–158.
4. Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. "Clustering event logs using iterative partitioning". In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2009, pp. 1255–1264.
5. Meiyappan Nagappan, Kesheng Wu, and Mladen A Vouk. "Efficiently extracting operational profiles from execution logsusing suffix arrays". In: *2009 20th International Symposium on Software Reliability Engineering*. IEEE. 2009, pp. 41–50.
6. Wei Xu et al. "Detecting large-scale system problems by mining console logs". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 117–132.
7. Meiyappan Nagappan and Mladen A Vouk. "Abstracting log lines to log event types for mining software system logs". In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE. 2010, pp. 114–117.
8. Tongqing Qiu et al. "What happened in my network: mining network events from router syslogs". In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 2010, pp. 472–484.
9. Dionysios Logothetis et al. "In-situ MapReduce for log processing". In: *USENIX ATC*. Vol. 11. 2011, p. 115.
10. Liang Tang, Tao Li, and Chang-Shing Perng. "LogSig: Generating system events from raw textual logs". In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. 2011, pp. 785–794.
11. David Lang. "Using sec". In: *; login:: the magazine of USENIX & SAGE* 38.6 (2013), pp. 38–43.
12. Masayoshi Mizutani. "Incremental mining of system log format". In: *2013 IEEE International Conference on Services Computing*. IEEE. 2013, pp. 595–602.
13. Michael Chow et al. "The mystery machine: End-to-end performance analysis of large-scale internet services". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 217–231.
14. Tatsuaki Kimura et al. "Spatio-temporal factorization of log data for understanding network events". In: *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE. 2014, pp. 610–618.
15. Rui Ding et al. "Log2: A cost-aware logging mechanism for performance diagnosis". In: *{2015 USENIX} Annual Technical Conference ({USENIX} ATC 15)*. 2015, pp. 139–150.
16. Risto Vaarandi and Mauno Pihelgas. "Logcluster-a data clustering and pattern mining algorithm for event logs". In: *2015 11th International conference on network and service management (CNSM)*. IEEE. 2015, pp. 1–7.
17. Jieming Zhu et al. "Learning to log: Helping developers make informed logging decisions". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 415–425.
18. Monika Dávideková and Michal Gregu Ml. "Software Application Logging: Aspects to Consider by Implementing Knowledge Management". In: *2016 2nd International Conference on Open and Big Data (OBD)*. IEEE. 2016, pp. 102–107.
19. Min Du and Feifei Li. "Spell: Streaming parsing of system event logs". In: *2016 IEEE 16th International Conferenceon Data Mining (ICDM)*. IEEE. 2016, pp. 859–864.
20. Hossein Hamooni et al. "Logmine: Fast pattern recognition for log analytics". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 2016, pp. 1573–1582.
21. Pinjia He et al. "An evaluation study on log parsing and its use in log mining". In: *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE. 2016, pp. 654–661.
22. Shilin He et al. "Experience report: System log analysis for anomaly detection". In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2016, pp. 207–218.
23. Keiichi Shima. "Length matters: Clustering system log messages using length of words". In: *arXiv preprint arXiv:1611.03213* (2016).
24. Lei Zeng et al. "Computer operating system logging and security issues: a survey". In: *Security and Communication Networks* 9.17 (2016), pp. 4804–4821.
25. De-Qing Zou, Hao Qin, and Hai Jin. "Uilog: Improving log-based fault diagnosis by log analysis". In: *Journal of computer science and technology* 31.5 (2016), pp. 1038–1052.
26. Boyuan Chen and Zhen Ming Jiang. "Characterizing and detecting anti-patterns in the logging code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 71–81.
27. Boyuan Chen and Zhen Ming Jack Jiang. "Characterizing logging practices in Java-based open source software projects–a replication study in Apache Software Foundation". In: *Empirical Software Engineering* 22.1 (2017), pp. 330–374.
28. Min Du et al. "Deeplog: Anomaly detection and diagnosis from system logs through deep learning". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1285–1298.
29. Pinjia He et al. "Drain: An online log parsing approach with fixed depth tree". In: *2017 IEEE International Conferenceon Web Services (ICWS)*. IEEE. 2017, pp. 33–40.
30. Pinjia He et al. "Towards automated log parsing for large-scale log data analysis". In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (2017), pp. 931–944.
31. Anwesha Das et al. "Desh: deep learning for system health prediction of lead times to failure in hpc". In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 2018, pp. 40–51.
32. Min Du and Feifei Li. "Spell: Online streaming parsing of large unstructured system logs". In: *IEEE Transactions onKnowledge and Data Engineering* 31.11 (2018), pp. 2213–2227.
33. Pinjia He et al. "A directed acyclic graph approach to online log parsing". In: *arXiv preprint arXiv:1806.04356* (2018).
34. Tatsuaki Kimura et al. "Proactive failure detection learning generation patterns of large-scale network logs". In: *IEICE Transactions on Communications* (2018).
35. Salma Messaoudi et al. "A search-based approach for accurate identification of log message formats". In: *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE. 2018, pp. 167–16710.
36. Amey Agrawal, Rohit Karlupia, and Rajat Gupta. "Logan: A distributed online log parser". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE. 2019, pp. 1946–1951.
37. Lingfeng Bao et al. "Statistical log differencing". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 851–862.
38. Jinyang Liu et al. "Logzip: Extracting hidden structures via iterative clustering for log compression". In: *{2019 34th IEEE/ACM} International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 863–873.
39. Zhongxin Liu et al. "Which variables should i log?" In: *IEEE Transactions on Software Engineering* (2019).
40. Xu Zhang et al. "Robust log-based anomaly detection on unstable log data". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 807–817.
41. Jieming Zhu et al. "Tools and benchmarks for automated log parsing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 121–130.

42. Hetong Dai et al. "Logram: Efficient log parsing using n-gram dictionaries". In: *IEEE Transactions on SoftwareEngineering* (2020).
43. Shilin He et al. "A Survey on Automated Log Analysis for Reliability Engineering". In: *arXiv preprint arXiv:2009.0723* (2020).
44. Shilin He et al. "Loghub: a large collection of system log datasets towards automated log analytics". In: *arXiv preprint arXiv:2008.06448* (2020).
45. Diana El-Masri et al. "A systematic literature review on automated log abstraction techniques". In: *Information and Software Technology* 122 (2020), p. 106276.
46. Junyu Wei et al. "On the Feasibility of Parser-based Log Compression in Large-Scale Cloud Systems". In: *19th USENIXConference on File and Storage Technologies (FAST 21)*. 2021, pp. 249–262.
47. *A. Chuvakin. Public Security Log Sharing Site.* URL: http://log-sharing.dreamhosters.com/ (visited on 06/2016).
48. *A. Chuvakin. Public Security Log Sharing Site.* URL: http://log-sharing.dreamhosters.com/ (visited on 2009).
49. *Ckmeans.1d.dp function — r documentation*. URL: https:%20 //www. rdocumentation. org/ packages/ Ckmeans. 1d. dp/ %20versions/3.4.0-1/topics/Ckmeans.1d.dp (visited on 01/02/2020).
50. *loess function — r documentation*. URL: https://www.%20rdocumentation.org/ packages/ stats/ versions/ 3.6. 1/topics/ %20loess (visited on 01/02/2020).

AUTHORS PROFILE

Tejaswini S, is currently a final year undergraduate student in the Computer Science and Engineering department at Rashtreeya Vidyalaya College of Engineering, Bangalore, Karnataka, India. Her research interests include vision and speech perception, multi-agent systems, and knowledge mining. She has actively taken part in many consultancy and R&D funded projects. Notably, her recent works have focused on projects related to autonomous vehicles and emotion recognition through speech.

Dr. Azra Nasreen, is currently an Assistant Professor in the Computer Science and Engineering department at Rashtreeya Vidyalaya College of Engineering, Bangalore, Karnataka, India. She has 15+ years of teaching experience. Her research areas include video analytics, and high-performance computing. She has co-authored multiple papers published in various international journals. She has also presented papers in international conferences. She has also guided many consultancy and R&D funded projects.