Slop: Towards an Efficient and Universal Streaming Log Parser

Zhiyuan Zhao[1], Chenxu Wang[1,2](✉), and Wei Rao[1]

[1] School of Software Engineering, Xi'an Jiaotong University, Xi'an 710049, Shaanxi, China
[2] MoE Key Lab for Intelligent Network and Network Security, Xi'an Jiaotong University, Xi'an 710049, Shaanxi, China
cxwang@mail.xjtu.edu.cn

Abstract. System logs record useful information such as execution paths and states of running programs. Log analysis is an important part of anomaly detection which is critical for system security. A primary step for log anomaly detection is to extract structured log templates (message types) from a mass of unstructured raw logs. However, conventional log parsers are designed to work offline, which needs to collect logs for a time period and then load all logs into memory for training. This greatly limits its applications to large-scale log analysis. With the continuous increase of log scales, online streaming methods are greatly desired now. Most of existing online methods are designed for specific log systems and there still lacks a universal log parser. In this paper, we present Slop, which is an efficient and universal streaming log parser. To improve the efficiency of Slop, we first group coming log messages into different partitions according to their lengths. Then, we extract the message types from different partitions. This avoids many unnecessary comparisons between logs and existing message types. To improve the universality and accuracy, we investigate the relationships between lengths of message types and the lengths of their raw logs. Based on the uncovered results, we design a nonlinear threshold criterion for message type extraction which is adaptive to several log systems. Finally, we implement a prototype of Slop and conduct extensive experiments to validate its effectiveness and efficiency based on diverse real-world datasets. It is shown that Slop obtains 55%–82% improvements in accuracy and achieves higher efficiency than state-of-the-art methods.

Keywords: Log parsers · Log partitioning · Nonlinear thresholds

1 Introduction

Log systems record the behavior and running states of systems and programs. Logs contain useful information such as execution paths which help operators to detect execution anomalies [1–3]. Log analytics plays important roles in the building of secure and trustworthy systems. In general, log files are composed of independent lines of unstructured text data, which is called a log message. A primary step for log analysis is to convert unstructured raw logs into structured log templates which are called message types. Implementations of such techniques are generally called as log parsers.

Existing log parsers can be roughly classified into two categories, namely offline methods and online methods. Offline methods such as LKE [4] and LogSig [5] generate message types based on clustering methods which divide log messages into different clusters. Heuristic methods such as SLCT [6] and IPLoM [7] generate candidates of messages types by counting the occurrences of words at different positions. However, offline methods are often limited by system resources as they need load a mass of log data into memory. In addition, log messages used for training are collected in a specific time period. If new message types are added after training, we have to train the parser again. With the development of computer science, the scale of logs is becoming much larger than before, especially with the emergence of distributed systems. For example, a large service system like HDFS can generate around 50 GB logs (120–200 million lines) per hour [8]. Therefore, online streaming methods are greatly demanded.

Drain [9] and Spell [10] are two recent online log parsers. These methods process log messages in a streaming manner, which works incrementally as log messages are being generated. However, there are two weaknesses of current online log parsers. First, there is an improvement space for existing online methods in both accuracy and efficiency. Second, these methods are designed for specific log systems and the parameter settings are not universal. Drain needs to specify the depth of a prefix tree before parsing log messages [9]. Spell generates message types based on a linear threshold criterion which is not capable for most log systems [10].

In this paper, we present Slop, an efficient and universal *s*treaming *lo*g *p*arser. Based on the intuition that in most cases log messages have the same length if they belong to the same message type, we perform a partition step to group incoming log messages according to their lengths. Then we extract message types for each partition independently. This greatly improves the efficiency by avoiding many unnecessary searches and comparisons. In order to guarantee the accuracy of Slop, we also combine the message types in different partitions since a small number of message types may generate logs with varying lengths (e.g., logs have different numbers of parameters). In order to improve the universality of our method, we investigate the relationships between the lengths of message types and the lengths of raw log messages for several log systems. We find that these two metrics are not linearly correlated. We then propose a nonlinear threshold criterion to extract message types from the raw log messages. This greatly improves the universality and accuracy of our approach. Compared with other methods, Slop is more adaptive and requires less domain knowledge of log systems. Finally, we implement a prototype of our method and conduct extensive experiments to validate its effectiveness and efficiency based on diverse datasets. The results clearly demonstrate that Slop outperforms state-of-the-art methods in both accuracy and efficiency.

In summary, we make the following contributions:

- We improve the efficiency of Slop by grouping incoming log messages into different partitions. This avoids many unnecessary comparisons in the step of message type extraction.
- We guarantee the accuracy of Slop by aggregating message types in different partitions and merging the message types belonging to the same types.
- We improve the universality and accuracy of Slop by proposing a nonlinear threshold criterion in the step of message type extraction.
- We conduct experiments based on the data collected from five real log systems to evaluate the performance and the result clearly shows the superiority of Slop.

The rest of this paper is organized as follows: In Sect. 2, we present the terminologies. Section 3 describes the methodology of Slop. Section 4 shows the selection of a proper threshold criterion for message type extraction. In Sect. 5, we conduct extensive experiments to evaluate the performance of Slop. After a review of related work in Sect. 6, we conclude the work in Sect. 7.

2 Terminologies

In this section, we describe the terminologies used in this paper. Figure 1 presents an example of the log parsing problem. The upper box shows the raw log messages and the lower contains the extracted message types. Log messages 1 and 2 are from BlueGene/L [11], and Log messages 3 to 5 are from HDFS [12]. Message types 1 to 4 are their templates, respectively.

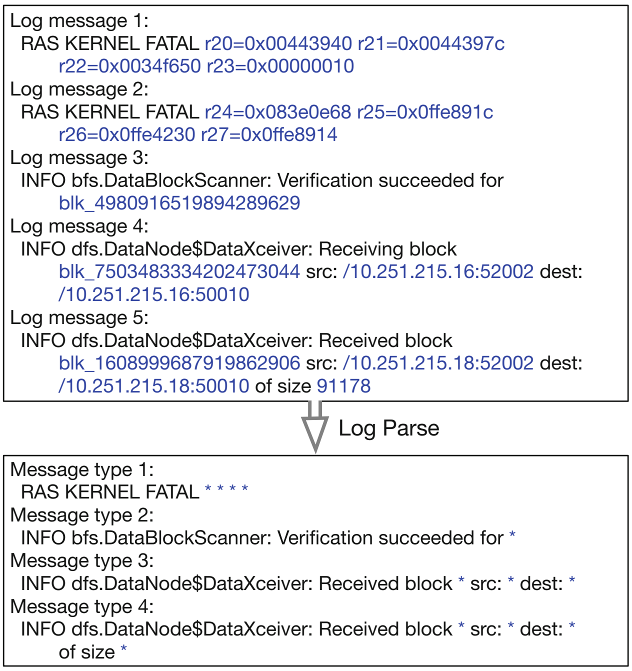
Log messages are independent text lines in a log file. It is a complete log that describes the behavior of the system or an application. A log message is constituted by constant tokens and variable tokens. The upper box in Fig. 1 contains a number of log messages, where constant and variable tokens are distinguished with black and blue colors, respectively. In the rest of this paper, we use m_i to denote a log message.

Message types are generative templates of log messages. A message type consists of constant tokens, which is the common part of a large number of log messages. The variable tokens are replaced by asterisks, as shown in the bottom box in Fig. 1. Message types have the same constant tokens but different numbers of variable tokens belong to the same type. In the rest of this paper, we use t_k to denote a message type.

Tokens are words delimited by whitespace, commas or colons in a log message. For example, RAS, KERNEL, FATAL, r20 = 0x0044397c presented in Fig. 1 are tokens. The length of a log is defined as the number of tokens.

Constant tokens are the common parts in many different log messages. For example, RAS, KERNEL, FATAL are constant tokens of log messages 1 and 2.

Variable tokens are the parts which vary in different log messages. Variable tokens are replaced by asterisks in Fig. 1. For example, the last four tokens in log messages 1 and 2 are variable tokens.



```
Log message 1:
  RAS KERNEL FATAL r20=0x00443940 r21=0x0044397c
       r22=0x0034f650 r23=0x00000010
Log message 2:
  RAS KERNEL FATAL r24=0x083e0e68 r25=0x0ffe891c
       r26=0x0ffe4230 r27=0x0ffe8914
Log message 3:
  INFO bfs.DataBlockScanner: Verification succeeded for
       blk_4980916519894289629
Log message 4:
  INFO dfs.DataNode$DataXceiver: Receiving block
       blk_7503483334202473044 src: /10.251.215.16:52002 dest:
       /10.251.215.16:50010
Log message 5:
  INFO dfs.DataNode$DataXceiver: Received block
       blk_1608999687919862906 src: /10.251.215.18:52002 dest:
       /10.251.215.18:50010 of size 91178
```

```
Message type 1:
  RAS KERNEL FATAL * * * *
Message type 2:
  INFO bfs.DataBlockScanner: Verification succeeded for *
Message type 3:
  INFO dfs.DataNode$DataXceiver: Received block * src: * dest: *
Message type 4:
  INFO dfs.DataNode$DataXceiver: Received block * src: * dest: *
       of size *
```

Fig. 1. An example of the log parsing problem (Color figure online)

3 Methodology of Slop

Figure 2 presents the basic workflow of Slop, which consists of five main steps, namely raw log preprocessing, log partitioning, message type prematching, message type extraction, and message type combination.

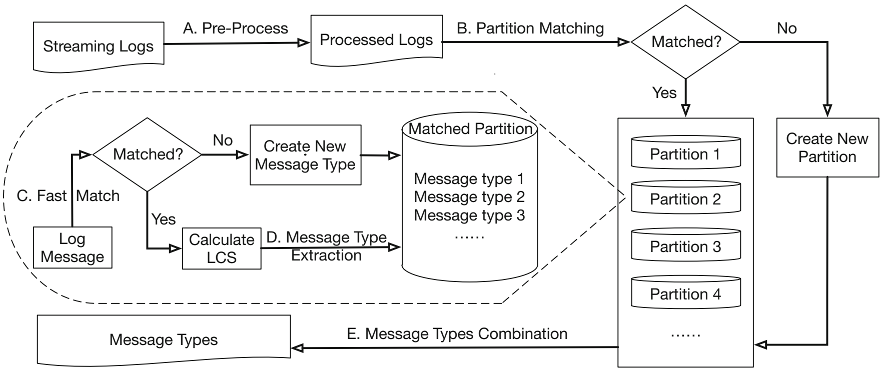


Fig. 2. Structure of slop

3.1 Raw Log Preprocessing

In this step, Slop deletes tokens that are ensured to be variable tokens, e.g., timestamps. Timestamps always increase in a single log file. However, timestamps have fixed positions in log messages and thus the efficiency can be greatly improved if we remove these tokens before log parsing. It is worth noting that this is not a necessary step. If users have no knowledge of where the timestamp is, they can choose to do nothing in this step. The accuracy of the results will not be affected without any preprocessing of the raw logs.

3.2 Log Partitioning

In this step, Slop groups an incoming log message into specific partitions according to the length. As shown in Fig. 2, for an incoming message, if it does not match any partitions according to its length, Slop will create a new partition. Message types are extracted in each partition independently. Each partition contains several message types. In the rest of this paper, we denote a partition as a set P_j, and $P_j = \{t_{1j}, t_{2j}, \ldots, t_{kj}\}$, where t_{kj} is the k-th message type in P_j. Clearly, partitioning the messages into small groups reduce many unnecessary comparisons. This is because the number of message types in each partition is much smaller than the total number of message types. In most cases, log messages have the same length if they belong to the same message type. Some log messages belonging to the same message type may have different lengths and thus are grouped into different partitions. We handle this problem by combining all message types and merge message types belonging to the same type. The details are described as in Sect. 3.5.

3.3 Message Type Prematching

When an incoming message m_i is grouped into a partition P_j, Slop first calculates the intersections between m_i and $t_{kj} \in P_j$. If the intersection length satisfies the threshold criterion (see Sect. 4), then m_i is a possible realization of t_{kj}. In the next step Slop extracts the message type and parameters of m_i, and update the message type t_{kj}. If the intersection length does not satisfy the threshold criterion for any message types, then a new message type is created and added to the partition.

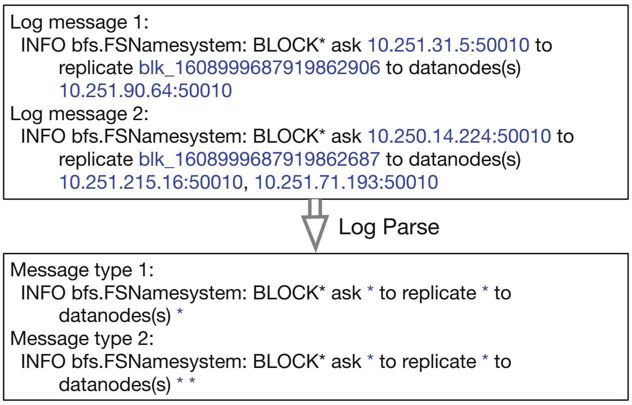
3.4 Message Type Extraction

If an incoming message m_i is prematched with a message type t_{kj}, Slop extract the message type and parameters of m_i, and update t_{kj} using the LCS (Longest Common Subsequence) method. The LCS problem is to find the longest subsequence common to all sequences in a set of sequences (often just two sequences). For example, given two sequences S_1 = ABCDEFG and S_2 = ABKDEFH, the longest common subsequence of S_1 and S_2 is ABDEF. There are two key points to note here. The first key point is that LCS needs to appear in both sequences simultaneously. The other is that the order of tokens in LCS needs to be the same order it appears in both sequences.

When we obtain the LCS between m_i and t_{kj}, then the remaining tokens (variable tokens) are parameters. We then replace the variable tokens with asterisks to obtain the message type of m_i. If the obtained message type is different from t_{kj}, we use it substitute t_{kj}.

3.5 Message Types Combination

As mentioned previously, the partitioning step may divide the log messages belonging to the same type into different groups. In this step, Slop combines all message types in different partitions to obtain the final set of message types and merges message types belonging to the same type. Figure 3 presents an example of two log messages from HDFS [12]. At the end of m_1, there is just one IP address as its parameter while there are two IP addresses in m_2. Although m_1 and m_2 have different lengths, they belong to the same message type.



```
Log message 1:
  INFO bfs.FSNamesystem: BLOCK* ask 10.251.31.5:50010 to
       replicate blk_1608999687919862906 to datanodes(s)
       10.251.90.64:50010
Log message 2:
  INFO bfs.FSNamesystem: BLOCK* ask 10.250.14.224:50010 to
       replicate blk_1608999687919862687 to datanodes(s)
       10.251.215.16:50010, 10.251.71.193:50010
```

```
Message type 1:
  INFO bfs.FSNamesystem: BLOCK* ask * to replicate * to
       datanodes(s) *
Message type 2:
  INFO bfs.FSNamesystem: BLOCK* ask * to replicate * to
       datanodes(s) * *
```

Fig. 3. An example from HDFS logs

To tackle this problem, we propose an algorithm to combine the obtained message types. Assume the there are n partitions and each partition contains a number of message types. Denote the set of final message types as P. Slop compares the message types from each partition with that in P. If a message type is a subsequence of the other, then the two message types will be combined as a single message type. The pseudo-code of the algorithm is presented in Algorithm 1. The function $intersection(t_{kj}, t_i)$ calculates the length of common constant tokens between t_{kj} and t_i. $|t_i|$ is the length of the message type t_i, which is the number of constant tokens.

Algorithm 1. Message types combination algorithm

```
Input: Partitions
Output: $P$: The final message type set
Initialization: $P = \emptyset$
1 for $j \leftarrow 1 : n$ do
2     foreach $t_{kj} \in P_j$ do
3         flag = True;
4         foreach $t_i \in P$ do
5             if $intersection(t_{kj}, t_i) == \min(|t_{kj}|, |t_i|)$ then
6                 flag = False;
7                 break;
8         if flag then
9             Add $t_{kj}$ to $P$;
```

It is worth noting that the algorithm is efficient since the total number of message type are usually small. For example, an HDFS [12] log file which contains more than 11 million log messages just has 39 message types. Additionally, the combination can be performed periodically with a much longer time interval. This further improves the efficiency.

4 A Nonlinear Threshold Criterion

In this section, we first present why a proper threshold criterion is necessary for message type extraction and the weaknesses of the linear threshold criterion. We then introduce a nonlinear threshold criterion which is universal for several common log systems.

4.1 The Necessity of a Proper Threshold

Although log messages m_3 and m_4 in Fig. 1 share a common token "INFO", they do not belong to the same message type. That is, we need a threshold to determine when an LCS can be regarded as a message type. In the rest of this paper, we use l_{ij} to denote the length of LCS between log messages m_i and m_j, and ω to denote the threshold. That is, if and only if the length of the LCS is greater than the threshold ω, these two log messages can be regarded to belong to the same message type.

However, how to decide a proper threshold criterion is a core and difficult problem. Spell introduces a linear threshold criterion which is defined as $\omega = |m_i|/2$ [10], where $|m_i|$ is the length of m_i. For example, the length l_{34} of the LCS of m_3 and m_4 is 1, and $\omega = 3$ (i.e., 6/2). Since l_{34} is not greater than ω, the LCS "INFO" of m_3 and m_4 cannot be regarded as a message type.

Such a threshold criterion works well for specific log systems. However, it encounters serious problems when applied to other log systems. For example,

Table 1. Statistical results for different log systems

Log systems	Logs	Logs ($l_{ij} \leqslant \omega$)	MT	MT ($l_{ij} \leqslant \omega$)
BlueGene/L	2000	1297 (65%)	112	29 (26%)
HDFS	2000	1061 (53%)	14	5 (36%)
HPC	2000	1331 (67%)	44	27 (61%)
Proxifier	2000	1958 (98%)	7	3 (43%)
Zookeeper	2000	324 (16%)	46	12 (26%)

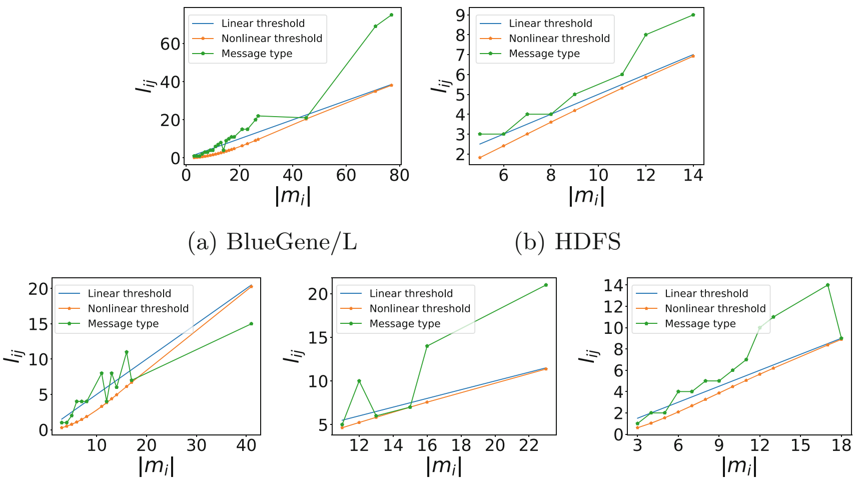
the lengths of log messages m_1 and m_2 in Fig. 1 are both 7, and their LCS l_{12} is of length 3. Thus, the log parser will determine that the two messages belong to different message types. We investigate such a problem in five log systems, namely BlueGene/L, HDFS, HPC, Proxifier, and Zookeeper. He et al. provide the datasets for the five system logs with ground-truth message types in their work [13]. They randomly select 2000 log messages from every dataset and extract their message types. We then examine how many logs and message types that violate the linear threshold criterion. The results are presented in Table 1. It is found that a large fraction of logs (the 3rd column) and message types (the 5th column) disobey the criterion. For instance, nearly 65% log messages have an LCS whose length is not greater than half the length of the message. These messages are generated from about 26% message types. The statistical results clearly show such a linear threshold criterion will result in high message type extraction errors. It is necessary to find a proper threshold criterion which is adaptive to different system logs.

4.2 Nonlinear Threshold

In order to find a proper threshold criterion, we look insight into the relationship between the length of log messages and the length of its corresponding message types based on the above five log systems. The results are shown in Fig. 4. The x-axis represents the length of log messages and the y-axis is the length of message types. We also plot the linear threshold criterion in the figures. It is shown that when the length of log messages is small, the length of most message types is below the line of the linear threshold. This indicates that short message types have a big chance to violate the criterion. Therefore, we should find a proper threshold criterion that satisfies: (i) The threshold curve should approximate to the message type curve and (ii) it should not surpass above the message type curve.

Take the above analysis into consideration, we find a proper nonlinear threshold criterion:

$$\omega(m_i) = \frac{1}{2}|m_i| \tanh(\frac{|m_i|}{\max(|m|)}T), \tag{1}$$



(a) BlueGene/L (b) HDFS

(c) HPC (d) Proxifier (e) Zookeeper

Fig. 4. Effectiveness of *tanh*.

where $|m_i|$ is the length of log messages, $\max(|m|)$ is the maximum length of log messages for a specific log system, T is an adjustable constant, and

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2}$$

The motivation of the above nonlinear threshold criterion is to reconcile the linear one with a $\tanh(x)$ function. However, we have to determine a proper value of the parameter T in order to satisfy the previously mentioned two constraints. The curve of the $\tanh(x)$ function is plotted in Fig. 5. We focus on the first quadrant of the $\tanh(x)$ function since the length of log messages are always positive. It is shown that the value of $\tanh(x)$ increases with the increase of x with a decreasing rate. The values of $\tanh(x)$ approximate to 1.0 when x is large enough. In fact, we want to apply the nonlinearity of $\tanh(x)$ when the length of a log message $|m_i|$ is small. Therefore, we have to scale the value of $\frac{|m_i|}{\max(|m|)}$ in order to satisfy the constraints. We find that when $x \geqslant 2.64$, it is enough to guarantee the value of $\tanh(x)$ approximate to 1.0. Hence, in this paper we set $T = 2.64$.

We also plot the nonlinear threshold criterion in Fig. 4. We can see that in all datasets, our nonlinear threshold criterion works very well with only one exception in the HPC log system [14]. The exceptional message type has a length of 44, and the logs corresponding to this message type just appear only once in the 2000 log messages.

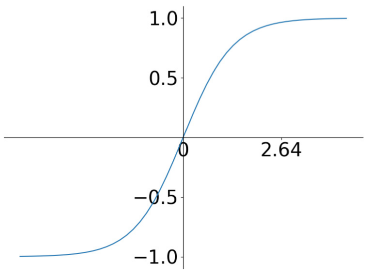


Fig. 5. The Tanh function

5 Experiments

In this section, we first introduce the datasets to conduct the experiments. Then we evaluate the effectiveness and efficiency of our approach by comparing with the state-of-the-art methods.

5.1 Datasets Description

We employ datasets collected from 5 real log systems for the experiments. Table 2 shows the statistical information of the datasets. *BlueGene/L* is collected by LLNL (Lawrence Livermore National Labs). It contains logs collected from a supercomputer called BlueGene/L [11]. *HDFS* is collected from a 203 nodes cluster which is deployed on the Amazon EC2 platform [12]. *HPC* is collected from a cluster, too, which has 49 nodes [14]. *Proxifier* is collected from a software called Proxifier. *Zookeeper* is a dataset used in [9] and it is collected from a 32-node cluster. These datasets can be obtained from [15].

Table 2. Datasets information

System	Log messages	Message types
BlueGene/L	4,747,963	183
HDFS	11,175,629	39
HPC	433,490	50
Proxifier	10,108	9
Zookeeper	74,380	64

5.2 Parameter Settings

We compare Slop with four log parsers namely IPLoM [7], LogSig [5], Spell [10], and Drain [9], which are proposed in 2009, 2011, 2016, and 2017, respectively. We left those methods that are very old like SLCT [6] which is proposed in 2003. We have tuned the parameters of these methods to achieve their best performance. Table 3 shows the parameters for different datasets. For IPLoM, *ct* is used to avoid further partitioning. Its value ranges in (0, 1]. LB is the lower bound to control how to find a bijective [7]. For LogSig, k is the number of message types [5,9,13]. Drain [9] has two parameters named depth (depth of fixed tree) and st. Depth is the depth of the fixed tree and st is the similarity threshold [9] which is used to select the most suitable log group for a log message. Spell and Slop need no parameters.

Table 3. Parameters of IPLoM, LogSig and Drain

Methods	Parameters	BlueGene/L	HDFS	HPC	Proxifier	Zookeeper
IPLoM	ct	0.4	0.35	0.18	0.55	0.4
	LB	0.01	0.25	0.25	0.25	0.65
LogSig	k	183	39	50	9	64
Drain	Depth	3	4	3	3	4
	st	0.3	0.4	0.5	0.3	0.3

5.3 Effectiveness Evaluation

In this experiment, we evaluate the effectiveness of Slop. The ground-truth of message types for each dataset is obtained as follows: First, we obtain four sets of message types by applying IPLoM [7], Drain [9], Spell [10] and Slop, respectively. We do not use LogSig because it needs the number of message types as its input [5]. If a message type is obtained by more than two methods, we regard it as a ground-truth. For each of the types extracted only by one method, we calculate its similarity with each of the previously obtained ground-truth. The similarity between two message types is defined as follows:

$$s = \frac{|\{t_1\} \cap \{t_2\}|}{|\{t_1\} \cup \{t_2\}|} \tag{3}$$

where $\{t_1\}$ is the set of constant tokens contained in t_1. A message type is added as a ground-truth if its similarity with any ground-truth message types is smaller than 0.5.

We first examine the performance of the methods for different log systems. We use n_{MT} to denote the number of message types generated by each method and n_G to denote the number of message types in the ground-truth. We then calculate $|n_{MT} - n_G|$ to see which method performs the best for different log systems. The result is shown in Fig. 6. The red line represents the baseline, which is always zero. It is shown that Slop performs better than other methods in almost every dataset. Spell [10] generate much more message types for HDFS [12] because the threshold criterion is not a suitable choice. Log messages whose constant tokens are less than half the length of the log messages will be regarded as a new message type. Thus, Spell extracts many redundant message types which belong to the same message type. Drain [9] also generates too many message types for Proxifier.

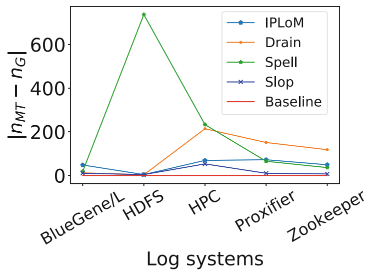


Fig. 6. Number of message types generated by each method (Color figure online)

We then employ the F1-score to evaluate the accuracy of these methods. F1-score is defined as follows:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}. \tag{4}$$

where *Precision* and *Recall* are defined as:

$$Precision = \frac{TP}{TP + FP}. \tag{5}$$

$$Recall = \frac{TP}{TP + FN}. \tag{6}$$

where TP, FP, and FN in the equations are True Positive, False Positive and False Negative, respectively. True Positive means that log messages generated by the same message type are correctly grouped together. False Positive is the case that log messages not belonging to the same message type are misclassified together. False Negative represents that message types belonging to the same message type are parsed as different message types. Table 4 shows the performance of each method on different datasets. It is shown that Slop achieves the highest scores on Precision, Recall and F1-score on all log systems except for the Zookeeper. Although Slop is not as good as IPLoM [7] and Drain [9] on Zookeeper [9], it still has good performance which is close to the highest score. Spell has very low recalls on most datasets because its FNs are too high. Slop reduces the FNs by the employment of the nonlinear threshold criterion.

5.4 Efficiency of Slop

To evaluate the efficiency of Slop, we randomly sample 20%, 40%, 60%, 80%, and 100% log messages from each dataset, respectively. The numbers of log messages in the sampled datasets are shown in Table 5. Figure 7 shows the running time of each log parser. We observe that, compared with other methods, the running time of LogSig [5] increases faster as the log size increases. This is because it uses a matrix to generate message types, whose time complexity is $O(n^2)$. IPLoM [7] shows the best performance on BlueGene/L [11]. This is because log messages from BlueGene/L are more structured than other datasets. So it is faster to count

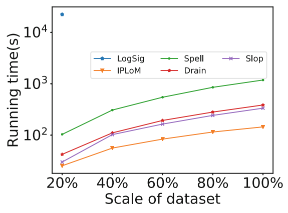
Table 4. Precision, Recall, F1-score of each methods

Dataset	Metrics	LogSig	IPLoM	Spell	Drain	Slop
BlueGene/L	Precision	0.83	0.97	0.91	0.97	**0.99**
	Recall	0.25	0.8	0.87	0.72	**0.9**
	F1-score	0.39	0.87	0.89	0.83	**0.94**
HDFS	Precision	0.82	0.92	0.92	0.92	**0.93**
	Recall	0.75	0.86	0.06	0.86	**0.93**
	F1-score	0.78	0.88	0.11	0.89	**0.93**
HPC	Precision	0.81	0.33	0.82	0.85	**0.97**
	Recall	0.56	0.25	0.73	0.69	**0.83**
	F1-score	0.66	0.29	0.77	0.76	**0.9**
Proxifier	Precision	0.73	**0.89**	0.86	0.88	**0.89**
	Recall	0.73	0.33	0.38	0.1	**0.78**
	F1-score	0.73	0.48	0.52	0.19	**0.82**
Zookeeper	Precision	0.82	0.95	**0.97**	0.95	0.94
	Recall	0.7	**0.95**	0.68	**0.95**	0.86
	F1-score	0.76	**0.95**	0.8	**0.95**	0.9

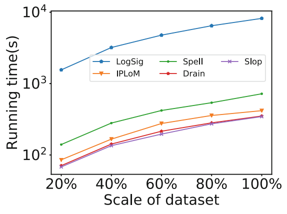
the token frequency at each position and search for bijections [7]. However, this method requires more memory than others. Spell [10] is more time-consuming than Drain [9]. Since when there is no matched message type in the prefix tree for a coming log message m_i, it has to calculate the LCS between m_i and each of the extracted message types t_{ij}. The time complexity of calculating LCS between m_i and t_{ij} is $O(|m_i||t_{ij}|)$. On the other hand, the depth of the prefix tree increases as the parsing process goes on. Drain [9] consumes slightly more time than Slop. Drain divides all log messages that contain digits to a same group. Although it restricts the maximum width and depth of the tree, there may be too many logs in a group [9]. So it costs more time for systems whose log messages contain too many digits. As shown in Fig. 7, except for the BlueGene/L dataset, Slop has the best performance among all the methods. This is because Slop partitions log messages by length, which reduces many unnecessary comparisons between log messages and message types.

Table 5. Different sizes of sample datasets

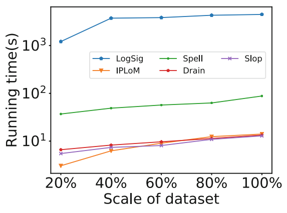
Dataset	20%	40%	60%	80%	100%
BlueGene/L	0.9m	1.8m	2.7m	3.6m	4.5m
HDFS	2.1m	4.3m	6.4m	8.5m	10.7m
HPC	84.7k	169k	254k	338.7k	423k
Proxifier	2k	3.9k	5.9k	7.9k	9.9k
Zookeeper	14.5k	29k	43.6k	58k	72.6k



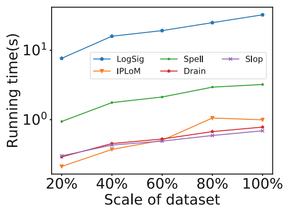
(a) BlueGene/L



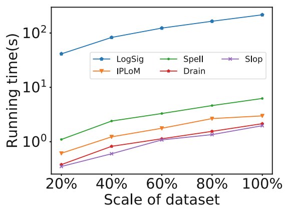
(b) HDFS



(c) HPC



(d) Proxifier

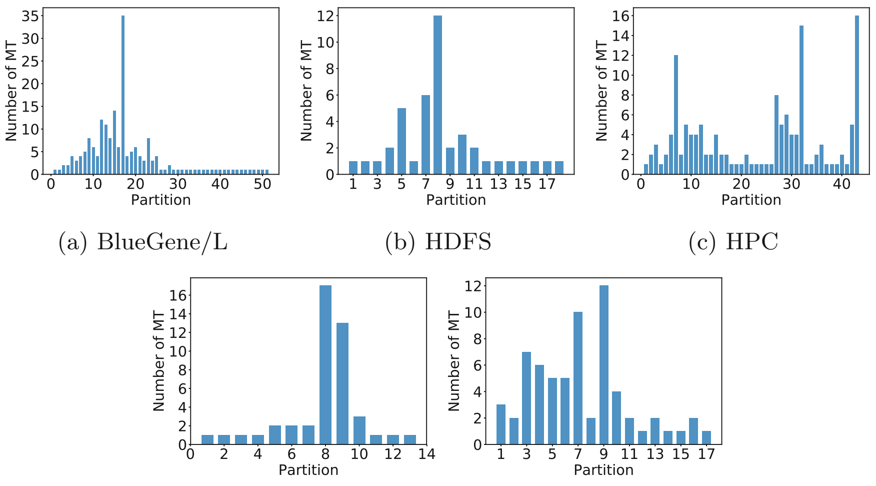


(e) Zookeeper

Fig. 7. Efficiency of each method on different log size.

The time complexity of Slop is $O(|P_j||m_i| + |t_{ij}||m_i|)n$, where $|P_j|$ is the number of message types in one partition, $|m_i|$ is the length of the log message, $|t_{ij}|$ is the length of the message type, and n is the number of log messages. For every log message, $|P_j||m_i|$ is the time complexity of prematching, and $|t_{ij}||m_i|$ is the time complexity of calculating LCS. These are only one calculation of LCS for a log message. Obviously, $|m_i|$ and $|t_{ij}|$ can be regarded as constants since they are far less than the number of log messages. The number of message types $|P_j|$ in each partition can also be regarded as a constant. Figure 8 plots the number of message types in each partition. Take the BlueGene/L dataset which has the most message types among the five datasets as an example, Slop produces 173 message types totally. With partitioning, it only needs to compare at most 35 times instead of 173 times in the stage of prematching. In the other four datasets, the number of message types in almost all partitions are less than 20. The number of message types $|P_j|$ in each partition is much smaller than non-partition, which greatly reduces the time complexity.

We also evaluate the average time for each online method to find the message type of a newly incoming log message. Table 6 shows the results. It is shown that Slop has the best performance to find or generate the message type of a log message. Such results clearly demonstrate that Slop satisfies the online processing requirement.



(a) BlueGene/L (b) HDFS (c) HPC

(d) Proxifier (e) Zookeeper

Fig. 8. Number of message types in different partitions.

Table 6. Average lookup time of online methods

Method	BlueGene/L	HDFS	HPC	Proxifier	Zookeeper
Spell (ms)	0.246	0.067	0.201	0.149	0.083
Drain (ms)	0.083	**0.031**	0.033	0.035	0.03
Slop (ms)	**0.074**	**0.031**	**0.032**	**0.032**	**0.029**

6 Related Work

There have been many researches on log parsing, which is critical for log anomaly detections such as DDoS attack detections [3] and performance failure detections [16–20]. To the best of our knowledge, the first log parser is called SLCT [6]. SLCT first calculates the distance between log messages and then employs a clustering technique to divide log messages into different groups, and finally extract the message types. Xu et al. [17,21] proposed a method to automatically generate message types through source code. However, in practice source codes are often not available in most cases. Through pre-defined regular expressions, message types could also be extracted from raw logs [4]. However, the definition of regular expressions needs domain knowledge, which is a high requirement for most log parser users. IPLoM [7] consists of three steps to group raw logs iteratively and search for bijective relationships between logs. LogSig [5] uses a metric to determine which message type the raw log belongs to. It requires the number of message types as input. However, in practice, it is hard to determine how many message types a specified system has.

To the best of our knowledge, Drain is the latest log parser proposed by He et al. in 2017 [9]. It is an online streaming log parser which parses raw logs using a fixed depth tree. It also needs the user to write regular expressions to preprocess the raw logs, which need domain knowledge like LKE [4]. Moreover, the parameter configurations of Drain [9] varies from system to system. It is usually hard to specify the parameters for different log systems. Spell [10] is another online streaming log parser. It parses log messages through computing longest common subsequence between log messages and message types. Every incoming log message needs to be compared with all message types until the message type of the log message is found. As the number of message types increases, this process becomes time-consuming and there are many unnecessary comparisons between log messages and message types.

In summary, all these methods face a common problem: different log systems need different parameters. However, we often do not know how to specify these parameters for different log systems. Although Spell does not need to tune the parameters by users, it is not adaptive to other systems. In our method, we design a nonlinear threshold criterion which is adaptive to most systems. We also partition the log messages according to their lengths to improve the efficiency.

7 Conclusion

With the continuous increase of log scale, online log parser is greatly desired now. In this paper, we propose Slop, an efficient and universal online streaming log parser. We improve the efficiency of Slop by grouping log messages into partitions. The message types extracted from different partitions are then combined and merged to guarantee its accuracy. We improve the universality of Slop by employing a nonlinear threshold criterion for message type extraction. We implement a prototype of Slop and conduct extensive experiments to evaluate its effectiveness and efficiency. The experimental results show that Slop outperforms the state-of-the-art log parsers in terms of accuracy and efficiency.

Acknowledgment. The research presented in this paper is supported in part by National Natural Science Foundation (No. 61602370, 61672026, 61772411, U1736205), Postdoctoral Foundation (No. 201659M2806, 2018T111066), Fundamental Research Funds for the Central Universities (No. 1191320006), Natural Science Foundation of Shaanxi Province (No. 2016JM6040), Postdoctoral Foundation of Shaanxi Province, CCF-Tencent Open Fund WeBank Special Funding, CCF-NSFOCUS open Fund.

References

1. Lou, J.-G., Fu, Q., Yang, S., Xu, Y., Li, J.: Mining invariants from console logs for system problem detection. In: USENIX Annual Technical Conference (2010)
2. Lou, J.-G., Fu, Q., Yang, S., Li, J., Wu, B.: Mining program workflow from interleaved traces. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 613–622. ACM (2010)

3. Yamanishi, K., Maruyama, Y.: Dynamic syslog mining for network failure monitoring. In: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, pp. 499–508. ACM (2005)
4. Fu, Q., Lou, J.-G., Wang, Y., Li, J.: Execution anomaly detection in distributed systems through unstructured log analysis. In: Ninth IEEE International Conference on Data Mining, ICDM 2009, pp. 149–158. IEEE (2009)
5. Tang, L., Li, T., Perng, C.-S.: LogSig: generating system events from raw textual logs. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, pp. 785–794. ACM (2011)
6. Vaarandi, R.: A data clustering algorithm for mining patterns from event logs. In: 3rd IEEE Workshop on IP Operations & Management, IPOM 2003, pp. 119–126. IEEE (2003)
7. Makanju, A.A., Zincir-Heywood, A.N., Milios, E.E.: Clustering event logs using iterative partitioning. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1255–1264. ACM (2009)
8. Mi, H., Wang, H., Zhou, Y., Lyu, M.R.-T., Cai, H.: Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. IEEE Trans. Parallel Distrib. Syst. **24**(6), 1245–1255 (2013)
9. He, P., Zhu, J., Zheng, Z., Lyu, M.R.: Drain: an online log parsing approach with fixed depth tree. In: 2017 IEEE International Conference on Web Services, ICWS, pp. 33–40. IEEE (2017)
10. Du, M., Li, F.: Spell: streaming parsing of system event logs. In: 2016 IEEE 16th International Conference on Data Mining, ICDM, pp. 859–864. IEEE (2016)
11. Oliner, A., Stearley, J.: What supercomputers say: a study of five system logs. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, pp. 575–584. IEEE (2007)
12. A. EC2. https://aws.amazon.com/tw/ec2/
13. He, S., Zhu, J., He, P., Lyu, M.R.: Experience report: system log analysis for anomaly detection. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering, ISSRE, pp. 207–218. IEEE (2016)
14. Los Alamos National Security LLC: Operational data to support and enable computer science research, January 2018. https://institutes.lanl.gov/data/fdata
15. LogPAI Team: Loghub, January 2018. https://doi.org/10.5281/zenodo.1147681
16. Du, M., Li, F., Zheng, G., Srikumar, V.: DeepLog: anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1285–1298. ACM (2017)
17. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 117–132. ACM (2009)
18. Ding, R., et al.: Log2: a cost-aware logging mechanism for performance diagnosis. In: USENIX Annual Technical Conference, pp. 139–150 (2015)
19. Zou, D.-Q., Qin, H., Jin, H.: UiLog: improving log-based fault diagnosis by log analysis. J. Comput. Sci. Technol. **31**(5), 1038–1052 (2016)
20. Zhang, W., Bastani, F., Yen, I.-L., Hulin, K., Bastani, F., Khan, L.: Real-time anomaly detection in streams of execution traces. In: 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering, HASE, pp. 32–39. IEEE (2012)
21. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.: Online system problem detection by mining patterns of console logs. In: Ninth IEEE International Conference on Data Mining, ICDM 2009, pp. 588–597. IEEE (2009)