Drain: An Online Log Parsing Approach with Fixed Depth Tree

Pinjia He*, Jieming Zhu*, Zibin Zheng†, and Michael R. Lyu*
*Computer Science and Engineering Department, The Chinese University of Hong Kong, China
{pjhe, jmzhu, lyu}@cse.cuhk.edu.hk
†Key Laboratory of Machine Intelligence and Advanced Computing (Sun Yat-sen University), Ministry of Education
School of Data and Computer Science, Sun Yat-sen University, China
zhzibin@mail.sysu.edu.cn

***Abstract*—Logs, which record valuable system runtime information, have been widely employed in Web service management by service providers and users. A typical log analysis based Web service management procedure is to first parse raw log messages because of their unstructured format; and then apply data mining models to extract critical system behavior information, which can assist Web service management. Most of the existing log parsing methods focus on offline, batch processing of logs. However, as the volume of logs increases rapidly, model training of offline log parsing methods, which employs all existing logs after log collection, becomes time consuming. To address this problem, we propose an online log parsing method, namely Drain, that can parse logs in a streaming and timely manner. To accelerate the parsing process, Drain uses a fixed depth parse tree, which encodes specially designed rules for parsing. We evaluate Drain on five real-world log data sets with more than 10 million raw log messages. The experimental results show that Drain has the highest accuracy on four data sets, and comparable accuracy on the remaining one. Besides, Drain obtains 51.85%∼81.47% improvement in running time compared with the state-of-the-art online parser. We also conduct a case study on an anomaly detection task using Drain in the parsing step, which determines the effectiveness of Drain in log analysis.**

***Index Terms*—Log parsing; Online algorithm; Log analysis; Web service management;**

I. Introduction

The prevalence of cloud computing, which enables on-demand service delivery, has made Service-oriented Architecture (SOA) a dominant architectural style. Nowadays, more and more developers leverage existing Web services to build their own systems because of their rich functionality and "plug-and-play" property. Although developing Web service based system is convenient and lightweight, Web service management is a significant challenge for both service providers and users. Specifically, service providers (e.g., Amazon EC2 [1]) are expected to provide services with no failures or SLA (service-level agreement) violations to a large number of users. Similarly, service users need to effectively and efficiently manage the adopted services, which have been discussed in many recent works (e.g., Web service monitoring [2]). In this context, log analysis based service management techniques, which employ service logs to achieve automatic or semi-automatic service management, have been widely studied.

Logs are usually the only data resource available that records service runtime information. In general, a log message is a line of text printed by logging statements (e.g., *printf()*, *logging.info()*) written by developers. Thus, log analysis techniques, which apply data mining models to get insights of system behaviors, are in widespread use for service management. For service providers, there are studies in anomaly detection [3], [4], fault diagnosis [5], [6] and performance improvement [7]. For service users, typical examples include business model mining [8], [9] and user behavior analysis [10], [11].

Most of the data mining models used in these log analysis techniques require structured input (e.g., an event list or a matrix). However, raw log messages are usually unstructured, because developers are allowed to write free-text log messages in source code. Thus, the first step of log analysis is log parsing, where unstructured log messages are transformed into structured events. An unstructured log message, as in the following example, usually contains various forms of system runtime information: *timestamp* (records the occurring time of an event), *verbosity level* (indicate the severity level of an event, e.g., INFO), and *raw message content* (free-text description of a service operation).

```
081109 204655 556 INFO dfs.DataNode$PacketResponder
: Received block blk_3587508140051953248 of size 67
108864 from /10.251.42.84
```

Traditionally, log parsing relies heavily on regular expressions [12], which are designed and maintained manually by developers. However, this manual method is not suitable for logs generated by modern services for the following three reasons. First, the volume of logs is increasing rapidly, which makes the manual method prohibitive. For example, a large-scale service system can generate 50 GB logs (120∼200 million lines) per hour [13]. Second, as open-source platforms (e.g., Github) and Web service become popular, a system often consists of components written by hundreds of developers globally [3]. Thus, people in charge of the regular expressions may not know the original logging purpose, which makes manual management even harder. Third, logging statements in modern systems updates frequently (e.g., hundreds of new logging statements every month [14]). In order to maintain a correct regular expression set, developers need to check all logging statements regularly, which is tedious and error-prone.

Log parsing is widely studied to parse the raw log messages automatically. Most of existing log parsers focus on offline, batch processing. For example, Xu et al. [3] design a method to automatically generate regular expressions based on source code. However, source code is often inaccessible in practice (e.g., Web service components). For general log parsing, recent studies propose data-driven methods [4], [15], which directly extract log templates from raw log messages. These log parsers are offline, and limited by the memory of a single computer. Besides, they fail to align with the log collecting manner. A typical log collection system has a log shipper installed on each node to forward log entries in a streaming manner to a centralized server that contains a log parser [16]. The offline log parsers need to employ all logs after log collection for a certain period (e.g., 1h) for the parser training. In contrast, an online log parser parses logs in a streaming manner, and it does not require an offline training step. Thus, current systems highly demand online log parsing, which is only studied in a few preliminary works [16], [17]. However, we observe that the parsers proposed in these works are not accurate and efficient enough, which make them not eligible for log parsing in modern Web service or Web service based systems.

In this paper, we propose an online log parsing method, namely Drain, that can accurately and efficiently parse raw log messages in a streaming manner. Drain does not require source code or any information other than raw log messages. Drain can automatically extract log templates from raw log messages and split them into disjoint log groups. It employs a parse tree with fixed depth to guide the log group search process, which effectively avoids constructing a very deep and unbalanced tree. Besides, specially designed parsing rules are compactly encoded in the parse tree nodes. We evaluate Drain on five real-world log data sets with more than 10 million raw log messages. Drain demonstrates the highest accuracy on four data sets, and comparable accuracy on the remaining one. Besides, Drain obtains 51.8%∼81.47% improvement in running time compared with the state-of-the-art online parser [16]. We also demonstrate the effectiveness of Drain in log analysis by tackling a real-world anomaly detection task [3].

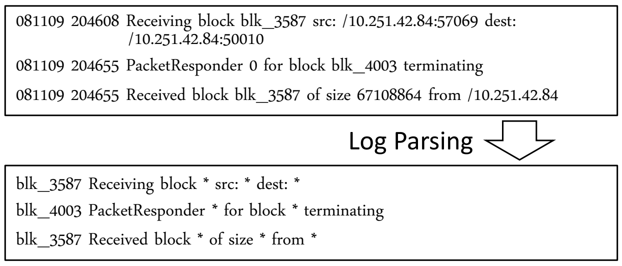
In summary, our paper makes the following contributions:

- This paper presents the design of an online log parsing method (Drain), which encodes specially designed parsing rules in a parse tree with fixed depth.
- Extensive experiments have been conducted on five real-world log data sets, which determine the superiority of Drain in terms of accuracy and efficiency.
- The source code of Drain has been publicly released [18], allowing for easy use by researchers and practitioners for future study.

The remainder of this paper is organized as follows. Section II presents the overview of log parsing process. Section III describes our online log parsing method, Drain. We evaluate the performance of Drain in Section IV. Related work is introduced in Section V. Finally, we conclude this paper in Section VI.

II. Overview of Log Parsing

The goal of log parsing is to transform raw log messages into structured log messages, as described in Figure 1.



```
081109 204608 Receiving block blk_3587 src: /10.251.42.84:57069 dest:
              /10.251.42.84:50010
081109 204655 PacketResponder 0 for block blk_4003 terminating
081109 204655 Received block blk_3587 of size 67108864 from /10.251.42.84
```

```
blk_3587 Receiving block * src: * dest: *
blk_4003 PacketResponder * for block * terminating
blk_3587 Received block * of size * from *
```

Fig. 1: Overview of Log Parsing

Specifically, raw log messages are unstructured data, including timestamps and raw message contents. The raw log messages in Figure 1 are simplified HDFS raw log messages collected on the Amazon EC2 platform [3]. In the parsing process, a parser distinguishes between the *constant* part and *variable* part of each raw log message. The constant part is tokens that describe a system operation template (i.e., log event), such as “Receiving block * src: * dest: *” in Figure 1; while the variable part is the remaining tokens (e.g, “blk_3587”) that carry dynamic runtime system information. A typical structured log message contains a matched log event and fields of interest (e.g, the HDFS block ID “blk_3587”). Typical log parsers [4], [15], [16], [17] regard log parsing as a clustering problem, where they cluster raw log messages with the same log event into a log group. The following section introduces our proposed log parser, which clusters the raw log messages into different log groups in a streaming manner.

III. Methodology

In this section, we briefly introduce Drain, a fixed **d**epth t**r**ee b**a**sed onl**i**ne log parsi**n**g method. When a new raw log message arrives, Drain will preprocess it by simple regular expressions based on domain knowledge. Then we search a log group (i.e., leaf node of the tree) by following the specially-designed rules encoded in the internal nodes of the tree. If a suitable log group is found, the log message will be matched with the log event stored in that log group. Otherwise, a new log group will be created based on the log message. In the following, we first introduce the structure of the fixed depth tree (i.e., parse tree). Then we explain how Drain parses raw log messages by searching the nodes of the parse tree.

A. *Overall Tree Structure*

When a raw log message arrives, an online log parser needs to search the most suitable log group for it, or create a new log group. In this process, a simple solution is to compare the raw log message with log event stored in each log group one by one. However, this solution is very slow because the number of log groups increases rapidly in parsing. To accelerate this process, we design a parse tree with fixed depth to guide the log group search, which effectively bounds the number of log groups that a raw log message needs to compare with.

The parse tree is illustrated in Figure 2. The *root node* is in the top layer of the parse tree; the bottom layer contains the *leaf nodes*; other nodes in the tree are *internal nodes*. Root

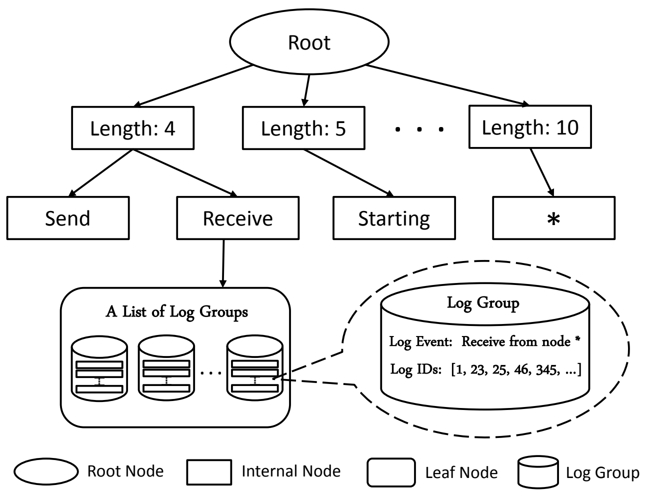


Fig. 2: Structure of Parse Tree in Drain (*depth* = 3)

node and internal nodes encode specially-designed rules to guide the search process. They do not contain any log groups. Each path in the parse tree ends with a leaf node, which stores a list of log groups, and we only plot one leaf node here for simplicity. Each log group has two parts: log event and log IDs. Log event is the template that best describes the log messages in this group, which consists of the constant part of a log message. Log IDs records the IDs of log messages in this group. One special design of the parse tree is that the depth of all leaf nodes are the same and are fixed by a predefined parameter *depth*. For example, the depth of the leaf nodes in Figure 2 is fixed to 3. This parameter bounds the number of nodes Drain visits during the search process, which greatly improves its efficiency. Besides, to avoid tree branch explosion, we employ a parameter $maxChild$, which restricts the maximum number of children of a node. In the following, for clarity, we define an n-th layer node as a node whose depth is n. Besides, unless otherwise stated, we use the parse tree in Figure 2 as an example in following explanation.

B. Step 1: Preprocess by Domain Knowledge

According to our previous empirical study on existing log parsing methods [19], preprocessing can improve parsing accuracy. Thus, before employing the parse tree, we preprocess the raw log message when it arrives. Specifically, Drain allows users to provide simple regular expressions based on domain knowledge that represent commonly-used variables, such as IP address and block ID. Then Drain will remove the tokens matched from the raw log message by these regular expressions. For example, block IDs in Figure 1 will be removed by "blk_[0-9]+".

The regular expressions employed in this step are often very simple, because they are used to match tokens instead of log messages. Besides, a data set usually requires only a few such regular expressions. For example, the data sets used in our evaluation section require at most two such regular expressions.

C. Step 2: Search by Log Message Length

In this step and step 3, we explain how we traverse the parse tree according to the encoded rules and finally find a leaf node.

Drain starts from the root node of the parse tree with the preprocessed log message. The 1-st layer nodes in the parse tree represent log groups whose log messages are of different log message lengths. By log message length, we mean the number of tokens in a log message. In this step, Drain selects a path to a 1-st layer node based on the log message length of the preprocessed log message. For example, for log message "Receive from node 4", Drain traverse to the internal node "Length: 4" in Figure 2. This is based on the assumption that log messages with the same log event will probably have the same log message length. Although it is possible that log messages with the same log event have different log message lengths, it can be handled by simple postprocessing. Besides, our experiments in Section IV-B demonstrate the superiority of Drain in terms of parsing accuracy even without postprocessing.

D. Step 3: Search by Preceding Tokens

In this step, Drain traverses from a 1-st layer node, which is searched in step 2, to a leaf node. This step is based on the assumption that tokens in the beginning positions of a log message are more likely to be constants. Specifically, Drain selects the next internal node by the tokens in the beginning positions of the log message. For example, for log message "Receive from node 4", Drain traverses from 1-st layer node "Length: 4" to 2-nd layer node "Receive" because the token in the first position of the log message is "Receive". Then Drain will traverse to the leaf node linked with internal node "Receive", and go to step 4.

The number of internal nodes that Drain traverses in this step is $(depth - 2)$, where *depth* is the parse tree parameter restricting the depth of all leaf nodes. Thus, there are $(depth - 2)$ layers that encode the first $(depth - 2)$ tokens in the log messages as search rules. In the example above, we use the parse tree in Figure 2 for simplicity, whose depth is 3, so we search by only the token in the first position. In practice, Drain can consider more preceding tokens with larger depth settings. Note that if *depth* is 2, Drain only considers the first layer used by step 2.

In some cases, a log message may start with a parameter, for example, "120 bytes received". These kinds of log messages can lead to branch explosion in the parse tree because each parameter (e.g., 120) will be encoded in an internal node. To avoid branch explosion, we only consider tokens that do not contain digits in this step. If a token contains digits, it will match a special internal node "*". For example, for the log message above, Drain will traverse to the internal node "*" instead of "120". Besides, we also define a parameter $maxChild$, which restricts the maximum number of children of a node. If a node already has $maxChild$ children, any non-matched tokens will match the special internal node "*" among all its children.

E. Step 4: Search by Token Similarity

Before this step, Drain has traversed to a leaf node, which contains a list of log groups. The log messages in these log groups comply with the rules encoded in the internal nodes along the path. For example, the log group in Figure 2 has log event "Receive from node *", where the log messages contain 4 tokens and start with token "Receive".

In this step, Drain selects the most suitable log group from the log group list. We calculate the similarity $simSeq$ between the log message and the log event of each log group. $simSeq$ is defined as following:

$$simSeq = \frac{\sum_{i=1}^{n} equ(seq_1(i), seq_2(i))}{n}, \qquad (1)$$

where seq_1 and seq_2 represent the log message and the log event respectively; $seq(i)$ is the i-th token of the sequence; n is the log message length of the sequences; function equ is defined as following:

$$equ(t_1, t_2) = \begin{cases} 1 & \text{if } t_1 = t_2 \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

where t_1 and t_2 are two tokens. After finding the log group with the largest $simSeq$, we compare it with a predefined similarity threshold st. If $simSeq \geq st$, Drain returns the group as the most suitable log group. Otherwise, Drain returns a flag (e.g., None in Python) to indicate no suitable log group.

F. Step 5: Update the Parse Tree

If a suitable log group is returned in step 4, Drain will add the log ID of the current log message to the log IDs in the returned log group. Besides, the log event in the returned log group will be updated. Specifically, Drain scans the tokens in the same position of the log message and the log event. If the two tokens are the same, we do not modify the token in that token position. Otherwise, we update the token in that token position by wildcard (i.e., *) in the log event.

If Drain cannot find a suitable log group, it creates a new log group based on the current log message, where log IDs contains only the ID of the log message and log event is exactly the log message. Then, Drain will update the parse tree with the new log group. Intuitively, Drain traverses from the root node to a leaf node that should contain the new log group, and adds the missing internal nodes and leaf node accordingly along the path. For example, assume the current parse tree is the tree in the left-hand side of Figure 3, and a new log message "Receive 120 bytes" arrives. Then Drain will update the parse tree to the right-hand side tree in Figure 3. Note that the new internal node in the 3-rd layer is encoded as "*" because the token "120" contains digits.

IV. Evaluation

A. Experimental Settings

1) Log Data Sets: The log data sets used in our evaluation are summarized in Table I. These five real-world data sets range from supercomputer logs (BGL and HPC) to distributed system logs (HDFS and Zookeeper) to standalone software logs (Proxifier). Companies rarely release their log data to the public, because it may violates confidential clauses. We obtained three log data sets from other researchers with their generous support. Specifically, BGL is a log data set collected by Lawrence Livermore National Labs (LLNL) from BlueGene/L supercomputer system [20]. HPC logs are collected from a high performance cluster, which has 49 nodes with 6,152 cores and 128GB memory per node [21]. HDFS is a log data set collected from a 203-node cluster on Amazon EC2 platform in [3]. We also collect two log data sets for evaluation. One is collected from Zookeeper installed on a 32-node cluster in our lab. The other are logs of a standalone software Proxifier.

2) Comparison: To prove the effectiveness of Drain, we compare its performance with four existing log parsing methods in terms of accuracy, efficiency and effectiveness on subsequent log mining tasks. Specifically, two of them are offline log parsers, and the other two are online log parsers. The ideas of these log parsers are briefly introduced as following:

- LKE [4]: This is an offline log parsing method developed by Microsoft. It employs hierarchical clustering and heuristic rules.
- IPLoM [15]: IPLoM conducts a three-step hierarchical partitioning before template generation in an offline manner.
- SHISO [17]: In this online parser, a tree with predefined number of children in each node is used to guide log group searching.

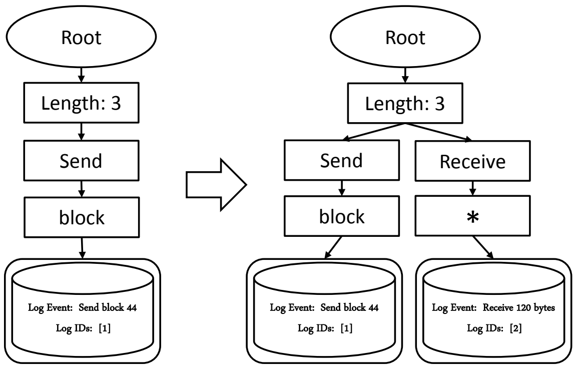


Fig. 3: Parse Tree Update Example ($depth$ = 4)

TABLE I: Summary of Log Data Sets

System	Description	#Log Messages	Log Message Length	#Events
BGL	BlueGene/L Supercomputer	4,747,963	10~102	376
HPC	High Performance Cluster (Los Alamos)	433,490	6~104	105
HDFS	Hadoop File System	11,175,629	8~29	29
Zookeeper	Distributed System Coordinator	74,380	8~27	80
Proxifier	Proxy Client	10,108	10~27	8

- Spell [16]: This method uses longest common sequence to search log group in an online manner. It accelerates the searching process by subsequence matching and prefix tree.

3) Evaluation Metric and Experimental Setup: We use F-measure [22], [23], which is a typical evaluation metric for clustering algorithms, to evaluate the accuracy of log parsing methods. The definition of accuracy is as the following.

$$Accuracy = \frac{2 * Precision * Recall}{Precision + Recall}, \tag{3}$$

where $Precision$ and $Recall$ are defined as follows:

$$Precision = \frac{TP}{TP + FP}, \tag{4}$$

$$Recall = \frac{TP}{TP + FN}, \tag{5}$$

where a true positive (TP) decision assigns two log messages with the same log event to the same log group; a false positive (FP) decision assigns two log messages with different log events to the same log group; and a false negative (FN) decision assigns two log messages with the same log event to different log groups. This evaluation metric is also used in our previous study [19] on existing log parsers.

TABLE II: Parameter Setting of Drain

	BGL	HPC	HDFS	Zookeeper	Proxifier
depth	3	4	3	3	4
st	0.3	0.4	0.5	0.3	0.3

We run all experiments on a Linux server with Intel Xeon E5-2670v2 CPU and 128GB DDR3 1600 RAM, running 64-bit Ubuntu 14.04.2 with Linux kernel 3.16.0. We run each experiment 10 times to avoid bias. For the preprocessing step of Drain (step 1), we remove obvious parameters in log messages (i.e., IP addresses in HPC&Zookeeper&HDFS, core IDs in BGL, block IDs in HDFS and application IDs in Proxifier). The parameter setting of Drain is shown in Table II. Besides, we empirically set $maxChild$ to 100 for all experiments. The number of children of a tree node rarely exceeds $maxChild$, because the encoded rules in the parse tree can already distribute the logs evenly to different paths. We also re-tune the parameters of other log parsers to optimize their performance, which is not presented here because of the space limit. We put them in our released source code [18] for further reference.

B. Accuracy of Drain

Accuracy demonstrates how well a log parser matches raw log messages with the correct log events. Accuracy is important because parsing errors can degrade the performance of subsequent log mining task. Intuitively, an offline log parsing method could obtain higher accuracy compared with an online one, because an offline method enjoys all raw log messages at the beginning of parsing, while an online method adjusts its parsing model gradually in the parsing process.

TABLE III: Parsing Accuracy of Log Parsing Methods

	BGL	HPC	HDFS	Zookeeper	Proxifier
Offline Log Parsers					
LKE	0.67	0.17	0.57	0.78	0.85
IPLoM	0.99	0.65	0.99	0.99	0.85
Online Log Parsers					
SHISO	0.87	0.53	0.93	0.68	0.85
Spell	0.98	0.82	0.87	0.99	**0.87**
Drain	**0.99**	**0.84**	**0.99**	**0.99**	0.86

TABLE IV: Running Time (Sec) of Log Parsing Methods

	BGL	HPC	HDFS	Zookeeper	Proxifier
Offline Log Parsers					
LKE	N/A	N/A	N/A	N/A	8888.49
IPLoM	140.57	12.74	333.03	2.17	0.38
Online Log Parsers					
SHISO	10964.55	582.14	6649.23	87.61	8.41
Spell	447.14	47.28	676.45	5.27	0.87
Drain	115.96	8.76	325.7	1.81	0.27
Improvement	**74.07%**	**81.47%**	**51.85%**	**65.65%**	**68.97%**

In this section, we evaluate the accuracy of two offline and two online log parsing methods on the data sets described in Table I. The evaluation results are in Table III. LKE fails to handle the data sets except Proxifier, because its $O(n^2)$ time complexity makes it too slow for the other data sets. Thus, for the other four data sets, as with the existing work [19], [24], we evaluate LKE's accuracy on sample data sets with 2k log messages randomly extracted from the original ones, while all parsers are evaluated on the 2k sample data sets in our previous paper [19].

We observe that the proposed online parsing method, namely Drain, obtains the best accuracy on four data sets, even compared with the offline log parsing methods. For data set Proxifier, Drain also has the second best accuracy (i.e., 0.86), and it is comparable to Spell, which obtains the highest accuracy (0.87) on this data set. LKE is not that good on some data sets, because it employs an aggressive clustering strategy, which can lead to under-partitioning. IPLoM obtains high accuracy on most data sets because of its specially-designed heuristic rules. SHISO uses the similarity of characters in log messages to search the corresponding log events. This strategy is too coarse-grained, which causes inaccuracy. Spell is accurate, but its strategy only based on longest common subsequence can lead to under-partitioning. Drain has the overall best accuracy for three reasons. First, it compounds both the log message length and the first few tokens, which are effective and specially-designed rules, to construct the fixed depth tree. Second, Drain only uses tokens that do not contain digits to guide the searching process, which effectively avoids over-partitioning. Third, the tunable tree depth and similar threshold st allows users to conduct fine-grained tuning on different data sets.

C. Efficiency of Drain

To evaluate the efficiency of Drain, we measure the running time of it and four existing log parsers on five real-world log data sets described in Table I. In Table IV, we demonstrate the running time of these log parsers. LKE fails to handle four data sets in reasonable time (i.e., days or weeks), so we mark the corresponding results as not available.

Considering online parsing methods, SHISO takes too much time on some data sets (e.g., takes more than 3h on BGL). This is mainly because SHISO only limits the number of children for its tree nodes, which can cause very deep parse tree. Spell obtains better efficiency performance, because it employs a prefix tree structure to store all log events found, which greatly reduces its running time. However, Spell does not restrict the depth of its prefix tree either, and it calculates the longest common subsequence between two log messages, which is time consuming. Compared with the existing online parsing methods, our proposed Drain requires the least running time on all five data sets. Specifically, Drain only needs 2 min to parse 4m BGL log messages and 6 min to parse 10m HDFS log messages. Drain greatly improves the running time of existing online parsing methods. The improvements on the five real-world data sets are at least 51.85%, and it reduce 81.47% running time on HPC. Drain also outperforms the existing offline log parsing methods. It requires less running time than IPLoM on all five data sets. Moreover, as an online log parsing method, Drain is not limited by the memory of a single computer, which is the bottleneck of most offline log parsing methods. For example, IPLoM needs to load all log messages into computer memory, and it will construct extra data structures of comparable size in runtime. Thus, although IPLoM is efficient too, it may fail to handle large-scale log data. Drain is not limited by the memory of single computer, because it processes the log messages one by one.

TABLE V: Log Size of Sample Datasets for Efficiency Experiments

BGL	400	4k	40k	400k	4m
HPC	600	3k	15k	75k	375k
HDFS	1k	10k	100k	1m	10m
Zookeeper	4k	8k	16k	32k	64k
Proxifier	600	1200	2400	4800	9600

Because log size of modern systems is rapidly increasing, a log parsing method is expected to handle large-scale log data. Thus, to simulate the increasing of log size, we also measure the running time of these log parsers on 25 sampled log data sets with varying log size (i.e., number of log messages) as described in Table V. The log messages in these sampled data sets are randomly extracted from the real-world data sets in Table I.

The evaluation results are illustrated in Figure 4, which is in logarithmic scale. In this figure, we observe that, compared with other methods, the running time of LKE raises faster as the log size increases. Because the time complexity of LKE is $O(n^2)$, and the time complexity of other methods is $O(n)$, while n is the number of log messages. IPLoM is comparable to Drain, but it requires substantial amounts of memory as explained above. Online parsing methods (i.e., SHISO, Spell, Drain) process log message one by one, and they all use a parse tree to accelerate the log event search process. Drain is faster than others because of two main reasons. First, Drain enjoys linear time complexity. The time complexity of Drain is $O(\ (d + cm)n\ )$, where d is the depth of the parse tree, c is the number of candidate log groups in the leaf node, m is the log message length, and n is the number of log messages. Obviously, d and m are constants. c can also be regarded as a constant, because the quantity of candidate log groups in each leaf node is nearly the same, and the number of log groups is far less than that of log messages. Thus, the time complexity of Drain is $O(n)$. For SHISO and Spell, the depth of the parse tree could increase during the parsing process. Second, we use the specially-designed $simSeq$ to calculate the similarity between a log message and a log event candidate. Its time complexity is $O(m_1 + m_2)$, while m_1 and m_2 are number of tokens in them respectively. In Drain, $m_1 = m_2$. By comparison, SHISO and Spell calculate the longest common subsequence between two sequences, whose time complexity is $O(m_1 m_2)$.

D. Effectiveness of Drain on Real-World Anomaly Detection Task

In previous sections, we demonstrate the superiority of Drain in terms of accuracy and efficiency. Although high accuracy is necessary for log parsing methods, it does not guarantee good performance in the subsequent log mining task. For example, because log mining could be sensitive to some critical events, little parsing error may cause an order of magnitude performance degradation in log mining [19]. To evaluate the effectiveness of Drain on subsequent log mining tasks, we conduct a case study on a real-world anomaly detection task.

We use the HDFS log data set in this case study. Specifically, raw log messages in the HDFS data set [3] records system operations on 575,061 HDFS blocks with a total of 29 log event types. Among these blocks, 16,838 are manually labeled as anomalies by the original authors. In the original paper [3], the authors employ Principal Component Analysis (PCA) to detect these anomalies. Next, we will briefly introduce the anomaly detection workflow, including log parsing and log mining. In log parsing step, all the raw log messages are parsed into structured log messages. Each structured log message contains the corresponding HDFS block ID and a log event. A source code-based log parsing method is used in the original paper, which is not discussed here because source code is inaccessible in many cases (e.g., in third party libraries). In log mining, we first use the structured log messages to generate an event count matrix, where each row represents an HDFS block; each column represents a log event type; each cell counts the occurrence of an event on a certain HDFS block. Then we use TF-IDF [25] to preprocess the event count matrix. Intuitively, TF-IDF gives lower weights to common event types, which are less likely to contribute to the anomaly detection process. Finally, the event count matrix is fed into PCA, which automatically marks the blocks as normal or abnormal.

In our case study, we evaluate the performance of the anomaly detection task with different log parsing methods

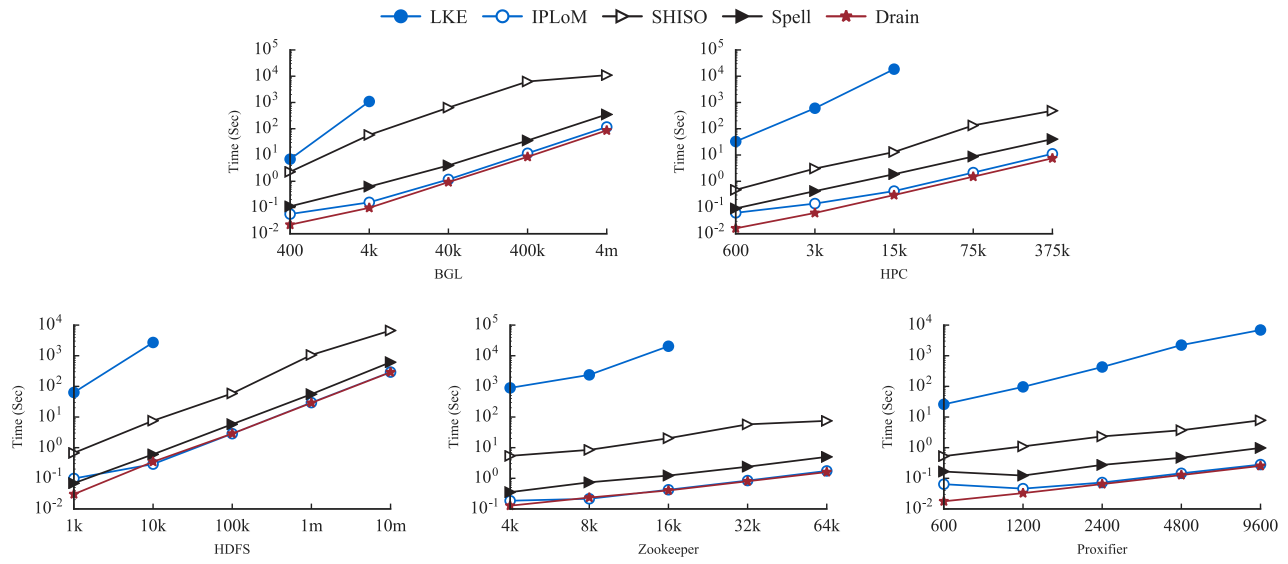


Fig. 4: Running Time of Log Parsing Methods on Data Sets in Different Size

TABLE VI: Anomaly Detection with Different Log Parsing Methods (16,838 True Anomalies)

	Parsing Accuracy	Reported Anomaly	Detected Anomaly	False Alarm
IPLoM	0.99	10,998	10,720 (63%)	278 (2.5%)
SHISO	0.93	13,050	11,143 (66%)	1,907 (14.6%)
Spell	0.87	10,949	10,674 (63%)	275 (2.5%)
Drain	0.99	10,998	10,720 (63%)	278 (2.5%)
Ground truth	1.00	11,473	11,195 (66%)	278 (2.4%)

used in the parsing step. Specifically, we use different log parsing methods to parse the HDFS raw log messages respectively and, hence, we obtain different sets of structured log messages. For example, an HDFS block ID could match with different log events by using different log parsing methods. Then, we generate different event count matrices, and fed them into PCA, respectively.

The experimental results are shown in Table VI. In this table, *reported anomaly* is the number of anomalies reported by the PCA model; *detected anomaly* is the number of true anomalies reported; *false alarm* is the number of wrongly reported ones. We use four existing log parsing methods to handle the parsing step of this anomaly detection task. We do not use LKE because it cannot handle this large amount of data. *Ground truth* is the experiment using exactly correct parsed results.

We can observe that Drain obtains nearly the optimal anomaly detection performance. It detects $10,720$ true anomalies with only 278 false alarms. Although 37% of anomalies have not been detected, it is caused by the log mining step. Because even when all the log messages are correctly parsed, the log mining model still leaves 34% of anomalies at large. Note that although IPLoM demonstrates the same anomaly detection performance as Drain, their parsing results are different. We also observe that SHISO, although has a high parsing accuracy (0.93), does not perform well in this anomaly detection task. By using SHISO, we would report $1,907$ false alarms, which are 6 times worse than others. This will largely increase the workload of developers, because they usually need to manually check the anomalies reported. Among the online parsing methods, Drain not only has the highest parsing accuracy as demonstrated in Section IV-B, but also obtains nearly optimal performance in the anomaly detection case study.

V. Related Work

Log Analysis for Service Management. Logs, which records system runtime information, are in widespread use for service management tasks, such as business model mining [8], [9], user behavior analysis [10], [11], anomaly detection [3], [4], [26], fault diagnosis [5], [6], performance improvement [7], etc. Log parsing is a critical step to enable automated and effective log analysis [19], because most of these techniques require structured log messages as input. Thus, we believe our proposed online parsing method can benefit these techniques and future studies on log analysis.

Log Parsing. Log parsing has been widely studied in recent years. Xu et al. [3] design a source code based log parser that achieves high accuracy. However, source code is often inaccessible in practice (e.g., Web service components). Some other work proposes data-driven approaches (LKE [4], IPLoM [15], SHISO [17], Spell [16]), in which data mining techniques are employed to extract log templates and split raw log messages into different log groups accordingly. Specifically, LKE and IPLoM are offline log parsers, which are studied in our previous evaluation study on offline log parsers [19]. SHISO and Spell are online log parsers, which parse log messages in a streaming manner, and are not limited by the memory of a single computer. In this paper, we propose an online log parser, namely Drain, that greatly outperforms existing online log parsers in terms of both accuracy and efficiency. It even performs better than the state-of-the-art offline parsers.

Reliability of Web Service Systems. Many recent studies focus on enhancing the reliability of Web service systems. Cubo et al. [27] use dynamic software product lines to reconfigure service failures dynamically. Service selection and recommendation are also widely studied [28], [29]. These studies usually employ QoS (quality of service) values to characterize the reliability of different Web services. Jurca et al. [30] propose a reliable QoS monitoring technique based on client feedback. Yao et al. [31] develop a model with accountability for business and QoS compliance. Besides, Chen et al. [32] propose a performance prediction method for component-based applications. Our proposed online log parser is critical for log analysis techniques, which can complement with these methods in reliability enhancement for Web service systems. The log analysis methods can also improve the reliability of many existing service systems [33], [34], [35].

VI. Conclusion

Log parsing is critical for log analysis based Web service management techniques. This paper proposes an online log parsing method, namely Drain, that parses raw log messages in a streaming manner. Drain adopts a fixed depth parse tree to accelerate the log group search process, which encodes specially designed rules in its tree nodes. To evaluate the effectiveness of Drain, we conduct experiments on five real-world log data sets. The experimental results show that Drain greatly outperforms existing online log parsers in terms of accuracy and efficiency. Drain even obtains better performance than the state-of-the-art offline log parsers, which are limited by the memory of a single computer. Besides, we conduct a case study on a real-world anomaly detection task, which demonstrates the effectiveness of Drain on log analysis tasks.

Acknowledgment

The work described in this paper was supported by the National Natural Science Foundation of China (Project No. 61332010 and 61472338), the National Basic Research Program of China (973 Project No. 2014CB347701), and the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14234416 of the General Research Fund).

References

[1] Amazon ec2. [Online]. Available: https://aws.amazon.com/tw/ec2/
[2] R. Ding, H. Zhou, J. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *ATC'15: Proc. of the USENIX Annual Technical Conference*, 2015.
[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordon, "Detecting large-scale system problems by mining console logs," in *SOSP'09: Proc. of the ACM Symposium on Operating Systems Principles*, 2009.
[4] Q. Fu, J. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM'09: Proc. of International Conference on Data Mining*, 2009.
[5] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an rbf neural network," *TR'12: IEEE Transactions on Reliability*, 2012.
[6] D. Q. Zou, H. Qin, and H. Jin, "Uilog: Improving log-based fault diagnosis by log analysis," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 1038–1052, 2016.
[7] Y. Sun, H. Li, I. G. Councill, J. Huang, W. C. Lee, and C. L. Giles, "Personalized ranking for digital libraries based on log analysis," in *WIDM'08: Proc. of the 10th ACM workshop on Web information and data management*, 2008, pp. 133–140.
[8] H. J. Cheng and A. Kumar, "Process mining on noisy logs-can log sanitization help to improve performance?" *Decision Support Systems*, vol. 79, pp. 138–149, 2015.
[9] H. R. Motahari-Nezhad, R. Saint-Paul, B. Benatallah, and F. Casati, "Deriving protocol models from imperfect service conversation logs," *TKDE'08: IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 12, pp. 1683–1698, 2008.
[10] X. Yu, M. Li, I. Paik, and K. H. Ryu, "Prediction of web user behavior by discovering temporal relational rules from web log data," in *DEXA'12: Proc. of the 23rd International Conference on Database and Expert Systems Applications*, 2012, pp. 31–38.
[11] N. Poggi, V. Muthusamy, D. Carrera, and R. Khalaf, "Business process mining from e-commerce web logs," in *Business Process Management*, 2013, pp. 65–80.
[12] D. Lang, "Using SEC," *USENIX ;login: Magazine*, vol. 38, 2013.
[13] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, pp. 1245–1255, 2013.
[14] W. Xu, "System problem detection by mining console logs," Ph.D. dissertation, University of California, Berkeley, 2010.
[15] A. Makanju, A. Zincir-Heywood, and E. Milios, "A lightweight algorithm for message type extraction in system application logs," *TKDE'12: IEEE Transactions on Knowledge and Data Engineering*, 2012.
[16] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *ICDM'16 Proc. of the 16th International Conference on Data Mining*, 2016.
[17] M. Mizutani, "Incremental mining of system log format," in *SCC'13: Proc. of the 10th International Conference on Services Computing*, 2013.
[18] Drain source code. [Online]. Available: http://appsrv.cse.cuhk.edu.hk/~pjhe/Drain.py
[19] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *DSN'16: Proc. of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.
[20] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *DSN'07*, 2007.
[21] L. A. N. S. LLC. Operational data to support and enable computer science research. [Online]. Available: http://institutes.lanl.gov/data/fdata
[22] C. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.
[23] Evaluation of clustering. [Online]. Available: http://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html
[24] L. Tang, T. Li, and C. Perng, "LogSig: generating system events from raw textual logs," in *CIKM'11: Proc. of ACM International Conference on Information and Knowledge Management*, 2011.
[25] G. Salton and C. Buckley, "Term weighting approaches in automatic text retrival," Cornell, Tech. Rep., 1987.
[26] W. Zhang, F. Bastani, I. L. Yen, K. Hulin, F. Bastani, and L. Khan, "Real-time anomaly detection in streams of execution traces," in *HASE'16: Proc. of the 14th International Symposium on High-Assurance Systems Engineering*, 2012, pp. 32–39.
[27] J. Cubo, N. Gamez, E. Pimentel, and L. Fuentes, "Reconfiguration of service failures in damasco using dynamic software product lines," in *SCC'15: Proc. of the 12nd International Conference on Services Computing*, 2015, pp. 114–121.
[28] S. Y. Hwang, W. P. Liao, and C. H. Lee, "Web services selection in support of reliable web service choreography," in *ICWS'10: Proc. of the 17th International Conference on Web Services*, 2010, pp. 115–122.
[29] S. Meng, Z. Zhou, T. Huang, D. Li, S. Wang, F. Fei, W. Wang, and W. Dou, "A temporal-aware hybrid collaborative recommendation method for cloud service," in *ICWS'16: Proc. of the 23rd International Conference on Web Services*, 2016, pp. 252–259.
[30] R. Jurca, B. Faltings, and W. Binder, "Reliable qos monitoring based on client feedback," in *WWW'07: Proc. of the 16th International Conference on World Wide Web*, 2007, pp. 1003–1012.
[31] J. Yao, S. Chen, C. Wang, D. Levy, and J. Zic, "Modelling collaborative services for business and qos compliance," in *ICWS'11: Proc. of the 18th International Conference on Web Services*, 2011, pp. 299–306.
[32] S. Chen, Y. Liu, I. Gorton, and A. Liu, "Performance prediction of component-based applications," *JSS'05: Journal of Systems and Software*, vol. 74, no. 1, pp. 35–43, 2005.
[33] A. Iwai and M. Aoyama, "Automotive cloud service systems based on service-oriented architecture and its evaluation," in *CLOUD'11: Proc. of the 4th International Conference on Cloud Computing*, 2011.
[34] J. Zhang, B. Iannucci, M. Hennessy, K. Gopal, S. Xiao, S. Kumar, D. Pfeffer, B. Aljedia, Y. Ren, M. Griss, S. Rosenberg, J. Cao, and A. Rowe, "Sensor data as a service–a federated platform for mobile data-centric service development and sharing," in *SCC'13: Proc. of the 10th International Conference on Services Computing*, 2013.
[35] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu, "Everything as a service (xaas) on the cloud: origins, current and future trends," in *CLOUD'15: Proc. of the 8th International Conference on Cloud Computing*, 2015, pp. 621–628.