LogParse: Making Log Parsing Adaptive through Word Classification

Weibin Meng[1, 7], Ying Liu[2, 7], Federico Zaiter[1, 7], Shenglin Zhang[3]✉, Yihao Chen[1, 7], Yuzhe Zhang[3]
Yichen Zhu[4], En Wang[5], Ruizhi Zhang[6], Shimin Tao[6], Dian Yang[6], Rong Zhou[6], Dan Pei[1, 7]
[1]Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
[2]Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China
[3]College of Software, Nankai University, Tianjin 300071, China
[4]Department of Statistis, University of Toronto, Toronto M5S 1A1, Canada
[5]College of Computer Science and Technology, Jilin University, Changchun 130012, China
[6]Huawei, Beijing, 100084, China
[7]Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China
mwb16@mails.tsinghua.edu.cn, liuying@cernet.edu.cn, zaitertf10@mails.tsinghua.edu.cn, zhangsl@nankai.edu.cn
chenyiha17@mails.tsinghua.edu.cn, zyzcs@mail.nankai.edu.cn, k.zhu@mail.utoronto.ca
wangen@jlu.edu.cn, {zhangruizhi, taoshimin, yangdian4, joe.zhourong}@huawei.com, peidan@tsinghua.edu.cn

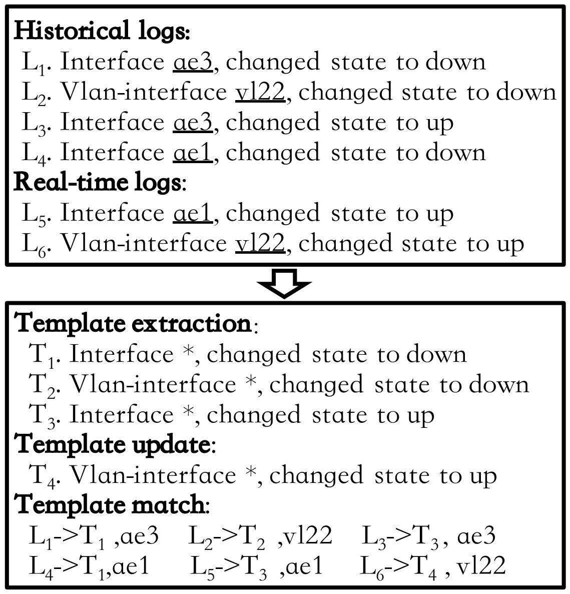
***Abstract*—Logs are one of the most valuable data sources for large-scale service (*e.g.,* social network, search engine) maintenance. Log parsing serves as the the first step towards automated log analysis. However, the current log parsing methods are not adaptive. Without intra-service adaptiveness, log parsing cannot handle software/firmware upgrade because learned templates cannot match new type of logs. In addition, without cross-service adaptiveness, the logs of a new type of service cannot be accurately parsed when this service is newly deployed. We propose LogParse, an adaptive log parsing framework, to support intra-service and cross-service incremental template learning and update. LogParse turns the template generation problem into a word classification problem and learns the features of template words and variable words. We evaluate LogParse on four public production log datasets. The results demonstrate that LogParse supports accurate adaptive template update (increased from 0.559 to nearly 1.0 parsing accuracy), and a trained LogParse is adaptive for a brand new service's log parsing. Because of LogParse's adaptiveness, we also apply LogParse to an interesting application, log compression and deployed log compression in a top cloud service provider. We package LogParse into an open-source toolkit.**

***Index Terms*—Log Analysis; Text Classification; Service Management; AIOps**

I. Introduction

Logs (see top half of Fig. 1), which record a vast range of events of services (*e.g.*, social network, datacenter devices), are one of the most valuable data sources for large-scale services maintenance [1]. Logs have been widely applied for monitoring status [2], [3], understanding events [4], [5], detecting anomalies [6], [7], and predicting failures [8].

A large-scale service is often maintained by abundant operators. For example, a social network application includes lots of softwares and underlying machines. Usually, an operator has incomplete information on the overall social network, and none of them is familiar with all logs generated by this social network. Besides, service logs are usually unstructured texts, they have to be properly parsed before they can be effectively used [9]. An unstructured log is essentially "printf"ed by services and thus follows some specific format: there is a *template* field sketching out the event and summarizing it, and a *variable* field varying from one log to another of the same template. Take Fig. 1 as an example, in L_1, "ae3" is a *variable* word, whereas the rest, *i.e.*, "Interface ..., changed state to up", is the *template* field. There are some methods of document sumarization [10], sentence compression [11], [12] and clustering [13] in NLP domain. However, [14] proves that NLP methods cannot parse logs accurately. To achieve the goal of automated log parsing, many rapid and accurate data-driven template extraction approaches, *e.g.*, Spell [15], LogSig [16]



Historical logs:
L_1. Interface <u>ae3</u>, changed state to down
L_2. Vlan-interface <u>vl22</u>, changed state to down
L_3. Interface <u>ae3</u>, changed state to up
L_4. Interface <u>ae1</u>, changed state to down
Real-time logs:
L_5. Interface <u>ae1</u>, changed state to up
L_6. Vlan-interface <u>vl22</u>, changed state to up

Template extraction:
T_1. Interface *, changed state to down
T_2. Vlan-interface *, changed state to down
T_3. Interface *, changed state to up
Template update:
T_4. Vlan-interface *, changed state to up
Template match:
L_1->T_1 ,ae3 L_2->T_2 ,vl22 L_3->T_3, ae3
L_4->T_1,ae1 L_5->T_3 ,ae1 L_6->T_4 , vl22

Fig. 1. Examples of network device logs and their templates

✉ Shenglin Zhang is the corresponding author.

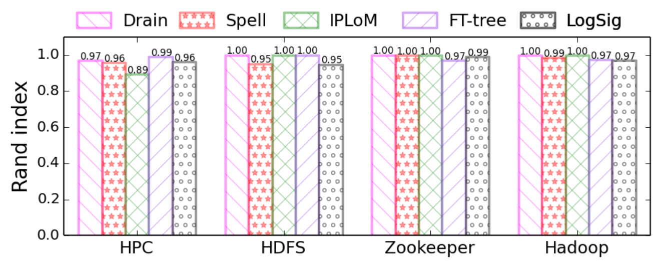


Fig. 2. Accuracy of traditional template extraction results when all the logs are used for training

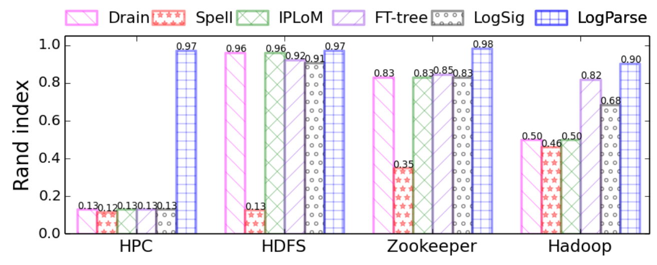


Fig. 3. Accuracy of traditional template extraction methods compared to LogParse when only 10% of logs are used for training

and IPLoM [17] have been proposed. However, the current log parsing methods are not *adaptive*. This adaptiveness consists of two aspects, *i.e.*, intra-service adaptiveness and cross-service adaptiveness.

For intra-service, operators continuously conduct software/firmware upgrades on services and underlying machines to introduce new features, fix bugs or improve performance (*e.g,* a social network service updates its applications), which can generate new types of logs [18]. These logs, however, cannot match any existing templates, and thus new templates have to be learned, which is called template update. To demonstrate services do generate many new types of logs at runtime, we set apart 10% of logs from four public log datasets (described in Section IV-A) and extract log templates using five template extraction methods. Then, we use these log templates to match the remaining 90% of logs. Table I shows the ratio of logs without corresponding existing templates in the remaining 90% of logs. The template set should be incrementally learned, otherwise, a new template set has to be learned from *all* the logs. Considering the large volume of the long-period historical logs, this is a vast amount of work. In Fig. 1, for example, the service generates two new logs (L_5 and L_6). We have to update the template set because L_6 cannot match any existing template.

Cross-service adaptiveness means that a model trained by service A is also suitable for service B. Log parsing without cross-service adaptiveness makes it labor intensive and time consuming to build and maintain a service type specific template set. In real-world service providers, different types of services are different in log syntax/semantics. For example, in the same social network service, “add nodes” and “delete nodes” functions generate different types of logs and have different template sets. As a result, current log extraction methods typically extract templates and build a dedicated template set for each service type. When a brand new service goes online, there are usually not enough historical logs to train accurate templates. For example, Fig. 2 and Fig. 3 show the accuracy of template extraction results when all logs and only 10% of logs are used for training respectively (detailed information is shown in Section IV). We observe that current template extraction methods achieve terrible accuracy when trained on a smaller sample of the logs. Nonetheless, as seen in Fig. 2, current log parsing methods are accurate enough for historical logs.

Thus, log parsing methods without adaptiveness significantly limit many log analysis applications, because many applications require to have a corresponding template for any given logs. An example of this is Log compression, which is a useful application of log analysis. For instance, Google and Facebook respectively generate 100 Petabyte and 10 Petabyte of log data per month [19]. These massive logs, if not properly compressed, will consume immense storage space. However, traditional compression methods cannot compress and query specific logs in real-time[20]. Fortunately, unstructured logs contain many redundant information. We can turn logs into “template index + variables” (*e.g.,* “T_1 + ae3” in Fig. 1), which saves substantial storage. Log compression based on templates also require templates for any given logs. In other words, it needs an adaptive log parsing method.

The key intuition is based on the following observations: The difference between template words (*e.g.,* “Interface” in Fig. 1) and variable words (*e.g.,* “ae3”) are obvious, both within the same service and among different services.

We *turn the template generation problem into a word classification problem* to determine whether a word belongs to template words. Then, we get word labels by using *any* existing traditional template extraction methods. Next, we represent *any* word by using vectors and learn the features of template words and variable words. Finally, a new template can be generated by combining template words from a new type of log. We face the following three challenges.

1. There is a massive number of words in logs, and many of these words may appear only once. Representing and classifying all words accurately is a challenging task.

2. Some words may be present in both template and variable fields, we need to classify them based on their context.

3. We have to get labels for training a word classifier, however, there are not labels for each word in raw logs.

To address the above challenges, we propose LogParse, an adaptive log parsing framework. The contributions of LogParse are as follows.

- LogParse learns new templates in an incremental manner, and thus only new types of logs generated after software/hardware upgrades have to be matched to the updated template set, which greatly reduces the amount of work comparing with rematching all the historical logs to templates. LogParse improves parsing accuracy

TABLE I
RATIOS OF UNMATCHED LOGS IN REMAINING 90% OF LOGS

Method	HPC	HDFS	Zookeeper	Hadoop
LogSig	0.953	0.478	0.222	0.699
IPLoM	0.025	3e-05	0.199	0.265
Spell	0.940	0.832	0.876	0.992
Drain	0.940	3e-05	0.198	0.173
FT-tree	0.933	7e-05	0.596	0.366

from 0.559 to nearly 1.0 when it updates templates incrementally.

- Since LogParse is cross-service adaptive, it is general to diverse types of services/machines. This way, we build and maintain only one template set for all types of services/machines, which saves a lot of time and resources.
- LogParse is able to assign/generate templates for any given logs, so that we can achieve novel log analysis applications by adopting LogParse. For example, LogParse can be easily applied for log compression.
- We have open-sourced[1] LogParse, and hope that it can be used for future research.

The rest of the paper is organized as follows: We discuss related works in Section II and propose our approach in Section III. The evaluation is shown in Section IV. In Section V, we introduce LogParse's applications. Finally, we conclude our work in Section VI.

II. Related Work

Service logs play an important role in service management. Log parsing serves as the the first step towards automated log analysis. To achieve the goal of automated template extraction, many data-driven approaches have been proposed. There are many categories of template extraction methods [9]. The first category is cluster-based methods, which log template forms a natural pattern of a group of log messages. From this view, log parsing can be modeled as a clustering problem, such as LogSig [16]. Next is longest common subsequence. For example, Spell [15] uses the longest common subsequence algorithm to parse logs in a stream. Iterative partitioning is used in IPLoM [17]. Some methods use heuristics to extract templates. As opposed to general text data, log messages have some unique characteristics. Consequently, Drain [21] propose heuristics-based log parsing methods. The final category is frequent items mining. Log templates can be seen as a set of constant tokens that occur frequently in logs, such as FT-tree [22].

However, for the intra-service scenario, most methods only focus on template extraction while the other two parts of log parsing, *i.e.*, template matching and update, which are of vital importance for log analysis, are not being properly combined to allow for adaptiveness.

[1]LogParse is available on Github: https://github.com/WeibinMeng/LogParse

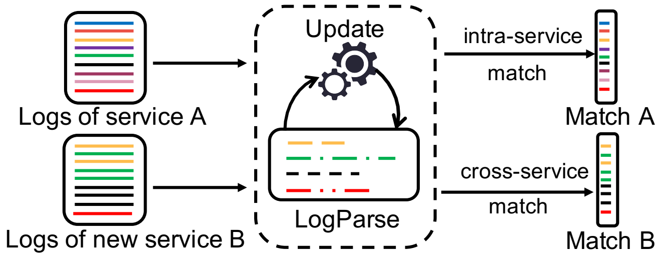


Fig. 4. Adaptiveness of LogParse. We train LogParse only by using logs of service A. Intra-service adaptiveness means that it matches new types of logs generated by service A. Cross-service adaptiveness means we use it to match real-time logs from a different service B.

Although FT-tree [22] and Drain [21] support incremental learning, their performance in such task is bad (shown in Fig. 3). What's more, there is no template extraction method with cross-service adaptiveness.

An adaptive log parsing method will improve the performance of many applications of log analysis (*e.g.*, log compression, anomaly detection, failure prediction base on unstructured logs). For example, PreFix [8], a failure prediction system based on switch logs, cannot handle new types of logs. DeepLog [7] addressed new templates learning by obtaining operators' feedback during anomaly detection. LogAnomaly [6] is a state-of-the-art method to detect anomalies based on unstructured logs. However, it deals with new types of logs by merging them into the most similar existing templates.

III. Design

A. Overview

The objective of LogParse is to be adaptive in order to support intra-service incremental learning and cross-service learning. LogParse can incrementally update the template set and thus new types of logs can be matched to the updated template set, without having to relearn the whole template set and rematch all the historical logs. In addition, LogParse can build the template set of a service based on the templates of another service, which can address the problem that previous log extraction methods cannot accurately learn templates for a newly deployed service because of insufficient amount of logs.

Fig. 4 shows adaptiveness of LogParse. For intra-service adaptiveness, we get word labels by using *any* existing traditional template extraction methods and train LogParse model in step#1 of Fig. 4. Then, we utilize the trained LogParse to match online logs of service A. If failed, it will learn a new template and incrementally update the template set. For cross-service adaptiveness, we use LogParse trained on service A directly, and then, match online logs of service B.

We show the detailed design of LogParse in Fig. 5. In the offline component, LogParse first extracts templates from historical logs (Section III-B). Then, it distinguishes template words from variable words based on templates learned from historical logs. Regarding these word classes as labels for

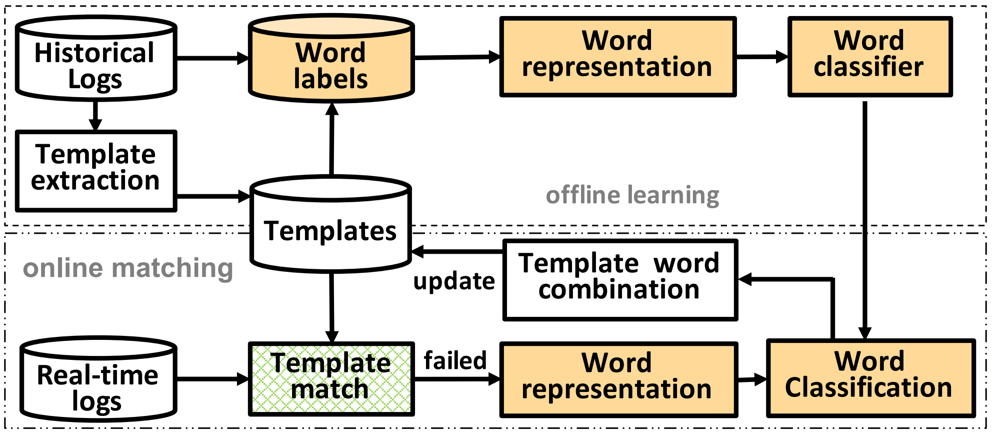


Fig. 5. Detailed design of LogParse

training the word classifier, it then classifies words into template or variable words using a binary classifier (Section III-D). When real-time logs are generated, LogParse matches them to templates (Section III-C). If a real-time log cannot match any existing template, LogParse constructs vectors for each word of this log, distinguishes between template words and variable words based on the trained word classifier, learns a new template, and adds it to the template set (Section III-D).

B. Template Extraction

Template extraction has been widely studied in previous works [9], [22], and it is not the main contribution of this work. Its objective is to learn templates from logs, based on the observation that a log is "printf"ed by services, and usually follows predefined structures. For example, the template of L_1 in Fig. 1, *i.e.,* "Interface ae3, changed state to down", is T_1, *i.e.,* "Interface *, changed state to down".

As shown in Fig. 2, *any* existing template extraction methods can accurately learn templates from long-period historical logs. Therefore, *we can use the template set automatically learned by these methods to distinguish between template words and variable words.*

C. Template Matching

After template extraction, we get a template set from historical logs. Motivated by the extremely efficient performance of the prefix-tree structure in packet forwarding [23], we build a prefix-tree for the template set to accelerate template matching. For example, Fig. 6 shows a prefix-tree for the logs in Fig. 1, where the templates with solid line boxes are existing templates. It is constructed as follows.

- We build a null root of a prefix-tree.
- We scan each existing template and sort a word list L by the order of each word's appearance in the template.
- We insert L into the prefix-tree, where each node is a word. If two templates share a common prefix (*e.g.*, "Interface, changed, state, to" in T_1 and T_3)), we only create a new subtree for different words.
- Each root-to-leaf path of the prefix-tree is a template.

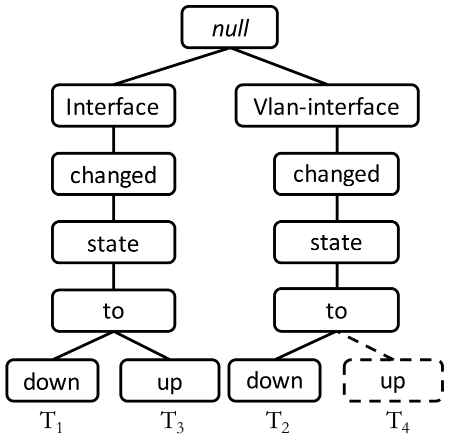


Fig. 6. The prefix tree constructed for the template set in Fig. 1

For example, "Interface, changed, state, to, down" in Fig. 6 is a template. Note that we can incrementally add new templates to this prefix-tree in case that new templates are learned from new types of logs. In this way, a log can be represented with its corresponding template index and variable words. For example, L_1 in in Fig. 1 can be represented with its template index T_1 and variable word "$ae3$".

A real-time log can match existing template following Algorithm 1. Firstly, we get the word list of the new log. Then, according to the order of word list, we find the word nodes from template tree (if they are in template tree). Finally, if we find a root-to-leaf path in existing template tree, the path is the matched template, otherwise, template search algorithm 1 will return "NULL" and we need to generate a new template for the new log (describe in Section III-D).

D. Template Generation

Learning a template from a new type of log equals to classifying the words in the log into template words and variable words. In Fig. 1, for example, "$Interface$, $changed$" are template words, and "$ae3$, $vlan22$" are variable words.

Algorithm 1 Template Search

```
Input: A template set trie $TST$ of all extracted log templates, and a sequence $LWS$ containing each word from a given log in order
Output: A matched template $MT$
1: Let a state $S$ record a matching state up to a given node of $TST$ with the pair $RMS$ being the remanent sequence of $LWS$ as each of its words are processed and $MN$ being the node from $TST$ yet to be matched
2: Set the root of $TST$, and the whole $LWS$ as the initial state of $S$
3: Create a queue $Q$ and enqueue $CS$
4: while $Q$ is not empty do
5:     Dequeue a state from $Q$ as the current state $CS$
6:     if $CS$ is valid and $MN$ is a leaf node of $TST$ then
7:         Let the traceback of $MN$ up to the root of $TST$ be matched template $MT$
8:         return $MT$
9:     end if
10:    for each child $C$ of $MN$ do
11:        if $C$ in $RMS$ then
12:            Create a new current state $NCS$ with the pair of the updated $RMS$ by removing its first word and $C$
13:            Enqueue $NCS$ in $Q$
14:        end if
15:    end for
16: end while
17: return NULL {No existing template was matched}
```

In this way, we transform the template update problem into a word classification problem. However, word classification faces three challenges: (1) Training a word classifier needs a large amount of labels. (2) New words can appear in new templates. (3) A variable word can appear in a template sometimes.

The template update of LogParse, as shown in the orange modules of Fig. 5, is proposed to address the above three challenges. It includes four steps, *i.e.,* word labeling, word representation, word classifier, and new template generation, as follows. As aforementioned, existing template extraction methods cannot update templates at runtime. In this paper, we propose a novel log parse framework, LogParse, which can be combined with all existing template extraction methods and enable them to update templates. The update procedures of LogParse are shown in Fig. 5 (orange modules).

1) Word Labeling: There are tens of thousands of templates in large service providers, and thus manually classifying template words and variable words so as to get labels for training a word classifier is nearly infeasible. To address this problem, LogParse distinguishes template words from variable words by combining historical logs and the template set automatically constructed by existing template extraction methods. Based on extensive investigations, we find that the template set learned by these methods are accurate enough

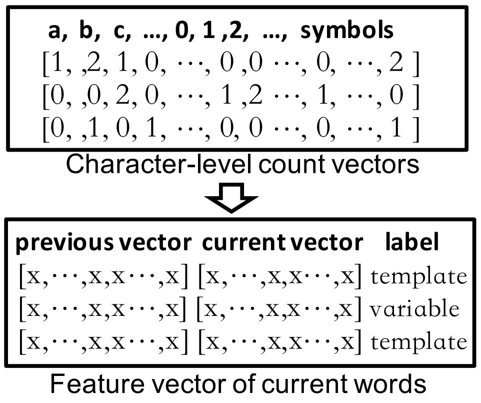

a, b, c, …, 0, 1 ,2, …, symbols
[1, ,2, 1, 0, ⋯, 0 ,0 ⋯, 0, ⋯, 2]
[0, ,0, 2, 0, ⋯, 1 ,2 ⋯, 1, ⋯, 0]
[0, ,1, 0, 1, ⋯, 0, 0 ⋯, 0, ⋯, 1]
Character-level count vectors
previous vector current vector label
[x,⋯,x,x⋯,x] [x,⋯,x,x⋯,x] template
[x,⋯,x,x⋯,x] [x,⋯,x,x⋯,x] variable
[x,⋯,x,x⋯,x] [x,⋯,x,x⋯,x] template
Feature vector of current words

Fig. 7. Example of feature vector

(with an average accuracy of 0.978), as shown in Fig. 2. In this way, LogParse automatically gets the labels for template words and variable words without any manual labelling, and thus solves the first challenge.

2) Word Representation: Most of approaches treat word representations as the cornerstone in natural language processing. Though it is effective, word-level representation (e.g., word embedding [24]) is inherently problematic: it assumes that each word type has its own vector that can vary independently [25]. However, new types of logs introduced by software/firmware upgrades aiming to add new features or fix previous bugs result in new templates being generated online. What's more, most words only occur once in logs and out-of-vocabulary (OOV) new words could not be addressed online using a word-level representation.

To address this problem, we represent each word in a log using a character-level count vector, as shown in Fig. 7. We assume that words which share common components (prefix, symbols, numbers) may be potentially related.

Specifically, for a given word, we construct a vector based on the number of each character in this word. The character-level count vectors with fixed dimensionality can represent any word because the set of different characters is fixed (*e.g.,* there are 128 different characters in ASCII). In this way, we address the second challenge. In addition, we can find that some variable words can also appear in templates. Therefore, classifying a word into a template word or a variable word just based on the word itself is not accurate, so the context of the word should be considered for its classification. Inspired by the n-gram concept, we classify a word by combining it with its previous word. For a given word, as shown in Fig. 7, we concat the count vectors of this and its previous words to construct its feature vector. Consequently, the third challenge is addressed.

3) Word Classifier: As mentioned above, LogParse transforms the log template update problem into a word classification problem. For each word, it constructs a vector by combining the character-level count vector of this word and its previous one. We utilize SVM [26], a popular machine learning method, to train a binary classifier. Note that, other machine learning methods can also be used to train the binary classifier. Applying SVM is not one of our contributions.

4) New Template Generation: In the online component, we search the prefix-tree constructed from the template set and match real-time logs to existing templates following Algorithm 1. If the new log matches no existing template, for each word in this log, we construct a feature vector by combining the character-level count vector of this and its previous word. Then, we classify each word by the trained word classifier. Finally, we construct a template based on the template words. In Fig. 1, for example, L_6 cannot match any existing template. According to the word classifier, LogParse finds that the words "Vlan-interface", "changed", "state", "to", "up" are template words and thus inserts them into the prefix-tree to update the template set (T_4 in Fig. 6).

E. LogParse's Application in Log Compression

As for log compression, traditional compression tools cannot decompress specific logs in real-time because they need to decompress whole file chunks [20]. Considering that when a failure occurs, operators tend to traceback the logs of similar failures for quick mitigation, real-time querying is an essential requirement for the operators.

Usually, the number of different templates are much smaller than that of different logs, and in a log the number of variable words are much smaller than that of all the words. Therefore the unstructured logs contain a lot of redundant information. A log can be easily compressed (matched) to, and quickly decompressed (recovered) from, its template and variables. In order to (de)compress every log, the template set should be kept up to date. Consequently, log parsing, which extracts templates from logs, matches logs to templates, and maintains a template set continuously, is a promising direction to efficiently and effectively compress logs.

In this paper, we apply LogParse on two storage scenarios: *short-term storage* and *long-term storage*. For short-term storage (*e.g.,* within one year), operators have to query logs in real-time. Traditional compression methods cannot decompress for given logs in real-time. They are only suitable to long-term storage. As is shown in Fig. 1, we utilize log templates and LogParse to parse real-time logs and save "template index + variables" to compress original logs. For long-term storage, we adopt double compression to process logs. That is, use traditional compression methods to compress "template index + variables" sequence, which will achieve the highest compression ratio.

IV. Evaluation of LogParse

A. Experiment Setting

In this section, we evaluate the performance of LogParse in supporting intra-service adaptiveness and cross-service adaptiveness. The datasets, template extraction methods, evaluation metrics and experimental setup of the experiments are as follows.

TABLE II
Detail of the datasets

Datasets	Description	# of logs
HPC	High performance cluster	433,489
HDFS	Hadoop distributed file system	11,175,629
ZooKeeper	ZooKeeper service	74,380
Hadoop	Hadoop MapReduce job	394,308

1) Datasets: We conduct experiments over four public log datasets from distributed systems, which are HPC [27], HDFS [28], ZooKeeper [27], and Hadoop [29]. The detailed information of these datasets is listed in Table II. For each dataset, [9] sampled logs and manually labelled each log's template, which serves as the ground truth for our evaluation.

2) Template Extraction Methods: As aforementioned, LogParse can incorporate any existing template extraction method, so template extraction (in the offline component) is not the main contribution of this work. To demonstrate the performance of LogParse regardless of the method used, we have implemented five template extraction methods: FT-tree [22], Drain [21], Spell [15], LogSig [16] and IPLoM [17] (see Section II for more details). The parameters of these methods are all set best for accuracy.

3) Evaluation Metrics: We apply $Rand\ index$ [30] to quantitatively evaluate the accuracy of template extraction. $Rand\ index$ is a popular method for evaluating the similarity between two data clustering techniques or multi-class classifications. What's more, $Rand\ index$ is applied to evaluating existing template extraction methods in the literature, such as in [22], [31]. For each template extraction method, we evaluate its accuracy by calculating the $Rand\ index$ between the manual classification results and the templates learned by it. Specifically, among the template learning results of a specific method, we randomly select two logs, *i.e.,* x and y, and define TP, TN, FP, FN as follows. TP: x and y are manually classified into the same cluster and they have the same template; TN: x and y are manually classified into different clusters and they have different templates; FP: x and y are manually classified into different clusters and they have the same template; FN: x and y are manually classified into the same cluster and they have different templates. Then $Rand\ index$ can be calculated using the above terms as follows: $Rand\ index = \frac{TP+TN}{TP+TN+FP+FN}$.

4) Experimental Setup: We conduct experiments on a Linux server with Intel Xeon 2.40 GHz CPU and 64G memory. We implement LogParse with Python 3.6 and have open-sourced LogParse on github[2].

B. Evaluation on Incremental Learning

LogParse can incorporate any existing template extraction method to learn templates from logs. It generates word representations for each word in the logs, and together with their

[2]LogParse is available on Github: https://github.com/WeibinMeng/LogParse

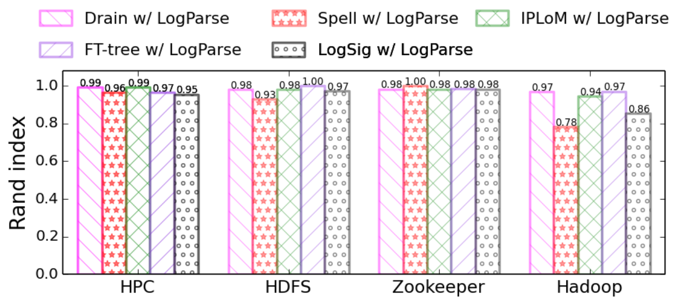


Fig. 8. Accuracy of template extraction results with (w/) LogParse when 10% logs are used for training

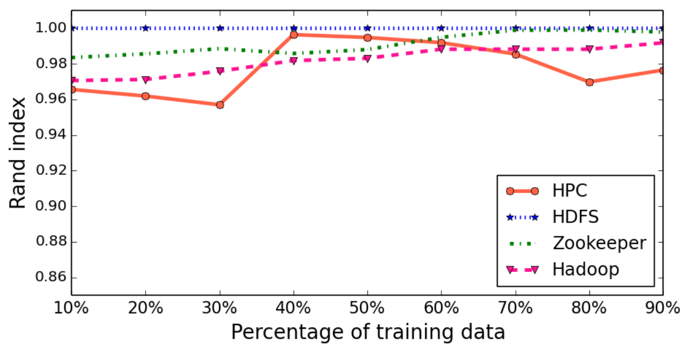


Fig. 9. The template extraction accuracy of LogParse as the percentage of training data changes

labels learned by the template extraction method, it then trains a word classifier. We find the accuracy of the word classifier heavily relies on that of template extraction. Fig. 2 shows the accuracy (in terms of $Rand\ index$) of Drain, Spell, IPLoM, FT-tree and LogSig on the datasets of HPC, HDFS, Zookeeper, Hadoop, respectively. For each method, it learns templates from each dataset and then matches each log in the dataset to a template. The existing template extraction methods are highly accurate in learning templates from logs. Specifically, they achieve an average accuracy of 97.80%. Consequently, we can directly use the automatic template extraction results of these methods to build word representations and train a word classifier.

As aforementioned, LogParse can incrementally update its template set in case that new types of logs are generated because of software/firmware upgrade. This is important because otherwise new types of logs cannot match any existing templates, and the long period (large volume) of historical logs have to rematch the new template set, which costs tremendous amounts of time and resources.

To demonstrate the performance of LogParse in supporting incremental learning and simulate the launch of new services, for each dataset, we apply each template extraction method to learn templates from 10% of their logs. Based on these templates, LogParse distinguishes template words and variable words, builds word representations, and trains a word classifier (see Fig. 5 for more details). After that, for the remaining 90% logs, LogParse first matches them to the existing templates, and if failed, it then builds their word representations, updates templates based on the trained word classifier, and matches them to the updated template set. If we only utilized template extracting methods without update, as shown in Fig. 3, the average template extraction accuracy is 0.559. However, Fig. 3 also shows that, when LogParse is applied to update templates, the template extraction accuracy on the remaining 90% logs of each dataset is relatively high. Note that, the accuracy of LogParse in Fig. 3 is the average accuracy obtained on each dataset as it is shown in Fig. 8. Clearly, the average template extraction accuracy of each method on each dataset is 0.958. In other words, LogParse improves the template extraction accuracy by 71.5% when new types of logs are generated.

To demonstrate how robust LogParse is to the scale of training data, Fig. 9 shows the template extraction accuracy (in terms of $Rand\ index$) of LogParse on the four datasets, as the percentage of training data increases from 10% to 90%, respectively. Henceforth, LogParse incorporates FT-tree as the log extraction method since FT-tree is robust to diverse types of logs. With LogParse automatically and continuously updating templates for new types of logs, the accuracies of template extraction are quite stable as the percentage of training data changes. In other words, LogParse is robust to different scales of training logs, and can achieve high template extraction accuracy when trained based on a small scale of training data.

C. *Evaluation on Supporting Cross-service Adaptiveness*

LogParse is cross-service adaptive, which enables it to accurately learn templates from logs when a new type of service is newly deployed. Based on these templates, an anomaly detection/prediction method can be trained, and the logs can be compressed, soon after the new type of service is deployed. Since different datasets are usually different in syntax, we conduct a cross-dataset experiment to demonstrate the performance of LogParse in the cross-syntax scenario. We therefore train LogParse based on the logs of one dataset (as shown in the upper part of Fig. 5) and match the logs of another dataset (as shown in the lower part of Fig. 5) based on the trained model. Table III shows the accuracy (in terms of $Rand\ index$) of LogParse for the cross-dataset template learning and matching. LogParse achieves close-to-one $Rand\ index$ for all the cases. For example, when LogParse is trained based on the HPC dataset and applied to match the logs of the HDFS dataset, the $Rand\ index$ is 0.983. On average, LogParse achieves a cross-dataset accuracy of 0.980, which strongly demonstrates that LogParse is cross-service adaptive.

V. Application in Log Compression

As aforementioned, LogParse, which is cross-service adaptive and supports incremental learning, can be used to compress logs and quickly query (decompress) specific logs after compression.

TABLE III
THE ACCURACY OF LOGPARSE IN CROSS-DATASET TEMPLATE LEARNING AND MATCHING

Training data	Testing data			
	HPC	HDFS	ZooKeeper	Hadoop
HPC	-	0.983	0.999	0.923
HDFS	0.982	-	0.993	0.974
Zookeeper	0.993	1.0	-	0.937
Hadoop	0.983	0.999	0.999	-

A log can be represented by its template and variable words. Since typically many logs share the same template, this representation can greatly save storage space, and thus effectively compress logs. Then the log can be quickly recovered from this representation. In this section, we show the performance of LogParse in log (de)compression, with the public HPC, HDFS, Zookeeper, and Hadoop datasets, and the logs collected from a top-tier global cloud service provider. We compare LogParse with three commonly used compression methods, *i.e.,* bzip [32], 7zip [33], zip [34]. In addition, we apply the compression ratio, *i.e.,* $\frac{size\ of\ compressed\ logs}{size\ of\ original\ logs}$, to evaluate the performance of each log compression method. Note that, accuracy of log parsing doesn't impact the log compression. This is because even if some templates are wrong, as variables are also saved together with template indexes, we can recover all compressed logs completely.

A. Evaluation on Public Datasets

Table IV shows the comparison ratios of LogParse-enabled compression approach, bzip, 7zip, zip, and their combinations. In LogParse-enabled compression, we simply apply LogParse to compress template components, which can be quickly decompressed. Then, we combine LogParse and traditional compression approaches to achieve double compression. The LogParse-enabled compression approach achieves an average compression ratio of 22.7%. Since logs in the HPC and Hadoop datasets usually have more variables, which are not compressed in LogParse-enabled compression (using "template + variable words" to represent a log), it has bad compression ratios on these datasets. After a long time, historical logs become less likely to be queried, and thus we can store them by combining LogParse-enabled compression approach with the commonly used compression methods. For example, when we combine it with bzip, 7zip, or zip, the combination achieves much smaller compression ratios than these commonly used methods, as shown in Table IV.

B. Deployment Experience

Considering the superior performance of LogParse in log compression, we have deployed it in a top tier cloud service provider to effectively parse and compress logs for network devices and firewalls. With LogParse, the service provider has represented (compressed) logs using the format of "template + variable words" since September 2018. The compression ratios of LogParse for switch and firewall logs are 16.4% and 23.3%, respectively. If LogParse is combined with bzip, the above compression ratios drop to 3.13% and 5.07%, respectively.

TABLE IV
THE COMPRESSION RATIOS OF LOGPARSE-ENABLED COMPRESSION APPROACH, BZIP, 7ZIP, ZIP AND THEIR COMBINATIONS

Method	HPC	HDFS	ZooKeeper	Hadoop	Average
LogParse	23.4%	14.1%	10.9%	42.4%	22.7%
bzip	5.7%	9.3%	3.0%	5.6%	6.1%
LogParse w/ bzip	3.4%	3.2%	2.4%	3.6%	3.2%
7zip	7.0%	8.5%	3.1%	5.0%	5.9%
LogParse w/ 7zip	3.7%	6.3%	2.8%	3.4%	4.1%
zip	8.6%	11.4%	4.8%	8.0%	8.2%
LogParse w/ zip	4.4%	7.8%	2.8%	5.0%	5.0%

TABLE V
COMPARISON OF THE QUERY TIME FOR ONE HUNDRED LOGS FROM THE DATASET COMPRESSED FROM 1 TB LOGS

Method	LogParse	bzip	7zip	zip
Time	0.212 ms	20.27 hours	1.12 hours	1.91 hours

This service provider usually compresses the logs every day, which we believe, is not rare for other cloud service providers. In addition, the provider generates TBs of logs every day, and operators usually have to query hundreds of logs to traceback a similar historical failure when a failure occurs. Therefore, we compare the time needed for querying 100 specific logs from a dataset compressed from 1TB of logs. Table V shows the comparison results. LogParse dramatically speeds up the decompression and decreases the time from hours to 0.212 ms.

VI. CONCLUSION AND FUTURE WORK

Logs play an important role in service maintenance and log parsing which is the first step of automated log analysis. However, current log parsing methods are not adaptive. We propose a novel log parsing framework, LogParse, to parse logs adaptively both in an intra-service and a cross-service manner. Our evaluation on four public production log datasets demonstrates LogParse improves parsing accuracy from 0.559 to nearly 1.0 when new types of logs are generated. A trained LogParse is also adaptive for a brand new service's log parsing. We have open-sourced LogParse, and hope that it can be used for future research. Besides, we demonstrate the power of LogParse when applied to log compression, enabling real-time querying of compressed logs. Other interesting log analysis applications (*e.g.,* log classification, log-based anomaly detection, log-based failure prediction, log summary) with similar requirements, where a template should always be assigned for a given log, could now be enabled using LogParse. We will utilize LogParse to improve more log analysis applications in the future.

VII. Acknowledgment

The work was supported by National Key R&D Program of China (Grant No. 2018YFB1800405), the National Natural Science Foundation of China (Grant Nos. 61772307, 61902200, 61772230, 61972450 and 61402257), the China Postdoctoral Science Foundation (Nos. 2019M651015, 2017M611322 and 2018T110247) and the Beijing National Research Center for Information Science and Technology (BNRist). We thank the anonymous reviewers for their valuable feedback. We thank Yixuan Zhang, Ya Su, Tianke Zhang for their helpful suggestions.

References

[1] Siddhartha Satpathi, Supratim Deb, R Srikant, and He Yan. Learning latent events from network message logs: A decomposition based approach. *arXiv preprint arXiv:1804.03346*, 2018.

[2] Tao Li, Yexi Jiang, Chunqiu Zeng, Bin Xia, Zheng Liu, Wubai Zhou, Xiaolong Zhu, Wentao Wang, Liang Zhang, Jun Wu, et al. Flap: An end-to-end event log analysis platform for system management. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1547–1556. ACM, 2017.

[3] Subhendu Khatuya, Niloy Ganguly, Jayanta Basak, Madhumita Bharde, and Bivas Mitra. Adele: Anomaly detection from event log empiricism. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018.

[4] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 60–70. ACM, 2018.

[5] Tatsuaki Kimura et al. Spatio-temporal factorization of log data for understanding network events. In *INFOCOM, 2014 Proceedings IEEE*, pages 610–618. IEEE, 2014.

[6] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. International Joint Conferences on Artificial Intelligence Organization*, volume 7, pages 4739–4745, 2019.

[7] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.

[8] Shenglin Zhang, Ying Liu, Weibin Meng, Zhiling Luo, Jiahao Bu, Sen Yang, Peixian Liang, Dan Pei, Jun Xu, Yuzhi Zhang, et al. Prefix: Switch failure prediction in datacenter networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems(SIGMETRICS)*, 2(1):2, 2018.

[9] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *Proceedings of the 41st International Conference on Software Engineering(ICSE)*, pages 121–130, 2019.

[10] Qiang Jipeng, Qian Zhenyu, Li Yun, Yuan Yunhao, and Wu Xindong. Short text topic modeling techniques, applications, and performance: A survey. *arXiv preprint arXiv:1904.07695*, 2019.

[11] Liangguo Wang, Jing Jiang, Hai Leong Chieu, Chen Hui Ong, Dandan Song, and Lejian Liao. Can syntax help? improving an lstm-based sentence compression model for new domains. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1385–1393, 2017.

[12] Thibault Fevry and Jason Phang. Unsupervised sentence compression using denoising auto-encoders. *arXiv preprint arXiv:1809.02669*, 2018.

[13] Jianhua Yin, Daren Chao, Zhongkun Liu, Wei Zhang, Xiaohui Yu, and Jianyong Wang. Model-based clustering of short text streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2634–2642. ACM, 2018.

[14] Weibin Meng, Ying Liu, Shenglin Zhang, Dan Pei, Hui Dong, Lei Song, and Xulong Luo. Device-agnostic log anomaly classification with partial labels. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–6. IEEE, 2018.

[15] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864. IEEE, 2016.

[16] 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*, pages 785–794. ACM, 2011.

[17] 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*, pages 1255–1264. ACM, 2009.

[18] Shenglin Zhang, Ying Liu, Dan Pei, Yu Chen, Xianping Qu, Shimin Tao, Zhi Zang, Xiaowei Jing, and Mei Feng. Funnel: Assessing software changes in web-based services. *IEEE Transactions on Services Computing*, 11(1):34–48, 2018.

[19] Gunasekaran Manogaran, Chandu Thota, and Daphne Lopez. Human-computer interaction with big data analytics. In *HCI challenges and privacy preservation in big data security*, pages 1–22. IGI Global, 2018.

[20] Danny Harnik, Ronen Kat, Dmitry Sotnikov, Avishay Traeger, et al. To zip or not to zip: Effective resource usage for real-time compression. In *USENIX FAST)*, pages 229–241, 2013.

[21] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

[22] Shenglin Zhang, Weibin Meng, Jiahao Bu, Sen Yang, Ying Liu, Dan Pei, Jun Xu, Yu Chen, Hui Dong, Xianping Qu, et al. Syslog processing for switch failure diagnosis and prediction in datacenter networks. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2017.

[23] Keith Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, volume 1991, pages 93–99, 1991.

[24] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185, 2014.

[25] Dongyun Liang, Weiran Xu, and Yinge Zhao. Combining word-level and character-level representations for relation classification of informal text. In *Proceedings of the 2nd Workshop on Representation Learning for NLP*, pages 43–47, 2017.

[26] Thorsten Joachims. Making large-scale svm learning practical. Technical report, Technical report, SFB 475: Komplexitätsreduktion in Multivariaten, 1998.

[27] Shilin He, Jieming Zhu, et al. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[28] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Largescale system problem detection by mining console logs. *Proceedings of SOSP'09*, 2009.

[29] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, pages 102–111. ACM, 2016.

[30] William M Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[31] Shenglin Zhang, Ying Liu, Weibin Meng, Jiahao Bu, Sen Yang, Yongqian Sun, Dan Pei, Jun Xu, Yuzhi Zhang, Lei Song, and Ming Zhang. Efficient and robust syslog parsing for network devices in datacenter networks. *IEEE Access*, 8:30245–30261, 2020.

[32] Julian Seward. b-zip. http://www.b-zip2.org/, 1996.

[33] Igor Pavlov. 7-zip. http://www.7-zip.org/, 1999.

[34] Phil Katz. zip. http://infozip.sourceforge.net, 1989.