Zhu YQ, Deng JY, Pu JC *et al.* ML-Parser: An efficient and accurate online log parser. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(6): 1412–1426 Nov. 2022. DOI 10.1007/s11390-021-0730-4

ML-Parser: An Efficient and Accurate Online Log Parser

Yu-Qian Zhu (朱玉倩), Jia-Ying Deng (邓佳颖), Jia-Chen Pu (蒲嘉宸)
Peng Wang* (王　鹏), *Member, CCF, IEEE*, Shen Liang (梁　燊), and Wei Wang (汪　卫), *Member, CCF*

School of Computer Science, Fudan University, Shanghai 200082, China

E-mail: {19212010018, 19212010083, 17212010028, pengwang5, sliang11, weiwang1}@fudan.edu.cn

Received June 21, 2020; accepted September 18, 2021.

Abstract A log is a text message that is generated in various services, frameworks, and programs. The majority of log data mining tasks rely on log parsing as the first step, which transforms raw logs into formatted log templates. Existing log parsing approaches often fail to effectively handle the trade-off between parsing quality and performance. In view of this, in this paper, we present Multi-Layer Parser (ML-Parser), an online log parser that runs in a streaming manner. Specifically, we present a multi-layer structure in log parsing to strike a balance between efficiency and effectiveness. Coarse-grained tokenization and a fast similarity measure are applied for efficiency while fine-grained tokenization and an accurate similarity measure are used for effectiveness. In experiments, we compare ML-Parser with two existing online log parsing approaches, Drain and Spell, on ten real-world datasets, five labeled and five unlabeled. On the five labeled datasets, we use the proportion of correctly parsed logs to measure the accuracy, and ML-Parser achieves the highest accuracy on four datasets. On the whole ten datasets, we use Loss metric to measure the parsing quality. ML-Parse achieves the highest quality on seven out of the ten datasets while maintaining relatively high efficiency.

Keywords log parsing, online approach, structure extraction, similarity measure

1 Introduction

A log is a text message that is generated in various services, frameworks, and programs, which serves as an important interface between developers and users. Log data can be highly valuable in that it conveys useful information from the users and can be easily extracted due to its partly structured nature. The rapid growth of log data has spawned numerous log data mining techniques, covering tasks such as anomaly detection, fault diagnosis, and performance improvement. For all these tasks, the first step to fully exploit log data is to effectively parse it.

Logs are printed by logging statements in the source code of programs. A log is composed of constants and variables. Constants are the words written in logging statements where variables are values of parameters in logging statements. Therefore, logs generated by one logging statement share the same constants and have different variables. When replacing the variables with a wildcard in a log, a log template is formed, which reveals the event type. The goal of log parsing is to summarize raw logs into log templates, as shown in Fig.1. Fig.1(a) shows a sequence of raw logs collected from the HDFS system released by [1]. After log parsing, each log is labeled as belonging to a log template. The corresponding templates are shown in Fig.1(b), in which "*" represents a variable. t_1, t_2, t_3 are template IDs. Each template ID corresponds to one log template.

The log parsing approaches can be categorized into offline approaches and online approaches. The former, offline log parsing approaches, require all logs are available before parsing and process them in a batch fashion. However, in real applications, logs are always collected in a streaming manner and applications have an increasing need for online monitoring and maintenance. Therefore, more and more approaches, like Drain [2] and Spell [3], focus on online log parsing which scans raw logs sequences and parses them sequentially.

However, existing approaches lack handling the

Regular Paper

This work was supported by the National Natural Science Foundation of China under Grant No. 61672163.

*Corresponding Author

081109 203518 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_−1608999687919862906 src: /10.250.19.102:54106 dest: /10.250.19.102:50010	t_1
081109 104321 28 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.250.15.198:50010 is added to blk_8894334107354324777 size 3549832	t_2
081109 203519 29 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.250.10.6:50010 is added to blk_−1608999687919862906 size 91178	t_2
081109 143353 12141 INFO dfs.DataNode$DataXceiver: Receiving block blk_4959462704623252283 src: /10.251.125.174:44166 dest: /10.251.125.174:50010	t_1
081109 212510 19 INFO dfs.FSNamesystem: BLOCK* ask 10.250.18.114:50010 to delete blk_−5140072410813878235	t_3
081109 203633 147 INFO dfs.FSNamesystem: BLOCK* ask 10.251.126.5:50010 to delete blk_−9016567407076718172 blk_−8695715290502978219	t_3

(a)

t_1: Receiving block * src: /* dest: /*
t_2: BLOCK* NameSystem.addStoredBlock: blockMap updated: * is added to * size *
t_3: BLOCK* ask * : * to delete blk_*

(b)

Fig.1. Illustration of raw logs collected from the HDFS system released by [1] and their log templates. The task of log parsing is to summarize the former into the latter. (a) Raw logs. (b) Log templates.

trade-off between effectiveness and efficiency. Specifically, we identify two factors affecting this trade-off.

The first factor is the granularity of parsing. Specifically, we call the case of only using the space character to split the log message into tokens as coarse-grained parsing, while that of using both space and other non-alphanumeric characters (like "-" and "/") to split as fine-grained parsing. Fine-grained parsers [4, 5] tend to have a better accuracy than coarse-grained ones [2, 3] when distinguishing between constants and variables, especially when parsing logs with complex structures. For example, for the log "FA||URL||taskID[2019353678] dealloc", a coarse-grained parser may summarize it as "* dealloc", while a fine-grained one may summarize it as "FA URL taskID * dealloc". Clearly, the latter retains more information useful for finding similar logs. Current online log parsing approaches, like Drain [2] and Spell [3], all adopt the simple coarse granularity. They can adopt fine granularity by changing the delimiters set to non-alphanumeric characters. However, fine granularity always leads to more tokens after tokenization. In the previous example, a coarse-grained parser obtains a sequence of length 2 while a fine-grained parser obtains a sequence of length 5. As the length of the token sequence increases, the cost of similarity computation increases. Especially, when the longest common subsequence (LCS) is used to measure the similarity, the cost increases in a quadratic fashion.

The second factor is the similarity measure used to match logs and log templates. For example, among existing online parsers, Drain [2] only supports simple comparisons between logs of the same length. Although Spell [3] can process variable-length logs by exploiting a similarity measure based on LCS, it assumes the logs belonging to the same template must have the exact prefix. Therefore, to avoid the expensive LCS computation, Spell uses a prefix tree to do early pruning, which may cause the incorrect grouping.

To address the drawbacks of existing approaches, in this paper, we propose a novel online log parser, called multi-layer parser (ML-Parser). It works in a streaming manner and can function as a real-time streaming service. The multi-layer framework aims to acquire a better parsing quality while maintaining relatively high performance. In the framework, most logs are handled through coarse-grained preprocessing and simple comparison in the first layer, while fine-grained preprocessing and LCS are used in higher layers to improve parsing quality by extending the search space to the logs of different lengths and extracting more hidden variables. We evaluate the proposed approach on real-world datasets which consist of more than 10 million lines of log messages. The results indicate that our approach outperforms various state-of-the-art techniques in parsing quality at the cost of limited extra time. In summary, our paper makes the following contributions.

• We bring a new perspective to the log parsing problem. We regard the granularity of parsing and the choice of similarity measure as two important factors in the trade-off between parsing effectiveness and efficiency.

• We present ML-Parser, an online streaming log parsing approach with a three-layer architecture, which can parse logs efficiently and accurately. We also theoretically guarantee the correctness, which means the comparison results in each layer are consistent, by introducing the concept of linear merge constraint.

• We conduct extensive experiments on 10 real-world log datasets. Results show that our approach achieves a better parsing quality at the cost of a limited performance decrease.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 describes the preliminaries. Section 4 presents an overview of the proposed ML-Parser. Section 5 proves the correctness of ML-Parser. Section 6 introduces the technical details. Section 7 reports the experimental results, and Section 8 concludes the paper.

2 Related Work

Logs can be applied to a variety of data mining tasks related to software systems, such as anomaly detection, usage analysis, and failure diagnosis. However, they are usually not structured enough to be directly used for analysis. Therefore, many approaches have been proposed to parse logs into structured formats. The traditional approach is to define regular expressions manually to extract event templates. To avoid tedious manual intervention, Xu *et al.*[6] proposed an approach using source code knowledge to infer message templates. However, it is often impractical to obtain all source code. In view of this, data-driven approaches for automated log parsing have been widely studied in both academia and industry. Simple logfile clustering tool (SLCT)[7] is known as the first automated log parser, which utilizes frequent pattern mining to extract event templates. However, it often fails to extract rare templates. LogSig[8] and log key extraction (LKE)[9] use distance metrics to cluster raw log messages into different groups. Iterative partitioning log mining (IPLoM)[10, 11] partitions log messages iteratively by certain log characteristics such as length and bipartite relationship between words. Some approaches exploit distributed computing to deal with large volumes of logs. LogMine[12] extracts templates in a hierarchical way in which different levels correspond to templates with different similarity thresholds. Logan[4] is a generalized approach for diverse datasets, utilizing generic variable identification for preprocessing, as well as a similarity score based on LCS. It also introduces a novel evaluation metric for log parsing approaches, which does not require manually labeled ground truth.

In the past few years, several online approaches have been proposed to parse logs in a streaming manner. Each online approach defines a function to measure the similarity between an event template and a raw log. When processing a log message, each approach merges it with the most similar template. However, computing the similarity between a log message and all templates is time-consuming due to a large number of templates. To tackle this issue, numerous speedup techniques have been raised. In length matters (LenMa)[13], each log message can be represented as a vector composed of word lengths and can be measured by the cosine similarity. The assumption is that logs belonging to the same template have a similar word length distribution. This assumption can be too strong for certain scenarios. Spell[3] and Drain[2] are two more applicable approaches. Spell uses a similarity measure based on LCS. It leverages an inverted index and a prefix tree to reduce the processing time. Drain[2] uses a fixed-depth parse tree which encodes specially designed rules for parsing. For each log, it only needs to compare the log with a small fraction of templates. Though Drain performs well on certain datasets, it is length-sensitive and inefficient in the case where logs start with variables.

3 Preliminaries

In this section, we describe the preliminaries of our approach, defining basic concepts related to log parsing. We first give the definition of log data and log entry.

Definition 1 (Log Dataset and Log Entry). *A log dataset is composed of a sequence of logs* $E = (e_1, e_2, \ldots, e_m)$*, where* m *is the length of the sequence and each* e_i $(1 \leqslant i \leqslant m)$ *is called a log entry. The log sequence is usually sorted by the timestamp.*

To operate on the word level, we need to convert each log to a token sequence through tokenization.

Definition 2 (Token Sequence). *A token sequence consists of a sequence of tokens or words* $s = (s_1, s_2, \ldots, s_n)$*, where* n *is the length of the token sequence.*

Definition 3 (Log Tokenization). *Given a delimiter set* $\mathcal{D}$*, log tokenization is the process of transforming a log entry* e *into a token sequence* $s_{\mathcal{D}}$ *by splitting* e *using the characters in* $\mathcal{D}$*.*

Next, we will introduce the definitions of constants and variables.

Definition 4 (Constant and Variable). *Given a group of similar log entries, for tokens appearing in corresponding positions in them, we call tokens that are common to all log entries as constants, and those that are inconsistent in each entry as variables.*

Our goal is to cluster similar log entries into a group and summarize them with a log template, which is obtained by removing variables while preserving constants in the log entries.

Definition 5 (Log Template). *Given a set of delimiters* $\mathcal{D}$*, a log template, denoted as* $t_{\mathcal{D}} = (s_1, s_2, \ldots, s_n)$*, is a token sequence, which preserves the constants while replacing variables with a wildcard token* $*$*.*

For example, given the logs "Process python.exe exited 0" and "Process cmd.exe exited 1", it can be inferred that "Process" and "exited" are constants, while "python.exe" and "java.exe" are variables if we use the space as the only delimiter of the tokens. We obtain the log template "Process * exited *" after preserving the constants while replacing variables with "*". We use log entries and token sequences interchangeably in the remainder of this paper.

Having defined the aforementioned concepts, we are finally in a position to provide the formal definition of log parsing. Here we restrict our methodology to clustering-based parsing, where we generate a template for each cluster, ensuring each entry in the cluster is similar to the template by some similarity measure. Concretely, we have the following definition.

Definition 6 (Log Parsing). *Given a log dataset* $E = (e_1, e_2, \ldots, e_m)$*, a delimiter set* $\mathcal{D}$ *and a similarity measure* Γ*, log parsing refers to the process of clustering the entries in* E *into groups and generating a template for each group, ensuring that for each entry* e *and its corresponding template* t*, it suffices that*

$$Sim_{\mathcal{D},\,\Gamma}(e, t) \geqslant \epsilon,$$

where $Sim_{\mathcal{D},\,\Gamma}(e, t)$ *is the similarity between the token sequence obtained by tokenizing* e *with* $\mathcal{D}$ *and the log template* t *under* Γ*, and* ϵ *is a user-defined similarity threshold. We call that* e *matches* t*. The template set is denoted as* $\mathcal{T} = \{t_1, t_2, \ldots, t_{|\mathcal{T}|}\}$*, where* $|\mathcal{T}|$ *is the number of templates in the set.*

Note that when calculating $Sim_{\mathcal{D},\,\Gamma}(e, t)$, we do not take into account the wildcard token $*$.

4 ML-Parser: An Overview

In this section, we present our ML-Parser approach for efficient and accurate online log parsing. We begin by detailing our options for the delimiter set $\mathcal{D}$ and the similarity measure Γ in Definition 6. Next, we move on to the architecture of ML-Parser. Finally, we present a linear merge constraint to ensure the correctness of our ML-Parser.

4.1 Delimiter Set and Similarity Measure

As it was mentioned in Definition 6, our clustering-based approach for log parsing requires a pre-defined delimiter set $\mathcal{D}$ and a similarity measure Γ. The choice of these two greatly affects the trade-off between parsing quality and efficiency. We now separately elaborate on the options we consider for these two in ML-Parser.

4.1.1 Delimiter

For the delimiter set $\mathcal{D}$, we consider two options. The first is a popular choice among early approaches[2, 3, 13], which is to use space as the sole delimiter. Concretely, we have the following definition.

Definition 7 (Delimiter Set with Space Only).

$$\mathcal{D}_{\rm s} = \{space\}.$$

$\mathcal{D}_{\rm s}$ is simple to set and is extensively used, but it cannot handle logs with complex structures. For example, for the log entries "FA||URL||taskID[2019353678] dealloc" and "FA||URL||taskID[2019353687] dealloc", the resulting log templates under $\mathcal{D}_{\rm s}$ are "* dealloc", which is clearly over-generalized.

Our second option for the delimiter set is to use non-alphanumeric characters like [4, 5]. Formally, we have the following definition.

Definition 8 (Non-Alphanumeric Delimiter Set).

$$\mathcal{D}_{\rm a} = \{c|c \textit{ is a non-alphanumeric character}\}.$$

$\mathcal{D}_{\rm a}$ is more effective at extracting hidden variables from raw logs. Let us consider the previous example with "FA||URL||taskID[2019353678] dealloc" and "FA||URL||taskID[2019353687] dealloc". Under $\mathcal{D}_{\rm a}$, we can obtain the template "FA URL taskID * dealloc", which is far more detailed than that obtained under $\mathcal{D}_{\rm s}$, as it preserves more constant tokens. Moreover, $\mathcal{D}_{\rm a}$ is better at telling whether two logs are similar. For example, let us suppose we use the Jaccard similarity[14] as the similarity measure. Specifically, the Jaccard similarity between two logs (or log templates) x and y under the delimiter set $\mathcal{D}$ is

$$Sim_{\mathcal{D},\,\rm Jaccard}(x, y) = \mathrm{Jaccard}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|},$$

where X and Y are the sets of tokens obtained by tokenizing x and y with $\mathcal{D}$ respectively. For the aforementioned two log entries, let us suppose we use $\epsilon = 0.5$ as the similarity threshold to decide whether they are similar to each other. Using $\mathcal{D}_{\rm s}$ as the delimiter leads to a similarity score of 0.33, which is below the threshold. In this way, we will falsely conclude that the two logs are dissimilar. In contrast, using $\mathcal{D}_{\rm a}$ yields a high similarity score of 0.67, which is over the threshold and is consistent with the fact that the two logs are of the same type.

4.1.2 *Similarity Measure*

For the choice of the similarity measure, we also consider two options. The first is the normalized Hamming (n-Ham) similarity for words, used by Drain[2]. Specifically, for two logs (or log templates) x and y, their n-Ham similarity under the delimiter set $\mathcal{D}$ is

$$Sim_{\mathcal{D},\,n\text{-Ham}}(x, y) = \frac{Ham(X, Y)}{n},$$

where $X = (X_1, X_2, \dots, X_n)$, and $Y = (Y_1, Y_2, \dots, Y_n)$ are the token sequences obtained by tokenizing x and y with $\mathcal{D}$ and

$$Ham(X, Y) = \sum_{i=1}^{n} equ(X_i, Y_i),$$

where

$$equ(X_i, Y_i) = \begin{cases} 1, & \text{if } X_i = Y_i\,, \\ 0, & \text{otherwise.} \end{cases}$$

The n-Ham similarity has a linear time complexity of $O(n)$. However, it cannot correctly handle cases where the two token sequences are of different lengths. For example, let us consider the logs "Process python.exe exited 0" and "Process NVIDIA share.exe exited 1". Under the alignment of n-Ham, "exited" will not be considered as a token shared by the two logs.

The second option we consider is the normalized longest common subsequence (n-LCS) similarity, which is inspired by the similarity measure used by Spell[3]. Concretely, n-LCS is based on the concept of LCS[15], which is defined as follows.

Definition 9 (Longest Common Subsequence, LCS). *Let* $X = (X_1, X_2, \dots, X_m)$ *and* $Y = (Y_1, Y_2, \dots, Y_n)$ *denote two sequences. The prefixes of* X *with* i *tokens and* Y *with* j *tokens are denoted as* X^i *and* Y^j *respectively. LCS of* X^i *and* Y^j *is*

$$LCS(X^i, Y^j) = \begin{cases} \emptyset, & \textit{if } i = 0 \textit{ or } j = 0, \\ LCS(X^{i-1}, Y^{j-1})\,\hat{}\,X_i, & \textit{if } i, j > 0 \textit{ and } x_i = y_j, \\ \max\{LCS(X^{i-1}, Y^j), LCS(X^i, Y^{j-1})\}, & \textit{otherwise.} \end{cases}$$

with

$$LCS(X, Y) = LCS(X^m, Y^n).$$

Here $\hat{}$ *means appending the* i*-th token in* X *to the existing sequence.*

For two logs (or log templates) x and y, their n-LCS similarity under the delimiter set $\mathcal{D}$ is

$$Sim_{\mathcal{D},\,n\text{-LCS}}(x, y) = \frac{|LCS(X, Y)|}{\max(|X|\,, |Y|)},$$

where X and Y are the token sequences obtained by tokenizing x and y with $\mathcal{D}$ respectively.

Compared with n-Ham, n-LCS boasts the ability to handle varying-length logs. For example, for the aforementioned logs "Process python.exe exited 0" and "Process NVIDIA share.exe exited 1", n-LCS can correctly take into account the shared token "exited". However, this comes at the price that n-LCS has a high time complexity of $O(n^2)$.

In conclusion, it is clear that for both delimiter sets and similarity measures, no single option has the optimal effectiveness and efficiency. To handle this trade-off, we propose a novel multi-layer framework, which will be introduced in Subsection 4.2.

4.2 Multi-Layer Framework of ML-Parser

In this paper, our choice is to utilize $\mathcal{D}_\mathrm{a}$ as the delimiter set and n-LCS as the similarity measure to online parse the log sequence. However, the fine-grained parsing and complex similarity will lead to a high computation cost. To address this issue, we propose a multi-layer parsing framework.

First, we give the rationale behind our approach. As analyzed beforehand, using $\mathcal{D}_\mathrm{s}$ is more efficient than using $\mathcal{D}_\mathrm{a}$, since the former leads to a shorter token sequence. Also, n-Ham is more efficient than n-LCS, since the complexity of the former is linear while that of the latter is quadratic. Therefore, we propose a three-layer framework to map logs to templates. The first layer utilizes $\mathcal{D}_\mathrm{s}$ and n-Ham, the second layer utilizes $\mathcal{D}_\mathrm{s}$ and n-LCS, and the third layer utilizes $\mathcal{D}_\mathrm{a}$ and n-LCS. In each layer, we maintain a set of templates, denoted as $\{\mathcal{T}^1, \mathcal{T}^2, \mathcal{T}^3\}$. Moreover, we build and keep the mapping relation between templates of different layers, as shown in Fig.2.

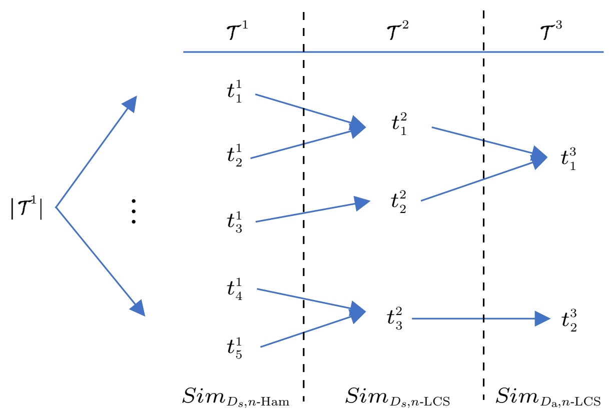


Fig.2. Structure of the multi-layer framework.

We use $|\mathcal{T}^1|$, $|\mathcal{T}^2|$, $|\mathcal{T}^3|$ to represent the number of templates in each layer, and t_j^1, t_k^2, t_l^3 to represent a template in $\mathcal{T}^1$, $\mathcal{T}^2$, and $\mathcal{T}^3$ respectively. j, k, l is the template ID in each layer respectively. When a new log e_i arrives, we first parse it with $\mathcal{D}_\mathrm{s}$, and compare the token sequence with templates in $\mathcal{T}^1$. If there exists a template t_j^1 satisfying:

- $Sim_{\mathcal{D}_\mathrm{s},\, n\text{-Ham}}(e_i, t_j^1) \geqslant \epsilon$, and
- the linear merge constraint between e_i and t_j^1 is satisfied, which will be introduced later,

then we add e_i into the log group of template t_j^1. Through the mapping relation between templates across layers, we can find the corresponding templates in $\mathcal{T}^2$ and $\mathcal{T}^3$ to which e_i belongs. Otherwise, we add a new template, $t^1_{|\mathcal{T}^1|+1}$ into $\mathcal{T}^1$ and turn to the second layer.

In the second layer, we try to find a template, say t_k^2, in $\mathcal{T}^2$, which satisfies the following two conditions,

- $Sim_{\mathcal{D}_\mathrm{s},\, n\text{-LCS}}(e_i, t_k^2) \geqslant \epsilon$, and
- the linear merge constraint between e_i and t_k^2 is satisfied.

Similarly, if either condition cannot be met, we turn to the third layer. Also, a new template $t^2_{|\mathcal{T}^2|+1}$ is inserted into $\mathcal{T}^2$. Otherwise, we build the mapping relation between $t^1_{|\mathcal{T}^1|+1}$ and t_k^2, and assign e_i into the group of t_k^2 and its corresponding template group in the third layer.

In the third layer, we first parse e_i with $\mathcal{D}_\mathrm{a}$, and try to find a template, say t_l^3, in $\mathcal{T}^3$, such that

$$Sim_{\mathcal{D}_\mathrm{a},\, n\text{-LCS}}(e_i, t_l^3) \geqslant \epsilon.$$

If so, we add e_i into the log group of t_l^3. Otherwise, we create a new template $t^3_{|\mathcal{T}^3|+1}$ in the third layer.

We use the log entries in Fig.3 as examples to illustrate the processing of different layers. Let the threshold ϵ be 0.5. For e_1 and e_2, we can easily obtain that $Sim_{\mathcal{D}_\mathrm{s},\, n\text{-Ham}}(e_1, e_2) = 0.67$, if the integers, 0 and 1, are ignored[①]. In this case, e_1 and e_2 are grouped together and represented by a template t_1^1, "Process * exited with code *". We also insert templates t_1^2 and t_1^3 of the same content into $\mathcal{T}^2$ and $\mathcal{T}^3$ respectively.

```
e1: Process python.exe exited with code 0
e2: Process cmd.exe exited with code 1
e3: Process NVIDIA share.exe exited with code 0
e4: WLSSPI-0001:zbmbankap03:10.119.165.82
e5: WLSSPI-0001:zbmbankap10:10.119.165.82
                    ⋮
```

Fig.3. Example for multi-layer processing.

Then, we start to process e_3. Although e_3 and t_1^1 only have two differences in the first layer, we cannot merge e_3 with t_1^1 because the lengths of the corresponding token sequences are different. Since we cannot find a match in the first layer, we create a new template t_2^1 for e_3 in the first layer and pass it to the second layer. We compute that $Sim_{\mathcal{D}_\mathrm{s},\, n\text{-LCS}}(e_3, t_1^2) = 0.57$. Thus, the similarity is higher than the threshold and t_2^1 is

①In practice, we use regular expressions to identify all the numbers in logs. Then the tokens recognized as numbers will be ignored when computing the similarity.

mapped to t_1^2. Since LCS can map tokens of unaligned positions, it is more applicable.

Finally, as for e_4 and e_5, it can be seen that they are strings connected by "-" and ":". If we use the delimiter set $\mathcal{D}_\mathrm{s}$, e_4 and e_5 are two different one-token sequences and $Sim_{\mathcal{D}_\mathrm{s},\, n\text{-Ham}}(e_4, e_5) = 0$. However, if we use $\mathcal{D}_\mathrm{a}$, e_4 and e_5 will be split into multiple tokens, in which most tokens are the same between e_4 and e_5. Therefore, in the third layer, they will be merged into one group.

As it was mentioned in Subsection 4.1, comparing token sequences based on $\mathcal{D}_\mathrm{s}$ is faster than that based on $\mathcal{D}_\mathrm{a}$. Also, using n-Ham is more efficient than using n-LCS. Therefore, it can be inferred that the time to process log entries increases layer by layer in the multi-layer framework. This framework can improve the performance greatly, as most of the log entries can be handled in the first or second layer. We will validate this in the experiments in Subsection 7.3.

5 Correctness of ML-Parser

By correctness, we mean that the log grouping and the set of templates generated by our multi-layer framework should be the same as those generated by only the third layer. In other words, if two logs, x and y, are grouped together in the first or second layer, it must hold that

$$Sim_{\mathcal{D}_\mathrm{s},\, n\text{-LCS}}(x, y) \geqslant \epsilon.$$

5.1 From the First Layer to the Second Layer

We first prove the correctness between the first layer and the second layer, by Lemma 1.

Lemma 1. *For any two token sequences X and Y, it holds true that*

$$|LCS(X, Y)| \geqslant Ham(X, Y).$$

Proof. For variable-length logs, LCS can not only catch the same token at the same position but also catch the same token of two logs even if they are at different positions while the Hamming distance only considers the equivalence of the token at the same position. Therefore, more shared tokens can be found by LCS and that is the reason why the length of the LCS sequence is larger than or equal to that of the Hamming distance. □

Since n-Ham is based on the Hamming distance, it can be easily deduced that for any two sequences, the value of n-LCS is greater than or equal to that of n-Ham with the same delimiter set. This means if a newly-arrived log finds a match in the first layer, it can also find a match in the second layer.

However, the same cannot be said for the log from the first (second) layer to the third layer, where the change of the delimiter set can cause inconsistency. If there are two log entries, "Process python.exe exited 0" and "Process java.exe exited java.lang.NullPointerException", their similarity score will be 0.5 in the first layer. In contrast, the score in the third layer is 0.43, which is lower than 0.5. If the similarity threshold ϵ satisfies $\epsilon \in (0.43, 0.5)$, then the comparison results are inconsistent between the first and the third layer. Under the multi-layer structure, the logs will be considered similar and the template will be updated to "Process * exited *", which is unintended if we remove the multi-layer framework and only use the third layer. In Subsection 5.2, we attempt to avoid this situation by introducing the concept of the linear merge constraint.

5.2 Linear Merge Constraint

To ensure the correctness of ML-Parser for the first (second) layer and the third layer, we now introduce the linear merge constraint. We first present some necessary definitions.

Definition 10 (New Token). *For a log entry e, X is the token sequence after the tokenization with $\mathcal{D}_\mathrm{s}$. If we switch from $\mathcal{D}_\mathrm{s}$ to $\mathcal{D}_\mathrm{a}$ to tokenize a log entry e, a new token sequence X' will be obtained. X' will be longer than or has the same length as X. Tokens which do not exist in X but in X' are denoted as the set of new tokens $NT(X)$. $NT(X)^i$ denotes the i-th new token.*

For example, with the log entry e "Process python.exe exited 0", we have the token sequences X "Process python.exe exited 0" and X' "Process python exe exited 0" with the delimiter sets $\mathcal{D}_\mathrm{s}$ and $\mathcal{D}_\mathrm{a}$, respectively. In this case, the new tokens set for X is $NT(X) = \{\text{python}, \text{exe}\}$.

For a log e, let the token set of X be s and that of X' be s', and then we have $NT(X) = s' \setminus s$, where "\" means the set difference.

We then provide the definitions of the position function and the reverse position function respectively.

Definition 11 (Position Function and Reverse Position Function). *Given a sequence $X = (X_1, X_2, \ldots, X_n)$, the position function of an element X_i is*

$$pos(X_i, X) = i,$$

while its reverse position function is

$$rpos(X_i, X) = n - i - 1.$$

Definition 12 (Linear Merge Constraint). *Given two log entries x and y, their token sequences with $\mathcal{D}_{\rm s}$ are X and Y, and new sequences with $\mathcal{D}_{\rm a}$ corresponding to X and Y are X' and Y', respectively. We define the normalized similarity between new tokens shared by the two logs x and y, denoted as $Sim_{\rm sa}(x,y)$, as follows*:

$$Sim_{\rm sa}(x,y) = \frac{\sum_i poseq(NT(X)^i, X', NT(Y)^i, Y')}{\max(|NT(X)|, |NT(Y)|)},$$

where

$$poseq(a, X', b, Y') = \begin{cases} 1, & if\ a == b\ \ and \\ & (pos(a,X') = pos(b,Y')\ \ or \\ & \ rpos(a,X') = rpos(b,Y')), \\ 0, & otherwise. \end{cases}$$

We call that x and y satisfy the linear merge constraint, if it holds that

$$Sim_{\rm sa}(x,y) \geqslant \epsilon.$$

For example, if we have two log entries e_1 "Process python.exe exited 0" and e_2 "Process NVIDIA share.exe exited 1", their token sequences under $\mathcal{D}_{\rm s}$ and $\mathcal{D}_{\rm a}$ are X, Y and X', Y', respectively. We will get the new tokens for X and Y: $NT(X) = ({\rm python}, {\rm exe})$, $NT(Y) = ({\rm share}, {\rm exe})$. Then, we will see $\sum_i posequ(NT(X)^i, X', NT(Y)^i, Y') = 1$ and $Sim_{\rm sa}(x,y) = 0.5$. The score of $Sim_{\rm sa}(x,y)$ is equal to the similarity threshold ϵ and the linear merge constraint is met.

5.3 From the First Layer to the Third Layer

With this constraint, we have the following two theorems.

Theorem 1. *Given two logs x and y, if the following two conditions are satisfied*:

1) *x and y are similar in the second layer, i.e.,*

$$Sim_{\mathcal{D}_{\rm s}, n\text{-LCS}}(x,y) \geqslant \epsilon,$$

2) *the linear merge constraint is met, i.e.,*

$$Sim_{\rm sa}(x,y) \geqslant \epsilon,$$

then x and y must be similar in the third layer, i.e.,

$$Sim_{\mathcal{D}_{\rm a}, n\text{-LCS}}(x,y) \geqslant \epsilon.$$

Proof. Given x and y, assuming X and Y are token sequences under $\mathcal{D}_{\rm s}$, and X' and Y' are those under $\mathcal{D}_{\rm a}$, we have

$$|LCS(X,Y)| \geqslant \max(|X|,|Y|) \times \epsilon.$$

Let $A(x,y) = Sim_{\rm sa}(x,y) \times \max(|NT(X)|,|NT(Y)|)$, and we have

$$A(x,y) \geqslant \epsilon \times \max(|NT(X)|,|NT(Y)|).$$

Thus we have

$$\begin{aligned} &|LCS(X,Y)| + A(x,y) \\ &\geqslant \epsilon \times (\max(|X|,|Y|) + \max(|NT(X)|,|NT(Y)|)). \end{aligned}$$

By Lemma 1, we have

$$|LCS(X'-X, Y'-Y)| \geqslant Ham(X'-X, Y'-Y).$$

It can be easily inferred that

$$\begin{aligned} &|LCS(X',Y')| \\ &\geqslant |LCS(X,Y)| + |LCS(X'-X, Y'-Y)|, \end{aligned}$$

and

$$|LCS(X'-X, Y'-Y)| \geqslant A(x,y).$$

Therefore

$$|LCS(X',Y')| \geqslant |LCS(X,Y)| + A(x,y).$$

Besides, we have

$$\begin{aligned} &\max(|X'|,|Y'|) \\ &\leqslant \max(|X|,|Y|) + \max(|NT(X)|,|NT(Y)|). \end{aligned}$$

Therefore

$$|LCS(X',Y')| \geqslant \epsilon \times \max(|X'|,|Y'|).$$

Thus

$$Sim_{\mathcal{D}_{\rm a}, n\text{-LCS}}(x,y) = \frac{|LCS(X,Y)|}{\max(|X'|,|Y'|)} \geqslant \epsilon. \qquad \square$$

Theorem 2. *Given two logs x and y, if the following two conditions are satisfied*:

1) *x and y are similar in the first layer, i.e.,*

$$Sim_{\mathcal{D}_{\rm s}, n\text{-Ham}}(x,y) \geqslant \epsilon,$$

2) *the linear merge constraint is met, i.e.,*

$$Sim_{\rm sa}(x,y) \geqslant \epsilon,$$

then x and y must be similar in the third layer, i.e.,

$$Sim_{\mathcal{D}_{\rm a}, n\text{-LCS}}(x,y) \geqslant \epsilon.$$

Proof. From Lemma 1, we have

$$|LCS(Xs, Ys)| \geqslant Ham(Xs, Ys),$$

then

$$Sim_{\mathcal{D}_{\rm s},\, n\text{-LCS}}(x, y) \geqslant Sim_{\mathcal{D}_{\rm s},\, n\text{-Ham}}(x, y) \geqslant \epsilon.$$

By Theorem 1, we have

$$Sim_{\mathcal{D}_{\rm a},\, n\text{-LCS}}(x, y) \geqslant Sim_{\mathcal{D}_{\rm s},\, n\text{-LCS}}(x, y) \geqslant \epsilon. \qquad \square$$

These two theorems guarantee the consistency between the first (second) layer and the third layer. Combined with Lemma 1, the correctness of our entire multi-layer structure is guaranteed.

6 Technical Details of ML-Parser

Having presented an overview of ML-Parser, we now dive into the technical details. As previously mentioned, our approach runs in a streaming manner, which means the log groups and templates are updated simultaneously as new logs arrive. Concretely, for a newly-arrived raw log entry, we first preprocess it with a user-defined regular expression set to remove variables that can be easily identified. The preprocessed log is fed into the multi-layer framework of ML-Parser.

In the first layer, we compare the preprocessed log with the templates in $\mathcal{T}^1$. If a match is found, the log is added to the matching group and the corresponding log template is updated by preserving constants in the group and replacing variables with wildcards. Otherwise, we generate a new log template for the new log in the first layer and feed it into the second layer to find a match in $\mathcal{T}^2$. If it is found, the matching template is updated and the mapping relation of the two templates in different layers is recorded. Otherwise, a similar process will take place from the second layer to the third layer.

6.1 Preprocessing of Log Entries

Before feeding the newly-arrived log entry into the multi-layer framework, we conduct preprocessing on it, which has been proved to be an important step to improve parsing performance [2, 4, 5, 16]. Concretely, we exploit regular expressions to pre-remove certain variables from the raw log. The regular expressions are provided by users as an optional way to feed in domain knowledge. For efficiency, only a very limited number of the simple regular expressions are used in preprocessing.

Specifically, we list the regular expressions used in Table 1.

Table 1. Regular Expressions Used

Variable Type	Example	Regular Expression
IP	126.1.17.9	(\d+\.){3}\d+
Decimal	1080	(ˆ|)\d+(|$)
Time	12:25:31	\d{2}:\d{2}(:\d{2})*
Hexadecimal	A2F35C	(ˆ|)([0-9a-fA-F]){3,}(|$)

After employing the regular expressions, we further split a log entry into a sequence of tokens with the delimiters presented in Subsection 4.1.

6.2 First Layer

After preprocessing, the log entry is fed into our multi-layer framework. In the first layer, the preprocessed log entry is compared with the templates in $\mathcal{T}^1$ that have the same length and begin with the same token as the entry. To efficiently find such templates, we use a prefix tree [17] to index the existing log templates in $\mathcal{T}^1$. Concretely, we exploit a prefix tree with the maximum depth, which is almost identical to the fixed-depth prefix tree in Drain [2]. All the templates are stored in leaf nodes.

Fig.4 shows an illustration of the prefix tree. The templates of different lengths are stored in different leaf nodes. For example, the leftmost leaf node stores all templates that satisfy: 1) the length of the templates' token sequence is 3; 2) the prefix of the templates' token sequence is (A, Z). The prefix tree can help us find the candidate templates that the new log should compare with.

After obtaining the templates to compare with from the prefix tree, we compute the similarity between these templates and the new log under $Sim_{\mathcal{D}_{\rm s},\, n\text{-Ham}}$. The template with the highest similarity will be selected. If the similarity score is larger than the user-defined threshold ϵ, then the log message should probably be in that group. However, we still require the linear merge constraint introduced in Subsection 5.2 to be met. If so, the log will be added into the group, the corresponding template will be updated by scanning the tokens in corresponding locations of the log and the template, and detecting and replacing the variables with wildcards. For example, the logs "Process python.exe exited with code 0" and "Process cmd.exe exited with code 1" will be merged as "Process * exited with code *". Accordingly, the children templates in the next two layers are also updated to keep consistent with the templates in the first layer. Otherwise, a new log group will be created, which only contains the new log. Also, the corresponding template is identical to the new log.

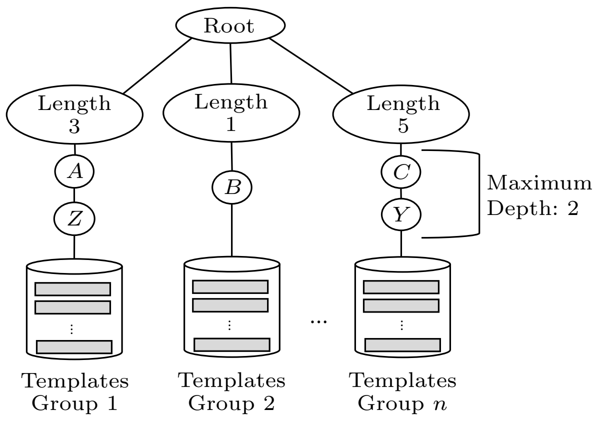


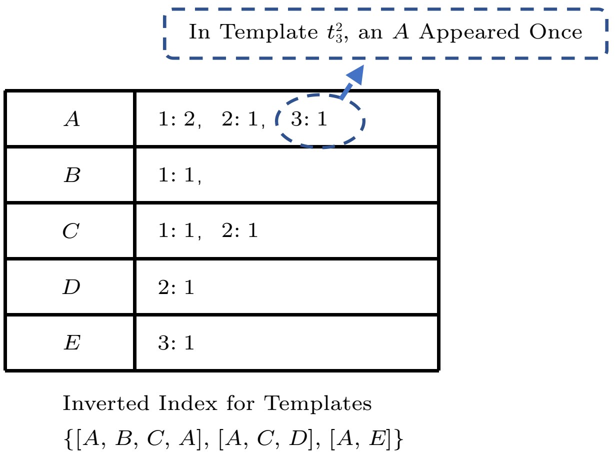
Fig.4. Illustration of the prefix tree used to store log templates in the first layer.

6.3 Second Layer

When no matching template is found in the first layer, the log e_i will be passed to the second layer, to determine whether it will match an existing template in $\mathcal{T}^2$. Concretely, we compare e_i with those templates currently residing in the second layer under $Sim_{\mathcal{D}_\mathrm{s},\, n\text{-LCS}}$. If the maximum similarity value exceeds the user-defined threshold ϵ and the linear merge constraint is met, then the matching template t_k^2 is merged with e_i. Otherwise, the merge is rejected and a new template $t^2_{|\mathcal{T}^2|+1}$ identical to the coming one will be added to $\mathcal{T}^2$ and will be passed to the third layer to find a match in $\mathcal{T}^3$. The merge is executed by replacing all the tokens in the templates which are not in the LCS sequence with wildcards. For example, the template "Process NVIDIA Share.exe exited with 1" and e_i "Process python.exe exited with 0" are merged as "Process * exited with *". Also, the children templates in the third layer should be updated accordingly.

In practice, we accelerate the matching process by pruning the unnecessary templates to calculate n-LCS with e_i. The idea is to exploit the Jaccard similarity to prune a potentially large number of templates that are definitively not similar enough to the new template under n-LCS. Specifically, it can be proved that if the Jaccard similarity between a log and a template is below the threshold ϵ, they are definitely dissimilar to each other under n-LCS.

To efficiently compute the Jaccard values, we maintain an inverted index built upon the tokens, which is shown in Fig.5. Specifically, for each unique token s_j, the inverted index has a $<K, V>$ pair. The key is s_j. The value is a set of $(id, count)$ pairs, where id is the ID of certain template in $\mathcal{T}^2$ and $count$ is the occurrence times of token s_j in this template. The wildcards are not indexed. With the inverted index, Jaccard values can be obtained in linear time. Note that when two templates are merged, the inverted index needs to be updated accordingly.


In Template t_3^2, an A Appeared Once
Inverted Index for Templates
{[A, B, C, A], [A, C, D], [A, E]}

A	1: 2, 2: 1, 3: 1
B	1: 1,
C	1: 1, 2: 1
D	2: 1
E	3: 1

Fig.5. Illustration of the inverted index used to store log template in the second layer. t_3^2 is the third template in the second layer.

6.4 Third Layer

The second layer mainly deals with logs of varying lengths. However, this is not enough because the logs may have complex structures occasionally. Specifically, in some cases, one token may be composed of a few simpler tokens connected with some non-space delimiters, such as specific serial numbers (e.g., "blk_38860"), equations (e.g., "UserName=jack"), and complex names of system components (e.g., "dfs.DataNode$PacketResponder"). Although they can be handled by introducing handcrafted regular expressions, this requires much manual effort. By contrast, we employ finer granularity to automatically deal with this.

To be specific, given the incoming log e_i, we parse it based on $\mathcal{D}_\mathrm{a}$ and compare the token sequence against templates in $\mathcal{T}^3$ using $Sim_{\mathcal{D}_\mathrm{a},\, n\text{-LCS}}$. Here we substitute the coarse-grained $\mathcal{D}_\mathrm{s}$ with the fine-grained $\mathcal{D}_\mathrm{a}$. Similar to the second layer, the search space can be reduced by exploiting the Jaccard similarity and the inverted index. If a match is found, the matching template and e_i will be merged. Otherwise, there are no log templates matching the log entry in all three layers and a new template will be created for this log in the third layer, which is identical to the new log entry.

7 Experiments

In this section, we evaluate the effectiveness and efficiency of ML-Parser, by comparing it with the state-of-the-art approaches. All experiments are performed on a Windows server with dual Intel® Xeon® CPU E5-2673 v3 @ 2.40 GHz CPUs, 384 GB memory. All approaches are implemented in Python 3.7.

We use the popular log benchmark which is publicly available in [1]. The detailed information of the used datasets is shown in Table 2. These datasets have been commonly used in [2,3,18], with sizes ranging from 2.24 MB to 2.74 GB.

We select three existing online log parsers to compete against our approach in terms of efficiency and effectiveness, which are Spell[3], Drain[2] and an improved version of Drain[18] (abbreviated as Drain2). The idea of Spell is to compare the log with templates using the LCS similarity metrics. Besides, Drain and Drain2 use a prefix tree and a simple loop method for speedup. On the other hand, Drain partitions all the log templates by length and then performs a search in a fixed-depth prefix tree. Drain2 features improvements such as eliminating the need to define a threshold for similarity comparison, taking tokens at the end of the sequences into the prefix tree, and introducing an optional merge mechanism for log templates of different lengths.

Table 3 shows the parameter settings of these approaches used in our experiments. For Spell and Drain, we use the settings recommended in [1]. As for Drain2[18], we set *st* to 0.95 for the Proxifier dataset and 1 for the rest, following [18]. Besides, the aforementioned rival approaches utilize coarse-grained preprocessing.

Since our ML-Parser utilizes fine-grained granularity, that is, $\mathcal{D}_{\rm a}$, we enable this option for the rival approaches for fairness. These modified rival approaches will be distinguished by the suffix "S" in their names. Note that although SpellS (fine-grained Spell) also uses $\mathcal{D}_{\rm a}$ and LCS, similar to our approach, it uses some approximate techniques to speed up, which impacts the accuracy.

7.1 Effectiveness of ML-Parser

In this subsection, we evaluate the parsing quality of our approach. We use the supervised metric accuracy applied in [2, 3, 18] and the unsupervised metric *Loss* in [4] for evaluation. The accuracy results of each approach are shown in Table 4. It can be seen that the proposed approach achieves rank-1 on four datasets, and rank-2 on the other datasets, which verifies the effectiveness of the proposed approach.

The *Loss* metric is based on the idea that the log templates should be neither over-generalized nor under-parsed. Given a set of log templates $\mathcal{T} = \{t_1, t_2, \ldots, t_{|\mathcal{T}|}\}$, we have

$$\begin{aligned}
&AverageTokenLost(t_i)\\
&= AverageMatchLength(t_i) - |t_i|\,,\\
&LengthFactor(\mathcal{T}) = (\log|\mathcal{T}|)^{\theta},\\
&QualityFactor(\mathcal{T}) = \sum_{i=1}^{|\mathcal{T}|}\left(\frac{AverageTokenLost(t_i)}{|t_i|}\right)^2,\\
&Loss = LengthFactor(\mathcal{T}) + QualityFactor(\mathcal{T}),
\end{aligned}$$

where $AverageMatchLength(t_i)$ is the average length of all sequences matched to template t_i during evaluation.

Table 2. Dataset Information

Dataset	Description	Dataset Size	Number of Logs	Length	Number of Events
HDFS	Hadoop distributed file system log	1.47 GB	1.12×10^7	8–29	30
Hadoop	Hadoop mapreduce job log	46.34 MB	3.94×10^5	0–52	N/A
Spark	Spark job log	2.74 GB	3.32×10^7	1–78	N/A
Zookeeper	ZooKeeper service log	9.94 MB	7.43×10^4	7–26	95
BGL	Blue Gene/L supercomputer log	708.76 MB	4.74×10^6	9–102	619
HPC	High performance cluster log	32.00 MB	4.33×10^5	6–104	104
Linux	Linux system log	2.24 MB	2.56×10^4	5–24	N/A
Mac	Mac OS log	16.10 MB	1.17×10^5	0–104	N/A
Proxifier	Proxifier software log	2.42 MB	2.13×10^4	10–27	8
Apache	Apache web server error log	4.90 MB	5.65×10^4	7–23	N/A

Table 3. Parameters for Approaches

Approach	Parameter Description	Value Range
Drain [2]	Similarity threshold *st*	$[0.2, 0.7]$
	Tree depth	$\{4, 5, 6\}$
Drain2 [18]	Merge threshold *mt*	$[0.95, 1]$
Spell [3]	Similarity threshold τ	$[0.5, 0.95]$
ML-Parser	Similarity threshold *st*	$[0.2, 0.7]$
	Prefix tree depth	{4, 5, 6}

Table 4. Accuracy for Each Approach

Approach	HDFS	BGL	HPC	Proxifier	Zookeeper
Drain	0.84	0.78	0.75	0.51	**0.95**
DrainS	0.67	0.75	0.72	0.50	**0.95**
Drain2	0.84	0.76	0.77	0.01	**0.95**
Drain2S	0.67	0.60	0.72	0.02	**0.95**
Spell	**0.99**	0.68	0.57	**1.00**	**0.95**
SpellS	0.32	0.20	0.57	0.52	**0.95**
ML-Parser	0.85	**0.81**	**0.78**	**1.00**	**0.95**

Note: The highest accuracy on each dataset is marked in bold. HDFS is the abbreviation of Hadoop Distributed File. BGL is the abbreviation of Blue Gene/L. HPC is the abbreviation of High Performance Cluster.

$LengthFactor$ reflects how many templates are generated in $\mathcal{T}$. On the other hand, $QualityFactor$ represents the ratio of generalized tokens, which indicates the extent of over-generalization. θ is a hyper-parameter to control the importance of one factor over another. It is set to 1.5 according to [4]. The smaller the metric $Loss$ is, the higher the quality of the result template set $\mathcal{T}$ is.

As shown in Table 5 and Table 6, most of the approaches that employ fine-grained preprocessing identify fewer templates and have a lower $Loss$ on most datasets. This is probably because more hidden variables can be found, which verifies our claim that fine granularity is preferred. On most datasets (7/10), ML-Parser has the smallest $Loss$. It is worth noting that for all four datasets of long logs ($length > 100$), our approach works better than SpellS.

Compared with other approaches, SpellS and our approach generate fewer log templates. However, on some datasets, SpellS has generated fewer templates, but suffered a higher $Loss$. This implies that some important tokens are lost in the parsing process. In summary, our approach outperforms all the other approaches except SpellS in terms of $Loss$ and the number of the templates generated. It obtains the lowest $Loss$ on seven out of 10 datasets among all the approaches.

7.2 Efficiency of ML-Parser

We now move on to the efficiency of our approach. Table 7 shows the running time of ML-Parser and rival

Table 5. Number of Log Templates Generated by Each Approach on Each Dataset

Approach	BGL	Hadoop	Spark	HDFS	HPC	Proxifier	Zookeeper	Linux	Apache	Mac
Drain	1 848	347	571	48	325	57	77	500	29	4 000
DrainS	1 251	316	353	55	122	13	88	548	39	2 090
Drain2	1 202	288	639	48	143	10	76	497	36	758
Drain2S	15 878	335	4 458	56	130	13	86	553	43	850
Spell	27 266	5 635	960	38	330	339	180	452	29	1 928
SpellS	298	181	148	30	61	8	78	297	27	535
ML-Parser	604	254	236	35	73	10	73	311	28	557

Table 6. Loss Values for Each Approach

Approach	BGL	Hadoop	Spark	HDFS	HPC	Proxifier	Zookeeper	Linux	Apache	Mac
Drain	20.68	14.22	16.43	7.71	14.00	8.24	9.09	15.52	6.21	23.92
DrainS	19.83	13.99	14.69	8.81	10.72	5.49	9.83	16.84	7.17	21.54
Drain2	19.08	13.53	15.13	7.72	11.24	3.64	9.07	15.52	6.80	17.13
Drain2S	18.60	13.99	15.03	8.81	10.92	5.39	9.83	16.84	7.17	21.54
Spell	32.65	25.39	18.22	7.11	15.29	14.15	11.85	15.23	6.21	20.87
SpellS	**13.71**	**12.11**	**11.63**	7.31	8.50	4.42	9.49	14.60	6.22	16.36
ML-Parser	16.36	13.12	12.90	**6.85**	**7.09**	**3.63**	**8.94**	**13.83**	**6.12**	**15.99**

Note: The smallest *Loss* on each dataset is marked in bold.

Table 7. Running Time (s) of Each Approach

Approach	BGL	Hadoop	Spark	HDFS	HPC	Proxifier	Zookeeper	Linux	Apache	Mac
Drain	966	33	5 411	2 079	75	4	13	4	8	29
DrainS	1 259	54	8 623	3 130	96	6	18	7	14	51
Drain2	941	35	5 453	2 174	78	5	14	5	10	29
Drain2S	1 540	55	8 673	3 145	100	6	19	7	15	43
Spell	3 970	103	5 283	1 908	67	11	13	6	8	39
SpellS	957	66	7 181	2 374	74	6	19	8	14	45
ML-Parser	1 193	47	7 286	2 570	83	6	15	8	12	39

approaches. As is shown, approaches with fine granularity generally run slower than other approaches. The only exception is SpellS, which is fast because it has a fixed threshold of 0.5 for pre-filtering in the prefix tree and the simple loop.

By bringing in more delimiters under a fine-grained setting, the log sequences are longer and it is more possible for a log to find a match. Our approach is slightly slower than Drain and Drain2. This is because 1) it has a much larger search space to handle varying-length logs; 2) it employs not just linear match; 3) it involves fine-grained tokenization. In conclusion, our approach achieves better parsing quality at the cost of a slightly longer running time.

7.3 Case Studies

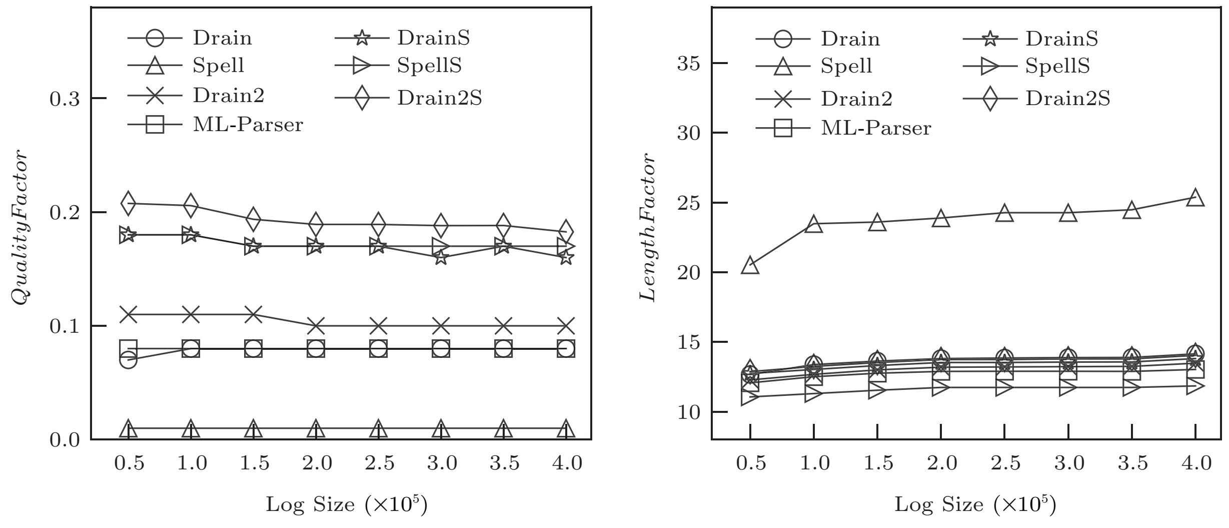
We perform case studies on two log datasets, Hadoop and Spark, to further showcase the effectiveness and efficiency of ML-Parser. We select the log datasets of Hadoop and Spark, due to the popularity of these systems. Moreover, the Spark dataset is the largest one, based on which we can test the performance of the proposed approach to handle the large dataset. We compare ML-Parser with Drain, Drain2, Spell and their fine-grained version with the suffix "S" in their names. Fig.6 shows the results of $LengthFactor$ and $QualityFactor$.

It can be seen that our approach has less templates (represented by $LengthFactor$) than the other approaches except SpellS on both datasets. Although SpellS generates fewer templates, it suffers a higher $QualityFactor$. The differences among different approaches are greater for larger and more complicated datasets. As with $QualityFactor$, we achieve almost the same $QualityFactor$ as Drain on Hadoop and have a lower $QualityFactor$ on Spark. This demonstrates the effectiveness of our linear merge constraint, which rejects unintended merges between log templates.

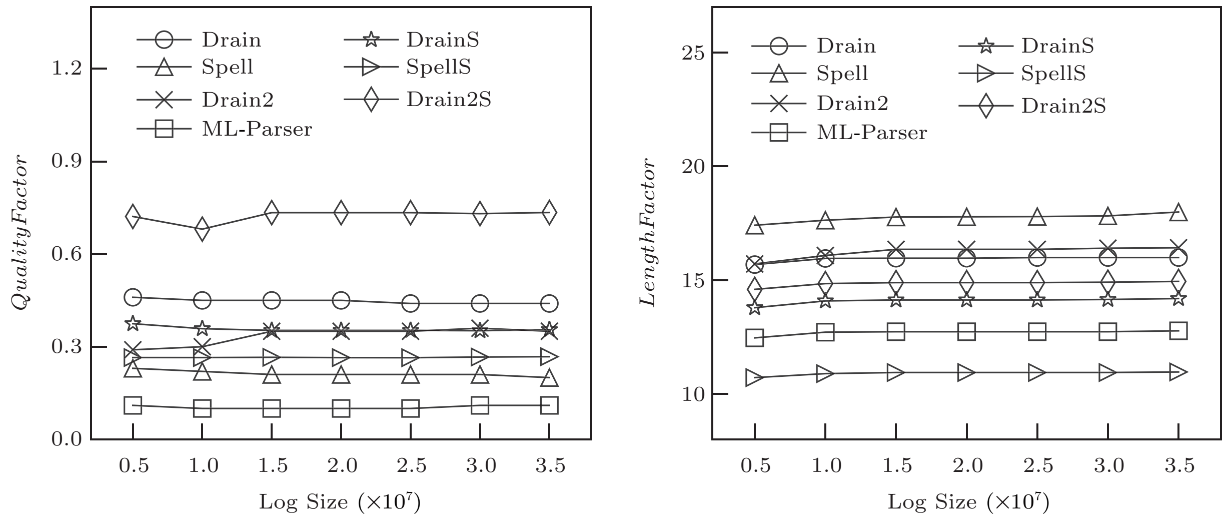
Table 8 shows the number of log templates returned in each layer of ML-Parser. It can be seen that in the first layer, we only have 550 templates in the Hadoop dataset and 805 templates in the Spark dataset. Compared with the number of raw logs, the number of log templates drops by more than 99%, which means that most logs can find a matching template in the first layer. It can also be inferred that the LCS-based merging in the third layer is more effective because fine-grained parsing extracts nested tokens and allows for a larger search space. The numbers in parentheses indicate the compression ratio of the log templates to the raw logs. For example, 1.390‰ of the first layer means after the Hadoop raw logs are abstracted into the log templates in the first layer, the number of log types can be reduced to 1.39‰ of the raw ones.

8 Conclusions

In this paper, we proposed ML-Parser, a novel online log parser that works in a streaming manner, which features a multi-layer framework to generate the log templates efficiently and accurately. By leveraging a prefix tree with the maximum depth and linear similarity search with a linear merge constraint, the LCS-based search process for log templates is speeded up. With incorporated fine-grained preprocessing, more tokens are extracted and more accurate log templates are generated. Our experiments showed that ML-Parser achieves the highest accuracy on four out of five labeled datasets. It also outperforms other approaches on seven out of 10 datasets in the unsupervised metric. In conclusion, ML-Parser can acquire better parsing quality while maintaining relatively high performance. For future work, we plan to investigate the reason for error matching between logs and templates, and explore the use of logs' semantic information to reduce error matching.



(a)



(b)

Fig.6. *LengthFactor* and *QualityFactor* of each approach on the (a) Hadoop and (b) Spark log datasets.

Table 8. Number of Log Templates in Each Layer of ML-Parser

Dataset	Number of Raw Logs	First Layer	Second Layer	Third Layer
Hadoop	394 308	550(1.390‰)	528(1.340‰)	254(0.640‰)
Spark	33 236 604	805(0.024‰)	781(0.023‰)	236(0.007‰)

References

[1] Zhu J, He S, Liu J, He P, Xie Q, Zheng Z, Lyu M R. Tools and benchmarks for automated log parsing. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice*, May 2019, pp.121-130. DOI: 10.1109/ICSE-SEIP.2019.00021.

[2] He P, Zhu J, Zheng Z, Lyu M R. Drain: An online log parsing approach with fixed depth tree. In *Proc. the 2017 IEEE International Conference on Web Services*, June 2017, pp.33-40. DOI: 10.1109/ICWS.2017.13.

[3] Du M, Li F. Spell: Streaming parsing of system event logs. In *Proc. the 2016 International Conference on Data Mining*, Dec. 2016, pp.859-864. DOI: 10.1109/ICDM.2016.0103.

[4] Agrawal A, Karlupia R, Gupta R. Logan: A distributed online log parser. In *Proc. the 35th IEEE International Conference on Data Engineering*, April 2019, pp.1946-1951. DOI: 10.1109/ICDE.2019.00211.

[5] Agrawal A, Dixit A, Kapadia D, Karlupia R, Agrawal V, Gupta R. Delog: A privacy preserving log filtering framework for online compute platforms. arXiv:1902.04843, 2019. https://arxiv.org/abs/1902.04843, Jan. 2021.

[6] Xu W, Huang L, Fox A, Patterson D, Jordan M I. Detecting large-scale system problems by mining console logs. In *Proc. the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, October 2009, pp.117-132. DOI: 10.1145/1629575.1629587.

[7] Vaarandi R. A data clustering algorithm for mining patterns from event logs. In *Proc. the 3rd IEEE Workshop on*

IP Operations & Management, Oct. 2003, pp.119-126. DOI: 10.1109/IPOM.2003.1251233.

[8] Tang L, Li T, Perng C S. LogSig: Generating system events from raw textual logs. In *Proc. the 20th ACM International Conference on Information and Knowledge Management*, October 2011, pp.785-794. DOI: 10.1145/2063576.2063690.

[9] Fu Q, Lou J G, Wang Y, Li J. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proc. the 9th IEEE International Conference on Data Mining*, Dec. 2009, pp.149-158. DOI: 10.1109/ICDM.2009.60.

[10] Makanju A A, Zincir-Heywood A N, Milios E E. Clustering event logs using iterative partitioning. In *Proc. the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, June 2009, pp.1255-1264. DOI: 10.1145/1557019.1557154.

[11] Makanju A, Zincir-Heywood A N, Milios E E. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 2011, 24(11): 1921-1936. DOI: 10.1109/TKDE.2011.138.

[12] Hamooni H, Debnath B, Xu J, Zhang H, Jiang G, Mueen A. LogMine: Fast pattern recognition for log analytics. In *Proc. the 25th ACM International on Conference on Information and Knowledge Management*, October 2016, pp.1573-1582. DOI: 10.1145/2983323.2983358.

[13] Shima K. Length matters: Clustering system log messages using length of words. arXiv:1611.03213, 2016. https://arxiv.org/abs/1611.03213, Jan. 2021.

[14] Levandowsky M, Winter D. Distance between sets. *Nature*, 1971, 234(5323): 34-35. DOI: 10.1038/234034a0.

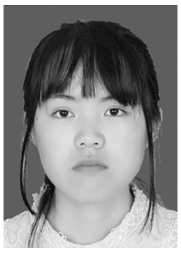
[15] Nakatsu N, Kambayashi Y, Yajima S. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 1982, 18(2): 171-179. DOI: 10.1007/BF00264437.

[16] He P, Zhu J, He S, Li J, Lyu M R. An evaluation study on log parsing and its use in log mining. In *Proc. the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 28-July 1, 2016, pp.654-661. DOI: 10.1109/DSN.2016.66.

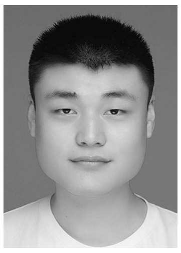
[17] Yang Y, Zhang W, Zhang Y, Lin X, Wang L. Selectivity estimation on set containment search. *Data Science and Engineering*, 2019, 4(3): 254-268. DOI: 10.1007/s41019-019-00104-1.

[18] He P, Zhu J, Xu P, Zheng Z, Lyu M R. A directed acyclic graph approach to online log parsing. arXiv:1806.04356, 2018. https://arxiv.org/abs/1806.04356, Jan. 2021.

Yu-Qian Zhu received her B.S. degree in software engineering from Donghua University, Shanghai, in 2019. She is currently studying for her M.S. degree in computer science at Fudan University, Shanghai. Her research interests include data mining and information retrieval.



Jia-Ying Deng received her B.S. degree in computer science from Donghua University, Shanghai, in 2019. She is currently studying for her M.S. degree in computer science at Fudan University, Shanghai. Her research interests include anomaly detection, data mining and pattern recognition.



Jia-Chen Pu received his B.S. degree in computer science from Shanghai Maritime University, Shanghai, in 2017, and his M.S. degree in computer science from Fudan University, Shanghai, in 2020. His research interests include deep learning, pattern recognition and artificial intelligence.

Peng Wang received his Ph.D. degree in computer science from Fudan University, Shanghai, in 2007. Now he is an associate professor in School of Computer Science, Fudan University, Shanghai. His research interests include database, data mining, and series data processing. He has published more than 30 papers in refereed international journals and conference proceedings.

Shen Liang received his Ph.D. degree in computer science from Fudan University, Shanghai, in 2020, and is now a post-doctoral researcher at Fudan University, Shanghai. His research interests include data management, data mining, artificial intelligence and their applications to smart manufacturing and health informatics.

Wei Wang received his Ph.D. degree in computer science from Fudan University, Shanghai, in 1998. Now he is a professor in School of Computer Science, Fudan University, Shanghai. His research interests include database, data mining, and series data processing. He has published more than 100 papers in refereed international journals and conference proceedings.