2016 IEEE 16th International Conference on Data Mining

Spell: Streaming Parsing of System Event Logs

Min Du, Feifei Li
School of Computing, University of Utah
mind@cs.utah.edu, lifeifei@cs.utah.edu

***Abstract*—System event logs have been frequently used as a valuable resource in data-driven approaches to enhance system health and stability. A typical procedure in system log analytics is to first parse unstructured logs, and then apply data analysis on the resulting structured data. Previous work on parsing system event logs focused on offline, batch processing of raw log files. But increasingly, applications demand online monitoring and processing. We propose an online streaming method Spell, which utilizes a longest common subsequence based approach, to parse system event logs. We show how to dynamically extract log patterns from incoming logs and how to maintain a set of discovered message types in streaming fashion. Evaluation results on large real system logs demonstrate that even compared with the offline alternatives, Spell shows its superiority in terms of both efficiency and effectiveness.**

I. Introduction

The increasing complexity of modern computer systems has become a significant limiting factor in deploying and managing them. Being able to be alerted and mitigate the problem right away has become a fundamental requirement in many systems. As a result, automatically detecting anomalies upon happening in an online fashion is an appealing solution. Data-driven methods are heavily employed to understand complex system behaviors, for example, exploring machine data for automatic pattern discovery and anomaly detection [1]. System logs, as a universal data source that contains important information such as usage patterns, execution paths, and program running status, are valuable assets in assisting these data-driven system analytics, in order to gain insights that are useful to enhance system health, stability, and usability.

The effectiveness of system log mining has been validated by recent literature. Logs could be used to detect execution anomalies [2], [3], [4], monitor network failures [5], or even find software bugs [6]. Researchers have also used system logs to discover and diagnose performance problems [7]. Recently to untangle the interleaved event logs from concurrent systems has also become a hot topic of research [8].

To alleviate the pain of diving into massive *unstructured* log data, in most prior work, the first and foremost step is to automatically *parse the unstructured system logs to structured data* [2], [3], [4], [6]. There have been a substantial study on how to achieve this, for example, using regular expressions [8], leveraging the source code [6], or parsing purely based on system log characteristics using data mining approaches such as clustering and iterative partitioning [2], [9], [10], [11]. Nevertheless, except the approach that uses regular expressions which requires domain-specific expert knowledge [8], hence, does not work for general purpose system log parsing, or the approach that leverages the source code [12] which is often unavailable, none of the previous methods could achieve online parsing in a streaming fashion. Some work claimed "online" processing, but with the requirement of doing some extensive offline processing first, and only then matching log entries with the data structures and patterns identified first through the offline, batched process [13].

There is also an increasing demand to properly manage and store system logs [14]. A log management system typically has a log shipper installed on each node to forward log entries to a centralized server, which often contains a log parser, a log indexer, a storage engine and a user interface. In such systems the default log parser only parses simple schema information such as timestamp and hostname. The log entry itself is treated as an unstructured text value. An online *structured* approach that could parse the event logs into structured data could make the logs much easier to query, summarize and aggregate.

Log entries are produced by the "print" statements in a system program's source code. As such, we can view a log entry as a collection of ("message type", "parameter value") pairs. For example, a log printing statement *printf("File %d finished.", id);* contains a *constant* message type *File finished* and a *variable* parameter value which is the file id. Hence, the goal of a *structured log parser* is to identify the message type *File * finished*, where * stands for the place holder for variables (parameter values).

Contributions. In this paper, we propose Spell, a structured Streaming Parser for Event Logs using an LCS (longest common subsequence) based approach. Spell parses unstructured log messages into structured message types and parameters in an online streaming fashion. The time complexity to process each log entry e is close to linear (to the size of e).

With streaming, real-time message type and parameter extraction produced by Spell, not only it provides a concise, intuitive summary for the end users, but the logs are also represented by clean structured data to be processed and analyzed further using advanced data analytics methods by downstream analysts. Using two state-of-the-art offline methods to automatically extract message types and parameters from raw log files as the competing baseline, our study shows that even compared with the offline methods, Spell still outperforms them in terms of both efficiency and effectiveness.

The rest of this paper is organized as follows. Section II provides the problem formulation and a literature survey. Section III presents our streaming Spell algorithm and a number of optimizations. Section IV evaluates our method

2374-8486/16 $31.00 © 2016 IEEE
DOI 10.1109/ICDM.2016.160

using large real system logs. Finally, section V concludes the paper and section VI is our acknowledgement.

II. Preliminary and Background

A. Problem formulation

System event logs are a universal resource that exists practically in any system. We use *system event logs* to denote the free-text audit trace generated by the system execution (typically in the `/var/log` folder). A log message or a log record/entry refers to one line in the log file, which is produced by a *log printing statement* in the source code of a user or kernel program running on or inside the system.

Our goal is to parse each log entry e into a collection of *message types* (and the corresponding parameter values). Here each message type in e has a one-to-one mapping with a log printing statement in the source code producing the raw log entry e. For example, a log printing statement:

```
printf("Temperature %s exceeds warning threshold\n", tmp);
```

may produce several log entries such as:

```
Temperature (41C) exceeds warning threshold
```

where the parameter value is `41C`, and the message type is:

```
Temperature * exceeds warning threshold.
```

Formally, a *structured log parser* is defined as follows:

Definition 1 (Structured Log Parser) Given an ordered set of log entries (ordered by timestamps), log$= \{e_1, e_2, \ldots, e_L\}$, that contain m distinct message types produced by m different log printing statements from p different programs, where the values of m and p (and the printing statements and the program source code) are unknown, a structured log parser is to parse log and produce all message types from those m statements.

A structured log parser is the first and foremost step for most automatic and smart log mining and data-driven log analytics solutions, and also a useful and critical step for managing logs in a log management system. Our objective is to design a *streaming* structured log parser such that it makes only one pass over the log and processes each log entry in an online, streaming fashion continuously. Without loss of generality, we assume that the size of each log entry is $O(n)$ words.

B. Related work

Mining interesting patterns from raw system logs has been an active research field for over a decade. Two major efforts in this area include generating features from raw logs to apply various data analytics, e.g. [3], [4], [6], and building execution models from system logs followed by comparing it with future system executions, e.g. [2]. There are also efforts in identifying dependencies from concurrent logs [3], [4], [8].

To achieve effective data-driven log analytics, the first and foremost process is to turn unstructured logs into structured data. Xu et al. [6] used the schema from log printing statements in the original programs' source code to extract message types. In [8], the raw logs are parsed using pre-defined, domain-specific regular expressions. There are efforts to make this process more automatic and more accurate. Fu et al. [2] proposed a method to first cluster log entries using pairwise weighted edit distance, and then perform recursively splitting. IPLoM [9], [15] explored several heuristics to iteratively partition system logs, such as log size and the bipartite relationship between words in the same log message. LogTree [10] utilized the format information of raw logs and applied a tree structure to extract system events from raw logs. LogSig [11] generates system events from textual log messages by searching the most representative message signatures. HELO [13] extracts constants and variables from message bodies, by first using an offline classification step and then performing online clustering based on the template set by the first step. HLAer [16] is a heterogeneous log analysis system which utilizes a hierarchical clustering approach with pairwise log similarity measures to assist log formatting. *All previous structured log parsing methods focus on offline batched processing or matching new log entries with previously offline-extracted message types or regular expressions (e.g., from source code).*

There are also commercial and open source softwares on log management and analysis. Splunk is a leading log management system that offers a suite of solutions to find useful information from machine data. Elastic Stack offers a rich set of open-sourced tools that could gather logs from distributed nodes, and then index, store, for user to query/visualize. All these tools provide interface to parse logs upon their arrival. However, their parsers are based on regular expressions defined by end users. The system itself can only parse very simple and basic structured schema such as timestamp and hostname, while log messages are treated as unstructured text values.

III. Spell: streaming structured log parser

We now present Spell, a streaming structured log parser for system event logs. Since a basic building block for Spell is a longest common subsequence (LCS) algorithm, hence, Spell stands for Streaming structured Parser for Event Logs using LCS. In what follows, we first review the LCS problem.

A. The LCS problem

Suppose Σ is a universe of alphabets (e.g., a-z, 0-9). Given any sequence $\alpha = \{a_1, a_2, ..., a_m\}$, such that $a_i \in \Sigma$ for $1 \leq i \leq m$, a subsequence of α is defined as $\{a_{x_1}, a_{x_2}, \ldots, a_{x_k}\}$, where $\forall x_i$, $x_i \in \mathbb{Z}^+$, and $1 \leq x_1 < x_2 < \cdots < x_k \leq m$. Let $\beta = \{b_1, b_2, ..., b_n\}$ be another sequence such that $b_j \in \Sigma$ for $j \in [1, n]$. A subsequence γ is called a common subsequence of α and β iff it is a subsequence of each. The longest common subsequence (LCS) problem for input sequences α and β is to find longest such γ. For instance, sequence $\{1, 3, 5, 7, 9\}$ and sequence $\{1, 5, 7, 10\}$ yields an LCS of $\{1, 5, 7\}$.

We observe that an LCS-based method can be developed to efficiently and effectively extract message types from raw system logs. This is a seemingly natural idea, yet has not been explored by existing literature. Our key observation is that, if we view the output by a log printing statement (which is a log entry) as a *sequence*, in most log printing statements, the constant that represents a message type often takes a majority part of the sequence and the parameter values take only a small portion. If two log entries are produced by the same log printing statement stat, but only differ by having different parameter values, the LCS of the two sequences is very likely to be the constant in the code stat, implying a message type.

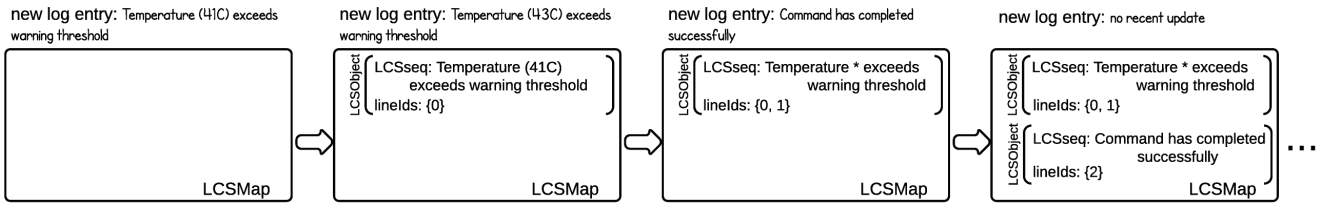

Fig. 1. Basic workflow of Spell.

The merit of using the LCS formulation to parse system event logs, as compared with the previously mentioned clustering and iterative partitioning methods, is that the LCS sequence of two log messages is naturally a message type, which makes streaming log parsing possible.

B. Basic notations and data structure

In a log entry e, we call *each word a token*. A log entry e could be parsed to a set of tokens using system defined (or as user input) delimiters according to the format of the log. In general common delimiters such as space and equal sign are sufficient to cover most cases. After tokenization of a log, each log entry is translated into a "token" sequence, which we will use to compute the longest common subsequence, i.e., $\Sigma = \{\text{tokens from } e_1\} \cap \{\text{tokens from } e_2\} \cdots \cap \{\text{tokens from } e_L\}$. Each log entry is assigned a unique line id which is initialized to 0 and auto-incremented for the arrival of a new log entry.

We create a data structure called *LCSObject* to hold currently parsed LCS sequences and the related metadata information. We use *LCSseq* to denote a sequence that's the LCS of multiple log messages, which, in our setting, is a candidate for the message type of those log entries. That said, each *LCSObject* contains an *LCSseq* and a list of line indices called *lineIds* that stores the line ids for the corresponding log entries that lead to this LCSseq. Finally, we store all currently parsed LCSObjects into a list called *LCSMap*. When a new log entry e_i arrives, we first compare it with all LCSseq's in existing LCSObjects in LCSMap, then based on the results, either insert the line id i to the lineIds of an existing LCSObject, or compute a new LCSObject and insert it into LCSMap.

C. Basic workflow

Our algorithm runs in a streaming fashion, as shown in Figure 1. Initially, the LCSMap list is empty. When a new log entry e_i arrives, it is firstly parsed into a token sequence s_i using a set of delimiters. After that, we compare s_i with the LCSseq's from all LCSObjects in the current LCSMap, to see if s_i "matches" one of the existing LCSseq's (hence, line id i is added to the lineIds of the corresponding LCSObject), or we need to create a new LCSObject for LCSMap.

Get new LCS. Given a new log sequence s produced by the tokenization of a new log entry e, we search through LCSMap. For the ith LCSObject, suppose its LCSseq is q_i, we compute the value ℓ_i, which is the length of the LCS(q_i, s). While searching through the LCSMap, we keep the largest ℓ_i value and the index to the corresponding LCSObject. In the end, if $\ell_j = \max(\ell_i' s)$ is greater than a threshold τ (by default, $\tau = |s|/2$, where $|s|$ denotes the length of a sequence s, i.e., *number of tokens* in a log entry e), we consider the LCSseq q_j and the new log sequence s having the same message type. The intuition is that the LCS of q_j and s is the maximum LCS among all LCSObjects in the LCSMap, and the length of LCS(q_j, s) is at least half the length of s; hence, unless the total length of parameter values in e is more than half of its size, which is very unlikely in practice, the length of LCS(q_j, s) is a good indicator whether the log entries in the jth LCSObject (which share the LCSseq q_j) share the same message type with e or not (which would be LCS(q_j, s)).

If there are multiple LCSObjects having the same max ℓ values, we choose the one with the smallest $|q_j|$ value, since it has a higher set similarity value with s. Then we use *backtracking* to generate a new LCS sequence to represent the message type for all log entries in the jth LCSObject and e. Note when using backtracking to get the new LCSseq of q_j and s, we mark '$\star$' at the places where the two sequences disagree, as the place holders for parameters, and consecutive adjacent '$\star$'s are merged into one '$\star$'. For instance, consider the following two sequences: s = `Command Failed on: node-127` and q_j =`Command Failed on: node-235 node-236`, LCSseq of the two would be: `Command Failed on:` $\star$. Once this is done, we update the LCSseq of the jth LCSObject from q_j to LCS(q_j, s), and add e's line id to the lineIds of the jth LCSObject.

If none of the existing q_i's shares an LCS with s that is at least $|s|/2$ in length, we create a new LCSObject for e in LCSMap, and set its LCSseq as s itself.

This completes the basic procedures in Spell, and most standard logs could be successfully parsed using this method. We further describe how to improve its efficiency.

D. Improvement on efficiency

In this section we show how to achieve nearly optimal time complexity for most incoming log entries (i.e., linear to $|s|$, the number of tokens of a log entry e). In our basic method, when a new log entry arrives, we'll need to compute the length of its LCS with each existing message type. Suppose each log entry is of size $O(n)$ for some small constant n (i.e., $n = |s|$), it takes $O(n^2)$ time to compute LCS of a log entry and an existing message type (using a standard dynamic programming (DP) formulation). Let m' be the number of currently parsed message types in LCSMap. The method in section III-C leads to a time complexity of $O(m' \cdot n^2)$ for each new log entry.

Note that since the number of possible tokens in a complex system could be large, we cannot apply techniques that compute LCS or MLCS efficiently by assuming a limited set of alphabets [17], [18], i.e., by assuming small $|\Sigma|$ values.

A key observation is that, for a vast majority of new log entries (over 99.9% in our evaluation), their message types

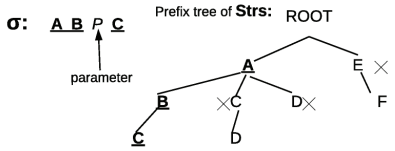

Fig. 2. Find the subsequence of σ using Prefix Tree.

are often already present in currently parsed message types (stored by LCSMap). Hence instead of computing the LCS between a new log entry and each exiting message type, we adopt a pre-filtering step to find if its message type already exists, which reduces to the following problem:

For a new string σ and a set of current strings $strs = \{str_1, str_2, ...str_m\}$, find the longest str_i such that $LCS(\sigma, str_i) = str_i$, and return true if $|str_i| >= \frac{1}{2}|\sigma|$.

In our problem setting, each string is a set of tokens and we simply view each token as a character.

1) Simple loop approach. A naive method is to simply loop through $strs$. For each str_i, maintain a pointer p_i pointing to the head of str_i, and another pointer pt pointing to the head of σ. If the characters (or tokens in our case) pointed to by p_i and pt match, advance both pointers; otherwise only advance pointer pt. When pt has gone to the end of σ, check if p_i has also reached the end of str_i. A pruning can be applied which is to skip str_i if its length is less than $\frac{1}{2}|\sigma|$. The worst time complexity for this approach is $O(m \cdot n)$.

2) Prefix tree approach. To avoid going through the entire $strs$ set, we could index str_is in $strs$ using a prefix tree, and prune away many candidates.

An example is shown in Figure 2 where $strs$ = {*ABC, ACD, AD, EF*}, and they are indexed by a prefix tree T. Instead of checking σ against every str_i in $strs$, we first check tree T and see if there is an existing str_i *that is a subsequence of* σ. If such a str_i is identified, we check if $|str_i| > \frac{1}{2}|\sigma|$). As shown in Figure 2, suppose σ=*ABPC*. Then by comparing each character of σ with each node of T, we could efficiently prune most branches in T, and mark the characters in σ that do not match any node in T as parameters. In this case, we identify *ABC* as the message type for σ, and *P* as its parameter.

For most log entries, it is highly likely that their message type already exists in tree T, so Spell will stop here, and the time complexity is only $O(n)$. This is optimal, since we have to go through every token in a new log entry at least once. However, this approach only guarantees to return a str_i if such str_i = LCS(σ, str_i) exists. It *does not* guarantee that the returned str_i is the longest one among all str_is that satisfy str_i = LCS(σ, str_i). For example, if σ=*DAPBC* while $strs$={*DA, ABC*}, the prefix tree returns *DA* instead of *ABC*.

In practice, we find that although the prefix tree approach does not guarantee to find the longest message type, its returned message type is almost identical to the results of simple loop method. That's because parameters in each log record tend to appear near the end. In fact one of the state-of-art offline methods [2] finds message types by using weighted edit distance and assigns more weight to the token closer to end as parameter position. In particular, the evaluation results show that for the Los Almos HPC log with 433,490 log records, for each new log entry, the message type returned by the prefix tree approach (if found), is 100% equal to the results returned by the simple loop method. But there also exist cases where the returned message type by prefix tree is less than $\frac{1}{2}$ number of tokens ($\frac{1}{2}|s|$) for a new log entry e while e's message type already exists in LCSMap.

That said, the *complete pre-filtering step in* Spell *is*, for each new log entry e, first find its message type using prefix tree, and if not found, apply the simple loop lookup. In evaluation section we'll show that Spell with pre-filtering step produces almost equally good results for all logs with much less cost.

For log entries (less than 0.1% in our evaluation) that do not find message types using the pre-filtering step, we compare the new log entry e with all existing message types to see if a new message type could be generated. However, instead of computing LCS between each message type q and e, we first compute their *set similarity score* using Jaccard similarity. Only for those message types that have more than half common elements (i.e., tokens) with e do we compute their LCS. Then if their LCS length exceeds $\frac{1}{2}|s|$, we adjust that message type and prefix tree T accordingly. Otherwise we simply add e to T and LCSMap as a new message type.

E. Time complexity analysis

Spell ensures that the size of LCSMap increases by one *only when* a new message type is identified; otherwise, a new log entry id is added to an existing LCSObject with an updated message type. This guarantees that LCSMap size is at most the number of total message types (which is m) that could be produced by the corresponding source code, which is a constant. In section III-D, we've shown that for the basic Spell, the time complexity for each new log entry is $O(m \cdot n^2)$, since the naive DP method to compute LCS is $O(n^2)$ for log entries of size $O(n)$, whereas our backtracking method is often cheaper and we only do it with a target message type in LCSMap which has the longest LCS length with respect to the new log entry if the length exceeds a threshold.

With the pre-filtering step, for each log entry, we'll first try to find its message type in prefix tree, then apply simple loop approach, and only for the small portion that are still not located, the LCSMap needs to be compared against. For L log records, suppose the number of log records that fail to find message types in pre-filtering step is F, and the number of log records that are returned in simple loop step is I. The amortized cost for each log record is only $O(n + \frac{(I+F)}{L} \cdot m \cdot n + \frac{F}{L} \cdot m \cdot n^2)$, where m is the number of message types and n is the log record length. In our evaluation, $\frac{(I+F)}{L} < 0.01$ and $\frac{F}{L} < 0.001$, thus the cost for each log record to find its message type in Spell is approximately only $O(n)$ in practice.

F. Remarks

Parsing each log message to extract their message type, though a vital step for many further data analysis, is not an easy task. It should be noted that no automatic approach is perfect for all possible logs. For example, even the approach that extracts log schema from the source code [6] that produces the corresponding log cannot achieve 100% accuracy.

IV. Evaluation

In this section, we evaluate the efficiency and effectiveness of Spell, by comparing it with two popular offline log parsing algorithms, on 2 real log datasets with different formats. All experiments were executed on a Linux machine with an 8-core Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz computer. We'll show that Spell not only is able to parse logs in an online streaming fashion, but also has outperformed the competing offline methods in terms of both efficiency and effectiveness.

The two offline algorithms to be compared are IPLoM [9], [15] and a clustering-based log parser [2] which we refer to as CLP. The idea of IPLoM is to partition the entire log into multiple clusters, where each cluster represents a set of log entries printed by the same print statement. The partition is done using a simple 3-step heuristic: i) partition by each log record length; ii) partition each cluster by the token position having least distinct tokens; iii) partition by the bipartite mapping between tokens in each cluster. It is so far the most lightweight automatic log parsing algorithm. CLP, on the other hand, is a frequently used algorithm by multiple log mining efforts [2], [3], [4]. It also partitions the log into clusters, while by first clustering using weighted edit distance, and then repeatedly partitioning until all clusters satisfy the heuristic - each position either has the same token, or is a parameter position.

TABLE I
PARAMETERS FOR ALL THREE ALGORITHMS

Spell	Value
message type threshold τ	0.5
CLP	**Value**
edit distance weight ν	10
cluster threshold ς	5
private contents threshold ϱ	4

IPLoM	**Value**
file support threshold	0.01
partition support threshold	0
lower bound	0.1
upper bound	0.9
cluster goodness threshold	0.34

Table I shows the default values of key parameters used for each algorithm. For parameters with recommended values that were clearly stated in the original papers, such as all parameters for IPLoM [9], we simply adopt those values. For others that were not clearly specified, we tested the corresponding method with different values until we got the best result (for the same log data) as in the original paper.

We use the supercomputer logs that were commonly used for evaluation by previous work [9], [13], [15], [16], shown in table II (count is the total number of log entries).

TABLE II
LOG DATASETS

Log type	Count	Message type ground truth
Los Alamos HPC log [1]	433,490	available online[2]
BlueGene/L log[1]	4,747,963	available online[3]

A. Efficiency of Spell

Figure 3 shows the total runtime of different methods when log size (the number of log records) grows bigger. Note that we tested different alternatives of the Spell method:

[1]CFDR Data, https://www.usenix.org/cfdr-data

[2]Los Alamos National Lab HPC Log message types, https://web.cs.dal.ca/~makanju/iplom/hpc-clusters.txt

[3]BlueGene/L message types, https://web.cs.dal.ca/~makanju/iplom/bgl-clusters.txt

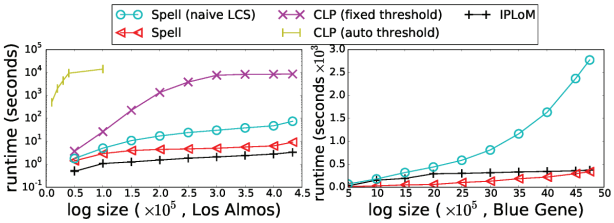

Fig. 3. Efficiency comparison of different methods.

TABLE III
AMORTIZED COST OF EACH MESSAGE TYPE LOOKUP STEP IN Spell

	Los Alamos HPC log	BlueGene/L log
prefix tree (ms)	0.006	0.011
simple loop (ms)	0.020	0.087
naive LCS (ms)	0.175	0.580

- Spell (naive LCS): compute the LCS using DP between new log entry and every existing message type.
- Spell: Spell with the pre-filtering step.

Figure 3 left shows the results on Los Almos HPC Log. *Note that runtime is measured by logarithm scale*. To parse the entire log with $433,490$ entries, Spell with naive LCS is about 75 seconds while it's only 9 seconds with pre-filtering. IPLoM shows the best efficiency, whereas Spell (with pre-filtering) is only slightly slower (within seconds). The CLP method has the worst efficiency (2-4 orders of magnitude slower than IPLoM and Spell). We tested two variants of CLP: 1) CLP (auto threshold): it automatically sets the cluster threshold ς by k-means clustering. When log size is bigger than 100,000, it's already too slow to run to completion. 2) CLP (fixed threshold): it uses a fixed threshold 5 calculated from smaller log file, which significantly improves the runtime. However it's still much slower than other methods. In later experiments we only use CLP with fixed threshold if applicable.

Figure 3 right shows the results on Blue Gene Log. *The runtime in this figure is measured in normal decimal scale.* We didn't include CLP in this experiment: even CLP with fixed threshold is too slow to finish as the Blue Gene log has nearly 5 million entries. Here the advantage of our pre-filtering step is clearly demonstrated. In particular, Spell with pre-filtering has outperformed IPLoM in terms of efficiency. With prefix tree, when the log size grows much faster than the number of message types, most log entries will find a match in prefix tree, and return immediately. Then for the majority of the rest, the message types could be found using simple loop approach. Only for a small amount of log records that are not matched in pre-filtering step, we will compare it with each existing message type. Noticeably, the runtime of Spell (naive LCS) increases exponentially. That's because when log size grows bigger, more message types also show up, and when each new log entry comes, it may need to be compared with a larger number of message types. This result clearly shows the importance of the pre-filtering step and how it has effectively mitigated the efficiency issues in the basic Spell method.

The amortized cost for each log entry to find its message type using different lookup method in the pre-filtering step is shown in Table III (in milliseconds). Recall that for each log entry, Spell first tries to find its message type in prefix tree,

TABLE IV
NUMBER (PERCENTAGE) OF LOG ENTRIES RETURNED BY EACH STEP

	Los Alamos HPC log	BlueGene/L log
prefix tree	397,412 (91.68%)	4,457,719 (93.89%)
simple loop	35,691 (8.23%)	288,254 (6.07%)
naive LCS	387 (0.09%)	1,990 (0.042%)

then simple loop, and finally uses naive LCS if not found in previous two steps. Table IV shows the number (percentage) of log entries that are returned in each step, showing that over 91% could be processed in prefix tree in $O(n)$ time, and over 99.9% in total could be processed by prefix tree and simple loop combined. The expensive naive LCS computation is only applied to less than 0.1% of log entries. Hence much overhead is reduced by pre-filtering step. We'll show later that it provides almost identical results with the costly naive LCS method.

B. Effectiveness of Spell

In this section we evaluate the effectiveness of Spell. After parsing, the log file is processed into multiple clusters, where each cluster represents one message type with the associated log records (as produced by the corresponding log parsing method). A parsed message type is considered as correct if all and only all log records printed by that message type (as identified through the ground truth) are clustered together. We run each method, compare the results with the ground truth generated by matching each log entry with its true message type from Table II, and calculate the accuracy, which indicates the total number of log entries that are parsed to correct message types over the number of total processed log records.

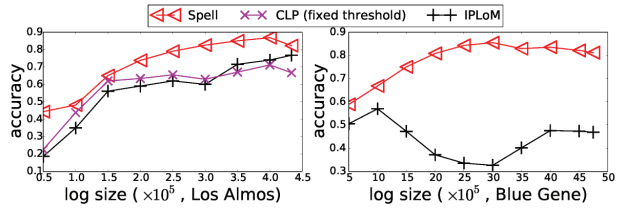

Fig. 4. Effectiveness comparison of different methods.

Figure 4 shows the comparison on supercomputer logs. With more log entries, number of message types also increases; and they don't necessarily show up uniformly over time. Hence, the effectiveness of a method does not necessarily show a steady trend as log size grows. We can see that in both charts, Spell achieves much better accuracy than other methods. IPLoM accuracy is acceptable in Figure 4 left for Los Almos log, and becomes terrible in Figure 4 right for Blue Gene log.

Note that the pre-filtering step in Spell may miss an existing message type t for a new log entry e if $\text{LCS}(t,s) \neq t$ but $\text{LCS}(t,s)| > |\text{LCS}(t',s)|$ when there is another existing message type t' that satisfies t'=$\text{LCS}(t',s)$, where s is the token sequence of e. To evaluate such potential degrade to the effectiveness due to the pre-filtering step, we show a comparison in Table V. The result shows that Spell with pre-filtering has achieved an accuracy nearly the same as that using only naive LCS. This means the pre-filtering step has almost no downgrade effect on the parsing results though it greatly reduces the parsing overhead.

TABLE V
COMPARISON OF Spell WITH AND WITHOUT PRE-FILTER

Spell	Los Alamos HPC log		BlueGene/L log	
With pre-filtering	True message types found	Accuracy	True message types found	Accuracy
False	55	0.822786	165	0.811798
True	55	0.822786	164	0.811791

V. CONCLUSIONS

We present a streaming structured log parser, Spell, for parsing large system event logs in streaming fashion. Spell works perfectly for online system log mining and monitoring. It is also a great addition to modern log management systems to provide end-users concise, real-time understanding of the system states. We propose pre-filtering to improve Spell's efficiency. Experiments over real system logs have clearly demonstrated that Spell has outperformed the state-of-the-art offline methods in terms of both efficiency and effectiveness.

VI. ACKNOWLEDGMENT

Min Du and Feifei Li were supported in part by grants NSF CNS-1314945 and NSF IIS-1251019. We wish to thank all members of the TCloud project and the Flux group for helpful discussion and valuable feedback.

REFERENCES

[1] M. Du and F. Li, "ATOM: Automated tracking, orchestration and monitoring of resource usage in infrastructure as a service systems," in *IEEE BigData*, 2015.
[2] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM*, 2009.
[3] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection." in *USENIX ATC*, 2010.
[4] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *SIGKDD*, 2010.
[5] K. Yamanishi and Y. Maruyama, "Dynamic syslog mining for network failure monitoring," in *SIGKDD*, 2005.
[6] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *SOSP*, 2009.
[7] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *NSDI*, 2012.
[8] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *ICSE*, 2014.
[9] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *SIGKDD*, 2009.
[10] L. Tang and T. Li, "LogTree: A framework for generating system events from raw textual logs," in *ICDM*, 2010.
[11] L. Tang, T. Li, and C.-S. Perng, "LogSig: Generating system events from raw textual logs," in *CIKM*, 2011.
[12] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *ICDM*, 2009.
[13] A. Gainaru, F. Cappello, S. Trausan-Matu, and B. Kramer, "Event log mining tool for large scale hpc systems," in *Euro-Par*, 2011.
[14] Z. Cao, S. Chen, F. Li, M. Wang, and X. S. Wang, "LogKV: Exploiting key-value stores for event log processing," in *CIDR*, 2013.
[15] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A lightweight algorithm for message type extraction in system application logs," *TKDE*, 2012.
[16] H. C. Xia Ning, Geoff Jiang and K. Yoshihira, "HLAer: A system for heterogeneous log analysis," in *SDM Workshop on Heterogeneous Learning*, 2014.
[17] Y. Li, H. Li, T. Duan, S. Wang, Z. Wang, and Y. Cheng, "A real linear and parallel multiple longest common subsequences (mlcs) algorithm," in *SIGKDD*, 2016.
[18] Y. Li, Y. Wang, Z. Zhang, Y. Wang, D. Ma, and J. Huang., "A novel fast and memory efficient parallel mlcs algorithm for longer and large-scale sequences alignments," in *ICDE*, 2016.