# Sequence-RTG: Efficient and Production-Ready Pattern Mining in System Log Messages

Louise Harding, Fabien Wernli, Frédéric Suter

# Sequence-RTG: Efficient and Production-Ready Pattern Mining in System Log Messages

Louise Harding, Fabien Wernli, Frédéric Suter
IN2P3 Computing Centre, CNRS, Villeurbanne, France

***Abstract*—System logs are a wealth of information that can be leveraged to control the behaviour of a computing and storage infrastructure, detect deviations from normal behaviour, and react accordingly by triggering some predefined actions.**

**System log management usually consists of a complex workflow that collects, standardises, indexes, stores, and visualises the log messages to help system administration teams in their daily operations. In large scale data centres such log management infrastructures can collect millions if not billions of messages per day. A key component in this workflow is the identification of message patterns, which requests the expertise of administrators. These patterns represent a template of both static and variable message parts against which a new log message can be matched. This crucial task is often done manually, but these patterns can change frequently making it time consuming for the human operators to keep up.**

**Therefore, we propose in this paper to automate the discovery of patterns in system log messages by extending the functionalities of an existing pattern mining framework, called Sequence. Our main objectives are to improve both the scalability of this framework and its capacity to be integrated into a complete system log management workflow.**

**We present how we addressed six main limitations of the seminal Sequence tool. These modifications led us to propose Sequence-RTG (Sequence-Ready-To-Go), a more efficient and production-ready version. We analyse its performance in terms of both speed, using data-sets of increasing sizes, and accuracy on data sets from the literature. We also show that two months after the introduction of Sequence-RTG within the system log management framework of the IN2P3 Computing Centre we reduced the fraction of messages that are not matched to a pattern from 75-80% to only 15%.**

## I. Introduction

Large scale data centres typically operate thousands of servers to provide scientific communities or companies with computing and storage resources, including complex electrical and climatic equipment to power up and cool down these servers. Additionally multiple software services and full system stacks are run on these machines. This amounts to tens of thousands of individual software and hardware components. Each of these components can inject information about its status, issues, or current activity into system logs in the form of unstructured or loosely structured text messages. These messages can contain numbers, symbols, or other information, such as URLs or IP addresses. Large scale data centres collect millions if not billions of such system log entries every day. This massive amount of data constitutes a wealth of information on the activity of a data centre, provided that it can be efficiently processed.

To manage and exploit information available in logs, system administrators usually deploy a complex infrastructure to first collect system logs in a unified and centralised way by standardising the meta-data supporting these messages [1], e.g. timestamps or server and service that generated the message. Then, these enhanced records are usually stored, indexed, and transformed into comprehensive graphs [2].

An important step is the detection of recurring patterns that allows the infrastructure to either send notifications to system or service administrators, e.g. in the event of a failure or malfunction, or trigger some predefined actions, e.g. restart a service or run an automated diagnostic task. In this context, a pattern is a sequence of meaningful character strings, with both fixed text for static parts and placeholders for variable parts, against which the newly produced unstructured messages injected into the log management system can be matched.

A typical approach is to compare a new log message to a *database of known patterns*. However, the onus of manually adding new patterns to this database is on the administrators. This poses a problem of scalability and maintenance. With each new software update or installation of new software and hardware, new events can appear or existing events potentially change and existing patterns must be frequently reviewed. For instance, at the IN2P3 Computing Centre (CC-IN2P3), one of the largest academic computing centres in France, only 20 to 25% of the log messages were corresponding to an entry in the pattern database before this work. In other words, 75 to 80% of events remained unknown, and potentially contained information important for the system administrators.

Therefore, we aim in this paper at automating the discovery of new patterns and the creation of the corresponding entries into the pattern database. This will allow us to increase the number of log entries that can be matched to a known pattern, which in turn will make searching, filtering, and data analysis much easier. One of the main challenges is to be able to handle the billions of messages generated each year, hence the proposed framework has to be efficient and scalable. It also has to be ready to be integrated into a more complex log management workflow. To this end, we propose *Sequence-RTG* (Sequence-Ready-to-Go), a production-ready and efficient pattern mining tool, based on the seminal Sequence framework [3], [4], which makes the following contributions:

- Enable the ingestion of a stream of composite messages that collates logs from various source systems;
- Improve the quality of the created patterns by adding two message partitioning steps before analysis;

- Make detected patterns persistent by storing them into a database, for reuse between executions;
- Add pattern export and formatting functionalities to ease the interaction with other log management components.

The remainder of this paper is organised as follows. Section II gives an overview of a typical system log management infrastructure deployed at a large scale data centre and details the pattern detection process and its related challenges. Section III details how we extended the Sequence framework to make it usable in production in a large data centre. In Section IV, we analyse its performance in terms of both speed, using data sets of increasing sizes, and accuracy on data sets from the literature and show how the introduction of Sequence-RTG within the existing system log management framework of the CC-IN2P3 drastically reduced the fraction of unmatched messages. Section V reviews the related work on pattern mining in system logs. Finally, Section VI summarises our contributions and presents some future work directions.

## II. Background

Fig. 1 shows the actual workflow of the system log management infrastructure deployed at CC-IN2P3. It starts with the multiple hardware sensors and software services, e.g. operating system, databases, containers, network tools, security, or software applications, that produce systems logs. All these logs are injected into *syslog-ng*, a centralised component in charge of the uniformisation of the logs and the standardisation of their associated metadata [1]. Then, the logs are analysed to identify if they match with known patterns which are stored in a database. When a pattern is recognised as known in the incoming logs, it can trigger a predefined action or, in many cases, it allows a small amount of information to be extracted from the message which is passed with the message to be stored. Otherwise, it passes through as unknown and no automated action can be taken.


Fig. 1. Workflow of the CC-IN2P3's log management infrastructure. It shows a diverse subset of systems sending their log data into syslog-ng, eg, databases, containers etc, which flows onto the pattern database which is maintained by hand. Matched and unmatched messages are sent directly to Elasticsearch.

Human intervention from system or service administrators is often required to identify a new pattern, define its meaning and the related actions it triggers, and add it to the database.

The stream of system logs is then usually indexed and stored for further analysis or forensics, here using the popular *Elasticsearch* framework, which is generally combined with *Kibana* and/or *Grafana* [5] for visualisation purposes [2].

In this paper we focus on the central part of this workflow that lies between the log collection by syslog-ng and the pattern database. Our objective is to automate the detection of patterns to improve the matched/unmatched message ratio and release the burden put on system and service administrators to manage these patterns. For our data centre, at the start of this project the percentage of unknown messages was sitting around 75-80%, with a throughput of 70 million messages per day, this leaves about 56 million messages to be analysed to discover their patterns. Indeed, relying on human identification and creation of new pattern entries for the database leads to only capture a limited number of patterns.

The automatic detection of patterns, or *pattern mining*, usually consists of the three following steps: (i) *tokenisation* breaks a log entry into small logical pieces called tokens; (ii) *analysis* compares the tokens at the same position to ascertain the variable and static parts of the pattern; and (iii) *parsing* identifies which messages belong to each pattern.

One of the key challenges in the automation of pattern mining is that the system logs produced by the multiple servers and services composing an infrastructure are not guaranteed to follow any particular format. They usually comprise textual information, which can be close to a fully formed sentence, often using the English language (but not always). It can also be data contained in formatted JavaScript Object Notation (JSON), or just raw numeric data such as IP addresses, measures of duration, and hexadecimal notation. Moreover, the log management system is generally unaware of the chosen format, as it is usually decided by the service developer. From one service to another the formatting differences can be huge. While initiatives such as the Common Event Expression [6] have suggested a common format of log messages across all systems would be useful, any standard format is far from being agreed on and formalised. However, in spite of the messages having no strict construction rules, there are sequences of text that, in themselves, are always repeatable that can be detected in these logs. Table I lists some of the most common elements found in system logs along with their data types.

The diversity and structure of these components complexify the tokenisation process. The obvious approach for natural language texts is to split by white space, e.g. spaces and tabulations, and punctuation, e.g. commas, full stop, etc. However, these delimiters can be part of elements in log messages, e.g. dates contain spaces, numbers can contain commas or decimal points, MAC addresses include colons. As a result, the tokenisation of log messages requires a more sophisticated approach to correctly identify words and non-words alike.

Once tokens have been extracted from the raw messages, different methods can be applied to analyse them and identify their static and variable parts, such as frequent pattern mining, iterative partitioning, clustering, or longest common sequence. We review these approaches in Section V. Once the patterns

TABLE I
TYPICAL ELEMENTS FOUND IN SYSTEM LOGS AND THEIR DATA TYPES.

| Element | Data Type |
|---|---|
| Date and Time stamps | DateTime |
| MAC addresses | Hexadecimal |
| IPv6 addresses | Hexadecimal |
| Port numbers | Integer |
| Line numbers and counts | Integer |
| Decimal numbers | Float |
| Duration | Text/Number |
| Uids and machine identifiers | Text/Integer |
| IPv4 addresses | Text |
| Words, Brackets, and Quotes | Text |
| Punctuation and control characters | Text |
| Email addresses | Text |
| URLs with/without query strings | Text |
| Host names and Protocols | Text |
| Paths | Text |
| Non-English characters | Text |
| Full SQL request queries | Text |
| Key/value pairs in many formats | Text |

have been discovered, the messages are compared one by one against the set of known patterns to identify which pattern they match and, if relevant, trigger further action or alert.

## III. MAKING SEQUENCE READY TO GO

The proposed production-ready system logs pattern mining framework is based on Sequence [3], which covers all the main steps of pattern mining and parsing. For the tokenisation of the log message, Sequence's scanner uses three finite state machines to determine: (i) hexadecimal tokens; (ii) datetime tokens; and (iii) tokens composed of all of the text and number types. Thanks to these state machines, Sequence can process messages in a single pass which makes it incredibly fast. Moreover, Sequence does not require any prior knowledge of the structure of the log message, nor Regex codes to define parts of the message. The full list of tokens that can be identified at scan time are: Time, IPv4, IPv6, Mac Address, Integer, Float, URL, or Literal.

After tokenisation, the Sequence analyser builds a trie with the tokens. The trie data structure allows for very fast search and retrieval [7]. Once the trie is built it performs a comparison of all of the tokens positioned at the same level that share the same parent and child nodes. During this comparison the relevant parts are merged to produce the patterns. Some other special types are also detected during the analysis phase, i.e. key/value pairs, email addresses, and host names.

Finally, Sequence has its own parser to match new messages against existing known patterns. It follows a similar process as while learning the messages, by first tokenising the messages, but instead of discovering patterns, it attempts to match new messages to a known pattern.

While Sequence appears to solve most of the challenges of the automatic detection of patterns in the system logs of a large data centre, we identified six main limitations that prevent the use of this tool in production:

1) Sequence expects to read from a single file from a single source system. However, most production systems collate the messages together and process them in real time. Sequence-RTG thus needs to handle streamed data coming from a mixture of services.
2) Sequence outputs discovered patterns to a text file that is regenerated each time the analysis is performed. This file is read back in for the parsing step, assuming the analysis is done infrequently. To run a continuous analysis in production, Sequence-RTG needs to collate the output of each execution into a summary database. This collated output is perfect for use during translation and export to another format.
3) Sequence inserts a whitespace between each token regardless of whether they were originally broken by space or not when reconstructing patterns from tokens after analysis [3]. This prevents the use of other parser formats to establish a match between pattern and message.
4) Sequence tends to add too many variables into patterns. Although the pattern works correctly, it can result in redundant meta-data enhancing the log message when it is parsed. Sequence-RTG has to minimise this.
5) Sequence stores its analysis tries in memory and, for very large datasets, it can exceed the allocated memory space. Sequence-RTG thus has to control the size of the dataset or break it into subsets during processing to minimise this risk of terminating the program.
6) Sequence has a documented inability to parse multi-line messages. Sequence-RTG needs to detect such messages and make decisions about how to handle them.

In the remainder of this section, we detail all the different modifications and extensions made to the seminal Sequence framework to implement the proposed Sequence-RTG and address the aforementioned limitations.

*Adding a Data Stream Ingester:* The complex log management systems deployed in large scale data centres aggregate and centralise messages from many source systems into a single stream. Moreover, these messages are often sent in near real time but a batch of messages is required to perform the analysis. To this end, we added a listener for the command line that allows the data to be piped in directly from the log management system without any message pre-processing required and Sequence-RTG waits to execute until the batch size is reached. Each item in the stream is simply expected to be using a JSON format with only two fields: *service* (the source system) from where the message originated and the unaltered log *message*.

Sequence-RTG then waits for a predefined batch size limit to be reached to process the ingested data. This limit is configurable and passed as a command line argument when Sequence-RTG is started. This allows system administrators to select a value adapted to their specific infrastructure. Ideally this number represents a good balance between having enough data to perform the comparison steps of the analysis and preventing a memory overload caused by too many messages.

*Making Patterns and Statistics Persistent:* Analysing system logs in a continuous way requires to be able to preserve patterns between the processing of different message batches. To this end, Sequence-RTG stores the patterns in a SQL database in a one-to-many relationship with their related services. We also include up to three unique examples for each pattern which are used as test cases for the syslog-ng pattern database or to give an insight to the system administrators as to which types of messages will match this pattern.

We label each pattern with a unique ID which is assigned when each message is parsed and matched. It is critical that this ID is not only unique but reproducible for each pattern and service. To achieve this, we compute a SHA1 hash of the concatenated text of the pattern and the service.

Moreover, we attach a set of statistics to the messages matched to each pattern to give a sense of the priority of each pattern for the review and manual promotion by the system administrators. These statistics include the number of times that the pattern has been matched since first discovered (count), how recently it was last matched (last matched date) and a calculated complexity score for the pattern. The complexity score is a guide for the quality of the pattern. For example, patterns that consist entirely of variables with no constant part are often overly patternised, thus increasing their probability of being impractical. This score can then be used to select only the strongest patterns when exporting them for review and integration with other systems.

*Addressing Whitespace Management issues in Tokenisation:* The original pattern reconstruction mechanism of Sequence separates each token by a whitespace regardless of whether the original message had one at that position or not. This makes it impossible to use an external pattern parser that would not be aware of this particular behaviour and thus hinders the integration of Sequence in a complex workflow.

A key addition to the Sequence-RTG scanner is to record the whitespace positions in the original message. We introduced a new token property called `isSpaceBefore`. As each message is scanned, the previous character passed to the scanner is saved and if it is a space, this property is set to true. we leverage this information to ensure the exact reconstruction of the pattern structure when we create it after analysis.

*Improving Quality Control and Memory Management:* Data from a centralised log management system usually come from multiple source systems. To avoid comparing messages from different services and minimise the risk of exceeding the memory, we created an extension of the *Analyze* method called *AnalyzeByService*. Fig. 2 shows the workflow of this new method. It performs a first partitioning of the data which groups the log records into subsets by service and then scans the messages into token sets. These scanned messages are then sent to the Sequence parser to see if they match an already known pattern. If a match is found the last matched date and the number of examples matched to this pattern are adjusted accordingly and no further processing occurs for this message.

Any message for which a match is not found is sent on to the analyser to be mined for new patterns. A second partitioning

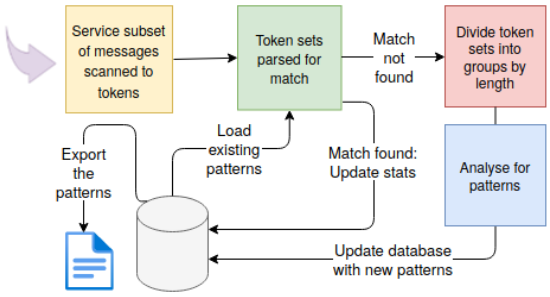


Fig. 2. AnalyzeByService workflow including pattern export.

of these unmatched messages occurs based on count of tokens in the set. Only token sets of the same length are compared in the same analysis trie for pattern discovery. The newly found patterns are eventually saved in the database for comparison against subsequent batches and exporting.

Using this new method and performing the two rounds of partitioning has the added side effect of better quality patterns compared with processing them as a single group.

*Handling Multi-Line Messages Properly:* A documented limitation to the original Sequence scanner is its inability to process multi-line log messages. Indeed, Sequence interprets a line break as the end of a message, resulting in a multi-line message being seen as multiple messages. Thanks to its JSON format Sequence-RTG can process the complete message as one unit. However, this can lead to a new set of challenges. For instance, the longest message we saw has 864 tokens. This introduces a risk that the size of the trie needed to analyse such large messages exceeds the available memory. After reviewing the small number of multi-line messages that occurred in our data centre, we decided to process them only to the first line break, create a pattern only for that first line, and add a marker that instruct the parser to ignore all the remaining text. This provided us with enough information to categorise the messages correctly without having the overhead of processing a very large number of tokens.

*Exporting the Patterns for Other Parsers:* The patterns stored in the database and output by Sequence are clear, simple strings with the variables being delimited by the % character. One example is shown below:

```
%action% from %srcip% port %srcport%
```

While this format is tailored for the Sequence parser, it does not contain enough information to be used in an existing log management system. Therefore we developed a new function (`ExportPatterns`) that can be run on-demand or periodically by system administrators when they want to review patterns. The objective is to be able to either directly use the exported patterns in another component of the log management workflow or allow for a manual review of a selected subset of the patterns deemed to be the most useful or correct. We considered three popular formats for use with two common log

management tools. Fig. 3 shows the same pattern formatted for syslog-ng's pattern database. The transformed Sequence pattern can be found under the pattern tag, but we also include some test cases from the saved examples in the database and the collected statistics. These test cases are used by syslog-ng to ensure that all the example messages match their pattern, and no other in the whole pattern database.

```
- <rule id="2908692bdd6cb4eca096eaa19afebd9e15650b4d">
  - <patterns>
      <pattern>@ESTRING:action: @from @IPvANY:srcip@ port @NUMBER:srcport@</pattern>
    </patterns>
  - <examples>
    - <example>
        <test_message program="sshd">Disconnected from 134.158.106.8 port 41496</test_message>
      - <test_values>
          <test_value name="action">Disconnected</test_value>
          <test_value name="srcip">134.158.106.8</test_value>
          <test_value name="srcport">41496</test_value>
        </test_values>
      </example>
    - <example>
        <test_message program="sshd">Disconnected from 127.0.0.1 port 49570</test_message>
      - <test_values>
          <test_value name="srcip">127.0.0.1</test_value>
          <test_value name="srcport">49570</test_value>
          <test_value name="action">Disconnected</test_value>
        </test_values>
      </example>
    </examples>
  - <values>
      <value name="seq-matches">2</value>
      <value name="seq-new">true</value>
      <value name="seq-created">2019-06-18</value>
      <value name="seq-last-match">2019-06-18</value>
    </values>
</rule>
```

Fig. 3. Fully formatted pattern with test cases for syslog-ng's pattern database. See [8] for more information on the pattern field formats.

We also implemented a YAML version that can be used alongside a DevOps tool such as Puppet to build the pattern database XML. YAML can be easier to use if files are maintained by hand, therefore it may be the preferred format before automation. Finally, we also added the ability to translate the patterns into the Grok format for Logstash as shown in Fig. 4. Selecting the pattern export format is a command-line flag and can be changed by administrators on a per run basis.

```
filter {
   grok {
     match => {"message" => "%{DATA:action} from %{IP:srcip} port %{INT:srcport}"}
     add_tag => ["2908692bdd6cb4eca096eaa19afebd9e15650b4d", "pattern_id"]
   }
}
```

Fig. 4. Pattern formatted for Logstash's Grok [9].

## IV. Evaluation

To evaluate the proposed Sequence-RTG pattern mining tool we start by assessing the performance of the newly introduced *AnalyzeByService* method. Then, we measure the acccuracy of Sequence-RTG on various data sets from the LogHub collection [10] and compare it with the different approaches studied in [11]. We also show how Sequence-RTG can be integrated into a complex log management workflow and reduces the fraction of unmatched messages. Finally, we discuss the current limitations of Sequence-RTG.

*Performance of the AnalyseByService Method:* To ensure that the new *AnalyzeByService* method could easily handle the volume of data produced in an actual production system at a large data centre, we ran some performance tests on a Windows 10 Professional laptop, with one Intel Core i7-6500 2.60GHz CPU and 8GB RAM and a 500GB SSD hard drive using version 1.12 of Go. Note that we excluded the export of the patterns to a file for use with another log management system, as this is unlikely to be the performance bottleneck in a complete log management workflow.

Fig. 5 shows the evolution of the time taken respectively by the Sequence *Analyze* and Sequence-RTG *AnalyzeByService* methods to process data sets of different sizes. The tests were run with an empty pattern database, so all records would be sent for analysis. The rationale is that if a subset were first removed during parsing, the method would complete more quickly. Instead, we want to measure the maximum likely running time in this experiment.

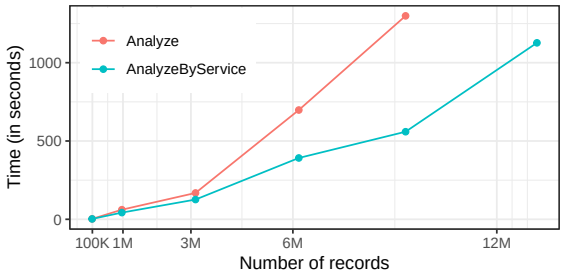


Fig. 5. Evolution of Sequence *Analyze* and Sequence-RTG *AnalyzeByService* processing time with data set size. The datasets contained an average of 241 unique services.

We can see that the Sequence-RTG *AnalyzeByService* method clearly outperforms the original *Analyze* method of Sequence, whose performance starts to degrade for data sets larger than 3 million entries. However, we can also note a similar performance degradation for the *AnalyzeByService* method for for the largest data set comprising 13.25 million log entries. This performance loss is likely due to the load induced by having a very large analyser trie to store in memory. This advocates for splitting the input stream of messages in batches to control the size of the trie, as explained in the previous section, and keep a reasonable processing time. According to these results a batch size of 100,000 messages seems appropriate for a use in production at CC-IN2P3.

*Accuracy:* To evaluate the accuracy of Sequence-RTG in both the discovery of a unique pattern for each type of message, or event, and the subsequent matching of messages to that pattern, we rely on a set of labelled log files from the LogHub collection [10] previously used to compare log parsers [11]. This collection contains log files from 16 different services each with 2,000 entries. It includes both the original system log file and a CSV file with the records pre-processed and labelled by a domain expert, which can be mapped to each other using line numbers [12]. The labelling involved tagging each log message with an event id (e.g. E1, E2, ...) to indicate which pattern or event they represent. We followed the same methodology detailed by Zhu *et al.* [11], who define the related accuracy score as the ratio of correctly matched log messages over the total number of log messages. This is done by evaluating if the event label in the pre-processed file

matches the event determined by the tool under evaluation. By using the event label directly, we ensured comparison remained fair in spite of not following the same pre-processing steps.

Table II shows the achieved accuracy scores for two different versions of log messages. First, we used the same pre-processed version of the logs as in [11], which had been modified to identify field types such as date, numbers, and other common fields that can be difficult to parse, and replace them with a <*> marker. Second, we use the full and unaltered log messages to measure how Sequence-RTG performs on messages coming directly from their production source in a continuous stream.

TABLE II
ACCURACY OF THE SEQUENCE-RTG PARSER USING PRE-PROCESSED DATA AND RAW LOG FILES COMPARED WITH THAT OF THE BEST PARSER FROM [11] FOR EACH DATA-SET. IN BOLD ARE THE SCORES FOR SEQUENCE-RTG THAT EQUALLED OR EXCEEDED THE BEST SCORE.

| Dataset | Pre-processed | Raw Logs | Best |
|---|---|---|---|
| HDFS | 0.941 | 0.942 | 1 |
| Hadoop | **0.975** | 0.898 | 0.957 |
| Spark | 0.979 | 0.979 | 0.994 |
| Zookeeper | **0.971** | **0.977** | 0.967 |
| OpenStack | 0.794 | 0.825 | 0.871 |
| BGL | 0.948 | 0.948 | 0.963 |
| HPC | 0.739 | 0.801 | 0.903 |
| Thunderb. | **0.971** | **0.969** | 0.955 |
| Windows | 0.993 | 0.993 | 0.997 |
| Linux | **0.702** | **0.701** | 0.701 |
| Mac | **0.925** | **0.924** | 0.872 |
| Android | 0.878 | 0.880 | 0.919 |
| HealthApp | **0.968** | 0.689 | 0.822 |
| Apache | **1** | **1** | 1 |
| OpenSSH | **0.975** | **0.975** | 0.925 |
| Proxifier | 0.643 | 0.402 | 0.967 |
| **Average** | **0.901** | **0.869** | 0.865 |

We can see that Sequence-RTG leads to equal or better accuracy than the best algorithm in [11] for 8 out of the 16 considered data-sets. For most of the other data-sets, Sequence-RTG achieves very similar levels of accuracy.

This accuracy is preserved with the use of the raw log data, but for two data-sets which stand out as having a significant accuracy drop: Health App and Proxifier. With the Health App logs, Sequence-RTG was unable to correctly process their datetime stamp which involved time-parts without a leading zero for single digit hour, minute, or second values(e.g. 20171224-0:7:20:444). Proxifier had a variable that was sometimes alphanumeric and sometimes pure integer. This resulted in two patterns created for one event, rendering nearly 50% of the results invalid.

*Integrating Sequence-RTG in a Production Workflow:* Fig. 6 illustrates how Sequence-RTG fits into a production log management workflow: syslog-ng reads, parses, and filters logs from sources and routes them to destinations. When a log has to be written, syslog-ng starts Sequence-RTG (or uses an already running instance) and pipes the log to its standard input. Sequence-RTG is thus a child process of syslog-ng. This service runs a 8-vCPU virtual machine instance hosted on an Intel Xeon Gold 5215@2.50GHz CPU and consumes half the resources of a vCPU on average.

Only the unmatched messages are sent to Sequence-RTG after first being parsed by the pattern database of syslog-ng. Indeed, there is no benefit from mining already known log messages. Moreover, this will lighten the load on Sequence-RTG over time as more patterns are discovered and promoted.

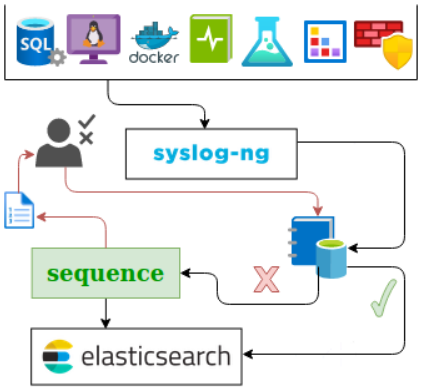


Fig. 6. Log management workflow after the integration of Sequence-RTG. This shows how the workflow has changed from Fig. 1 Matched messages are sent directly to Elasticsearch as before, but the unmatched messages pass through Sequence-RTG for pattern analysis beforehand. Sequence-RTG generates a pattern file for review and promotion.

System administrators are still involved in the review and promotion process of the new patterns to production. However, the overhead of creating these patterns directly from the syslog-ng outputs has virtually disappeared. The only remaining tasks consists in copy-pasting the patterns generated by Sequence-RTG and modify them slightly if need be. Moreover, it allows the system administrators to dedicate some time to add any related actions, such as new alerts during the promotion of patterns. The estimated gain with regard to maintaining the patterns entirely by hand as it was done before the integration of Sequence-RTG is roughly estimated to a day per month for one person.

As an aside, this workflow is how we have implemented Sequence-RTG at the CC-IN2P3. It is possible, however, to use Sequence-RTG as an ad-hoc service that is run only when needed from a file of messages to make patterns to save doing it by hand, or to run from unknown messages once a month to keep on top of system changes. With this in mind, the review process can also be an optional step, for us this is to mitigate cases where the optimal pattern has not been discovered and to ensure actions are added when needed. If a user prefers, this output file from Sequence-RTG can be automatically deployed to the patterndb or Logstash config and used without review.

*Impact of Sequence-RTG on Unmatched Messages:* Implementing the scenario in Fig. 6 at CC-IN2P3 allowed us to run Sequence-RTG in a continuous way while exporting patterns on demand when system administrators had the capacity to review and promote some of them into the pattern database.

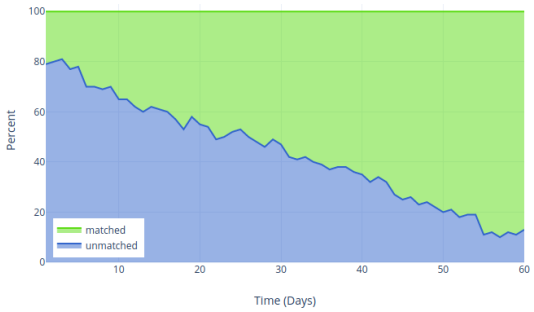


Fig. 7. Evolution of matched/unmatched message ratio after the introduction of Sequence-RTG.

Fig. 7 shows that over 60 days, thanks to Sequence-RTG and with a small investment in time to review the patterns from the system administration team, the percentage of unknown messages dropped down to approximately 15%. We found occasionally, thanks to the pattern database format, that during evaluation with its test cases, they would match more than one pattern. In these instances, the most correct pattern would be promoted and the other discarded.

With a workload oscillating between 70 and 100 million log messages per day and using a batch size of 100,000 records, the average running time of Sequence-RTG for the analysis of messages was of 7.5 seconds over the last year. The wait time between executions of the analysis process was initially of approximately 15 minutes, i.e. the time to get a full batch. As new patterns were promoted, the volume of unknown messages in the input stream automatically decreased over time. As a result, the time to fill a full batch of 100,000 unknown messages and invoking Sequence-RTG to run a new analysis increased to about 25-30 minutes.

This study also showed that a single instance of Sequence-RTG was enough to keep pace with the considered workload. However, if the capacity of Sequence-RTG needed to be scaled up, the messages could be divided simply by sending groups of services to any number instances of Sequence-RTG, thanks to the newly introduced *AnalyzeByService* method. In this case each instance could have its own database as there is no crossover with patterns between different services.

*Limitations:* During our evaluation of Sequence-RTG, we identified three particular types of data that cause problems with the patternisation of the messages. First some path strings are processed correctly but some may remain as static text and generate multiple patterns for a single event. Second, processing the raw logs of the Health App data-set showed the DateTime finite state machine of Sequence cannot correctly detect time stamps where the leading zero on a time part is not present. Third, alphanumeric fields where it is common for the data to be fully numeric in some cases may result in the production of two patterns for the same event. Such fields were present in the Proxifier log with entries of 64 or 64* for the same position in messages representing the same event. If the pattern is being reviewed by a system administrator these can be merged, but otherwise it would result in logging these as separate events.

We also occasionally found log messages that contain fields delimited by the % sign, which Sequence uses to delimit its tokens. If these remain in the pattern as static text, unfortunately they will cause an unknown tag error at parsing time.

Lastly, Sequence-RTG unfortunately struggles to find patterns if only one or two examples of the message is present. In this case the message can become under-tokenised or can be entered as a word for word pattern if no tokens are found during the scan step. This can be monitored by setting a save threshold. Any pattern whose count of matches is less than the threshold is considered useless and thus not saved.

## V. Related work

The accurate and efficient parsing of system log messages has been a challenging area of research over many decades. Many research teams and corporate companies have proposed solutions with varying levels of accuracy and maintenance cost to the administration team. Some commercial systems offer pre-defined patterns for known log file types, often based on Regular Expressions (RegEx) [2], [13] which can lighten the load of maintenance, but for large data centres this is usually not sufficient to cover all the systems in use.

With the growth of text mining and machine learning techniques, several works have proposed that log patterns could be learned or mined, reducing the need for creating these patterns by hand or maintaining a library from known systems. In 2018 Zhu *et al.* compared thirteen log parsers documented in previous research studies over the last fifteen years [11]. They identified six different approaches among the thirteen parsers: frequent pattern mining [14]–[16], heuristics [17], iterative partitioning [18], clustering [19]–[22], longest common sequence [23], parsing tree [24], and an evolutionary algorithm [25]. The most frequently used are frequent pattern mining and clustering.

These different techniques have varied levels of accuracy on logs from diverse systems. Some of these algorithms also need some pre-processing of the log messages to ensure good results. Zhu *et al.* thus pre-processed the logs using some simple RegEx to determine common fields such as IP address, datetime and replace them with <⋆> before passing the messages to the algorithms for pattern discovery. While this does help with the accuracy of the automatic processing, this step requires manual intervention by domain experts, which adds to the maintenance cost of the system.

Once the messages had been categorised by each algorithm, they measured the accuracy using the ratio of correctly parsed log messages over the total number of log messages. Table III, taken from [11], shows these accuracy results for the top four performing algorithms, based on their average score for accuracy across the 16 system log files tested, namely Drain [24], IPLoM [18], AEL [17], and Spell [23].

The Drain algorithm [24] is ranked best overall. It is an online algorithm, i.e. it processes each record, one by one, as they are submitted. After a pre-processing step, the message is tokenised and sent to a fixed depth parsing tree, created from other messages of the same token length, to determine

TABLE III
ACCURACY RESULTS FROM [11] USING PRE-PROCESSED DATA [12] FOR THE TOP FOUR METHODS. FOR EACH DATASET THE MOST ACCURATE ALGORITHM(S) AMONG THE 13 TESTED IS IN BOLD.

| Dataset | AEL | IPLoM | Spell | Drain |
|---|---|---|---|---|
| HDFS | 0.998 | **1** | **1** | 0.998 |
| Hadoop | 0.538 | 0.954 | 0.778 | 0.948 |
| Spark | 0.905 | 0.920 | 0.905 | 0.920 |
| Zookeeper | 0.921 | 0.962 | 0.964 | 0.967 |
| OpenStack | 0.758 | **0.871** | 0.764 | 0.733 |
| BGL | 0.758 | 0.939 | 0.787 | **0.963** |
| HPC | **0.903** | 0.824 | 0.654 | 0.887 |
| Thunderb. | 0.941 | 0.663 | 0.844 | 0.955 |
| Windows | 0.690 | 0.567 | 0.989 | **0.997** |
| Linux | 0.673 | 0.672 | 0.605 | 0.690 |
| Mac | 0.764 | 0.673 | 0.757 | 0.787 |
| Android | 0.682 | 0.712 | **0.919** | 0.911 |
| HealthApp | 0.568 | 0.822 | 0.639 | 0.780 |
| Apache | **1** | **1** | **1** | **1** |
| OpenSSH | 0.538 | 0.802 | 0.554 | 0.788 |
| Proxifier | 0.518 | 0.515 | 0.527 | 0.527 |
| **Average** | **0.754** | **0.777** | **0.751** | **0.865** |

the pattern that it best matches. If no match is found, it adds a new path in the tree.

The second top performer is an iterative partitioning approach to find clusters of similarly formatted messages called IPLoM [18]. After tokenising, the algorithm takes four steps. First, it clusters the token sets that are of the same length, then it builds sub-clusters based on token position. In other words, it looks for a word that is common at the same position of many messages. The third step searches for bijective relationships between two tokens, i.e. where the two values are always the same in their respective positions. The last step is to output the pattern. If all the values at the same position are the same, it is constant in the pattern, if there is a high variation, then it is marked as a variable.

AEL [17] is a log abstraction algorithm made of three steps: Anonymize, Tokenize, and Categorize. The Anonymize step uses simple heuristics to identify variables in the messages defined by text that followed an equal sign or certain keywords. These values are replaced in the log message with a variable marker. The Tokenize method divides the messages into groups based on the count of words and number of variables marked in the text. Finally the Categorize method compares the contents inside each group to determine the patterns.

The online approach followed by Spell [23] performs tokenisation using spaces and the equal sign as stop characters for defining the tokens. For the analysis phase, it uses a longest common subsequence methodology to build a map of the tokens. As with Drain, each new message is tested to see if it matches a pattern already in the map, otherwise a new pattern entry is added.

While the aforementioned tools are promising candidates for solving our challenges, they all require a pre-processing of log messages. This raises real concerns about the maintenance needed to stay ahead of the constant change experienced in our data centre. Conversely, the documentation and early tests with Sequence showed that it could use the raw log messages and still provide robust results. This motivated our choice to to extend this framework with the hope of achieving a level of accuracy in line with the state of the art.

## VI. CONCLUSION

Leveraging the wealth of information contained in system logs can ease the daily operations of system administrators in large data centres by allowing the automatic raising of alerts or triggering of predefined actions. However, this requires identifying the recurring patterns in these logs, a task that often remains manual as it requires some expert knowledge.

To reduce this burden put on humans, we proposed in this paper to extend an existing pattern mining framework called Sequence. We identified and addressed several limitations of this tool to propose Sequence-Ready-To-Go, a more efficient and production-ready version of this framework. One of the main advantages of Sequence-RTG is its capacity to be integrated to a complete log management workflow used in production in a large data centre to assist in the discovery of patterns for keeping the pattern database up to date. Sequence-RTG can also be used as a stand-alone product thanks to its own built-in parser.

Our evaluation of the performance of Sequence-RTG showed it is comparable in accuracy to other methods for system log message parsing available today, while being efficient and able to support a high throughput of data which is critical for a large data centre such as the CC-IN2P3. Once integrated into our log management workflow, Sequence-RTG allowed us to match over 60% of the system log messages generated each day to a pattern, and thus reduce the fraction of messages that are not matched to a pattern from 75-80% to only 15%, with only a small time investment from the system administators to review and promote the patterns.

As future work, we aim at addressing the remaining limitations of Sequence that we listed in Section IV. First, we will have to review and modify the date/time state machine to make it accept single digit time parts. We also would like to implement a fourth finite state machine to deal with the many variations of what can be considered as a "path". Another interesting feature would be to consider tokens that exhibit *semi-constant* values. In other words, tokens for which a variable only takes a few different values. In the current version of Sequence-RTG, a single pattern will be identified. However, it would be more interesting to create as many patterns as there are variations of this semi-constant variable, each pattern having a constant value at its position.

Finally, we plan to go further in the exploitation of system logs and apply statistical and/or machine learning algorithms to the logs to distinguish what could be an anomaly from what is likely to be routine extra load when there are important variations in the number of issued system log entries. The massive volume of logs and the capacity to be integrated into a production workflow will again be important concerns in the development of such an anomaly detection tool.

## AVAILABILITY

Sequence-RTG has been developed in Open Source, under the Apache License, Version 2.0 and is publicly available online on GitHub. To ensure the reproduction and further investigation of the presented code and results, we also prepared an experimental artifact that comprises a copy of the data and notebooks used in the accuracy testing. It includes a folder containing original data from [12], in the event this changes from time of writing. It also contains, for each service, two JSON files, i.e. pre-processed data and full log text, and Jupyter notebooks that show how we evaluate these files, and a CSV file for each service to map Sequence-RTG pattern-ids to the corresponding labels in the original data-set. This artifact is available online on figshare.

## REFERENCES


[1] "Syslog-ng - A Log Management Solution," [Online], available: https://www.syslog-ng.com/products/log-management-software/.

[2] "The ELK stack: Elasticsearch, Logstash, Kibana," [Online], available: https://www.elastic.co/elastic-stack.

[3] "SEQUENCE, High Performance Log Analyzer and Parser," [Online], available: https://sequencer.io.

[4] "Original SEQUENCE code," [Online], available: https://github.com/zentures/sequence.

[5] "Grafana: Dashboard anything. Observe everything." [Online], available: https://grafana.com/oss/grafana/.

[6] "Common Event Expression - A Unified Event Language for Interoperability," [Online], available: https://cee.mitre.org/about/.

[7] E. Fredkin, "Trie Memory," *Communications of the ACM*, vol. 3, no. 9, p. 490–499, Sep. 1960.

[8] "Syslog-ng Administration Guide," [Online], available: https://www.syslog-ng.com/technical-documents/doc/syslog-ng-open-source-edition/3.26/administration-guide/81.

[9] "Grok filter plugin," [Online], available: https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html.

[10] "Loghub: A Collection of System Log Datasets for Intelligent Log Analysis," [Online], available: https://github.com/logpai/loghub.

[11] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and Benchmarks for Automated Log Parsing," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP)*, H. Sharp and M. Whalen, Eds., Montreal, Canada, May 2019, pp. 121–130.

[12] "LogParser: A Toolkit and Benchmarks for Automated Log Parsing," [Online], available: https://github.com/logpai/logparser.

[13] "Splunk: How Splunk Enterprise Handles Your Data," [Online], available: https://docs.splunk.com/Documentation/Splunk/latest/Data/WhatSplunkdoeswithyourdata.

[14] R. Vaarandi, "A Data Clustering Algorithm for Mining Patterns from Event Logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations Management (IPOM)*, Kansas City, MO, 2003, pp. 119–126.

[15] M. Nagappan and M. Vouk, "Abstracting Log Lines to Log Event Types for Mining Software System Logs," in *Proceedings of the 7th International Working Conference on Mining Software Repositories*, Cape Town, South Africa, may 2010, pp. 114–117.

[16] R. Vaarandi and M. Pihelgas, "LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs," in *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, Barcelona, Spain, Nov. 2015, pp. 1–7.

[17] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper)," in *Proceedings of the Eighth International Conference on Quality Software (QSIC)*, H. Zhu, Ed., Oxford, UK, Aug. 2008, pp. 181–186.

[18] A. Makanju, A. Zincir-Heywood, and E. Milios, "Clustering Event Logs Using Iterative Partitioning," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Paris, France, Jun. 2009, pp. 1255–1264.

[19] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," in *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM)*, Miami, FL, Dec. 2009, pp. 149–158.

[20] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "LogMine: Fast Pattern Recognition for Log Analytics," in *Proceedings of the 25th ACM International Conference on Information and Knowledge Management (CIKM)*, Indianapolis, IN, Oct. 2016, pp. 1573–1582.

[21] L. Tang, T. Li, and C. Perng, "LogSig: Generating System Events from Raw Textual Logs," in *Proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM)*, Glasgow, United Kingdom, Oct. 2011, pp. 785–794.

[22] K. Shima, "Length Matters: Clustering System Log Messages using Length of Words," *ArXiv*, vol. abs/1611.03213, 2016.

[23] M. Du and F. Li, "Spell: Streaming Parsing of System Event Logs," in *Proceedings of the 16th IEEE International Conference on Data Mining (ICDM)*, Barcelona, Spain, Dec. 2016, pp. 859–864.

[24] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," in *Proccedings of the 2017 IEEE International Conference on Web Services (ICWS)*, Honolulu, HI, Jun. 2017, pp. 33–40.

[25] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A Search-Based Approach for Accurate Identification of Log Message Formats," in *Proceedings of the 26th Conference on Program Comprehension (ICPC)*, Gothenburg, Sweden, May 2018, pp. 167–177.