MANIPAL
ACADEMY of HIGHER EDUCATION
(Deemed to be University under Section 3 of the UGC Act, 1956)

MANIPAL SCHOOL OF INFORMATION SCIENCES
(A Constituent Unit of MAHE, Manipal)

Log Analytics

Vunet Systems Private Limited

Srinidhi Bhat N
Reg. No 201051016

Master of Engineering
ME (Machine Learning)

Project Start Date: 02/07/2021

Jithesh Kaveetil
Chief Technical Officer
Vunet Systems Pvt. Ltd.
Bangalore

Samarendranath B
Assistant Professor-Senior Scale
MSIS
MAHE, Manipal

Index

Abstract

Log Analysis is the process of reviewing, interpreting, and understanding computer-generated records called logs. Logs are generated by a range of programmable technologies, including networking devices, operating systems, applications, and more. A log consists of a series of messages in a time sequence that describes activities going on within a system. Log files may be streamed to a log collector through an active network, or they may be stored in files for later review. Either way, log analysis is the delicate art of reviewing and interpreting these messages to gain insight into the inner workings of the system.

My work was mainly aiding two components - Automatic Log Parsing and Transaction Monitoring. Automatic Log Parsing included parsing the logs automatically and then giving us the result in the form of templates of all the logs that are parsed. Transaction monitoring includes the processing of logs to understand the flow of a transaction across different logical stages. Both these components collectively have text analytics as their domain, and a core usage of Natural Language Processing and Machine Learning is in play.

Since a log is a collection of words/ sentences that a system outputs for later use, it contains plenty of information that can help us understand the state of underlying system, anomalies, recognize patterns, correlation analysis, understand vulnerabilities, prediction of next event, insights, and prescription for events, and much more, after making the logs structured.

Introduction

Logs are usually the only data resource available that records service runtime information. In general, a log message is a line of text printed by logging statements (e.g., printf(), logging.info()) written by developers. Thus, log analysis techniques, which apply data mining models to get insights into system behaviors, are in widespread use for service management. Most of the data mining models used in these log analysis techniques require structured input. However, raw log messages are usually unstructured, because developers are allowed to write free-text log messages in source code. Thus, the first step of log analysis is log parsing, where unstructured log messages are transformed into structured events.

An unstructured log message, as in the following example, usually contains various forms of system runtime information: timestamp (records the occurring time of an event), verbosity level (indicate the severity level of an event, e.g., INFO), and raw message content (free-text description of a service operation).

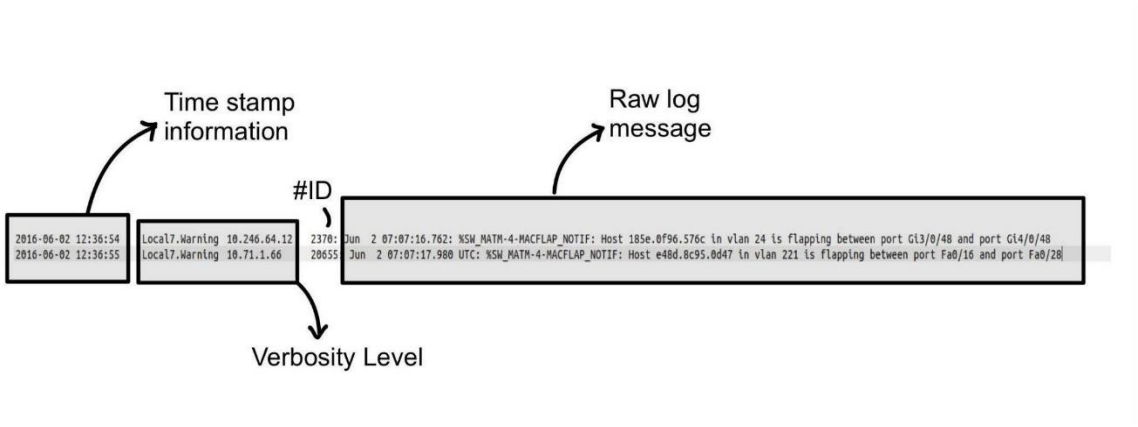

Figure 1: Structure of a Log

Traditionally, log parsing relies heavily on regular expressions, which are designed and maintained manually by developers. However, this manual method is not suitable for logs generated by modern services for the following three reasons. First, the volume of logs is increasing rapidly, which makes the manual method prohibitive. Second, as open-source platforms (e.g., GitHub) and Web services become popular, a system often consists of components written by hundreds of developers globally. Thus, people in charge of the regular expressions may not know the original logging purpose, which makes manual management even harder. Third, logging statements in modern systems are updated frequently. To maintain a correct regular expression set, developers need to check all logging statements regularly, which is tedious and error-prone. Log parsing is thus widely studied to parse the raw log messages automatically.

The goal of log parsing is to transform raw log messages. Specifically, raw log messages are unstructured data, including timestamps and raw message contents. The raw log messages in Figure 2 are simplified HDFS raw log messages collected on the

Amazon EC2 platform. In the parsing process, a parser distinguishes between the constant part and variable part of each raw log message. The constant part is tokens that describe a system operation template (i.e., log event), such as “Receiving block * src: * dest: *” in Figure 2; while the variable part is the remaining tokens (e.g, “blk 3587”) that carry dynamic runtime system information. A typical structured log message contains a matched log event and fields of interest (e.g, the HDFS block ID “blk 3587”). Typical log parsers regard log parsing as a clustering problem, where they cluster raw log messages with the same log event into a log group. The following section introduces our proposed log parser, which clusters the raw log messages into different log groups in a streaming manner. Parsing unstructured messages into structured log messages is described in Figure 2.

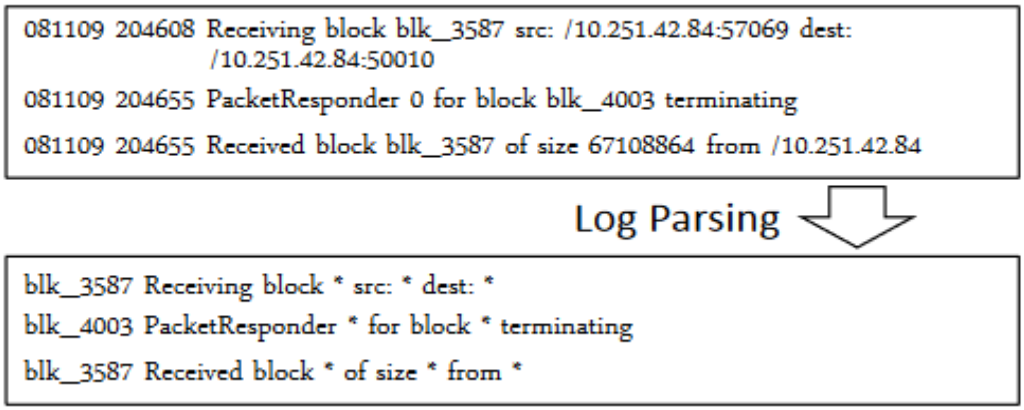

Figure 2: Log Parsing in Action

The nature of the work at the company, in the eyes of my project, can be broadly classified into two foundational concepts - Natural Language Processing and AIOps.

Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data. The goal is a computer capable of "understanding" the contents of documents, including the contextual nuances of the language within them. The technology can then accurately extract information and insights contained in the documents as well as categorize and organize the documents themselves.

Artificial Intelligence for IT Operations (AIOps) is a term coined to be an industry category for machine learning analytics technology that enhances IT operations analytics. Such operation tasks include automation, performance monitoring, and event correlations among others. There are two main aspects of an AIOps platform: machine learning and big data. To collect observational data and engagement data that can be found inside a big data platform and requires a shift away from sectionally segregated IT data, a holistic machine learning, and analytics strategy is implemented against the combined IT data.

Combining both these operations gives us the ability to pipeline the incoming log data, both online as well as saved data (for some time), and eventually provision of insights and analysis on the data. The whole system, when in place, must account for observability with the data provided.

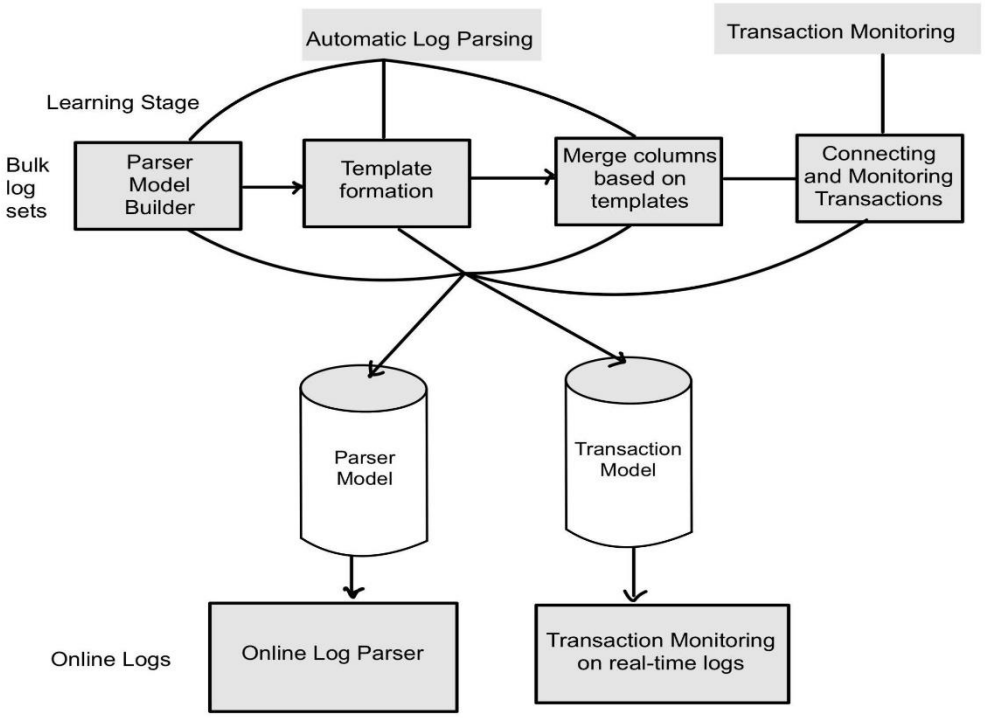

Figure 3: High level Architecture of Log Parsing

Project

My contribution to the project included -

a. Log Parsing
b. Transaction Monitoring
c. Drain: An Online Log Parsing Approach with Fixed Depth Tree
d. Profiling the time & memory metrics

Automatic Log Parsing

The Automatic Log Parser accepts information from the input log file and uses it to cluster the logs and generates a set of string patterns (log templates) found across the log file, which is a summarized report that gives you an idea of what kind of events are captured by the logs in the input log file.

A typical log consists of the date and time stamp, a reference or a serial number, and the actual content of the log. The first step of log parsing is to separate the body from the other information derived from the log. After which, the body of the log will be converted into a template by a parsing machine learning method such as Drain. This parsing method is unsupervised and hence, a template represents the collection of logs similar to each other. A hint of supervision later by categorizing again, on the templates formed by parsing, would get us the final templates.

Transaction Monitoring from parsing logs

Transaction Monitoring helps us understand the various connections between the different transactions of the transaction

lifecycle. This is done by connecting different IDs that appear on the logs of different transactions within the lifecycle. The different IDs are first predicted by regex equations, and this helps in the categorization of the logs as belonging to different aspects of the transactions. Analysis of the categorizations helps in understanding metrics such as time for a transaction to happen, different components used, errors within these components, anomalies and so much more. For example, if a Unified Payment Interface application is considered, the sender types in the value of the money, verifies his passcode, connects to his bank account, the money is transferred securely to the sender, SMS service is delivered to both the parties - the sender and the receiver, and the transaction is complete when the receiver receives it through his bank account. This is an entire transaction, so monitoring of such transactions provides us value.

Automatic Timestamp analysis is done in both the steps of automatic log parsing - prediction from the available logs. Since there are many ways in which a timestamp and date-time can be written, it makes it a difficult proposition for one to manually type in different types of timestamps to detect it later. Preparing an automatic way of this detection and conversion of it into a pattern (regex) that is common to all the logs present is a requirement here.

Overall, the objectives of Log Analytics are -

1. To support the user in understanding the log files
2. To automate the identification

3. To automate log stitching between the logs of the same transaction
4. To create Dashboards that contain insights from the results of log stitching
5. To support for determining the quality of the input logs

Drain: An Online Log Parsing Approach with Fixed Depth Tree

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

Method

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

which effectively bounds the number of log groups that a raw log message needs to compare with.

The parse tree is illustrated in Figure 4. The root node is in the top layer of the parse tree; the bottom layer contains the leaf nodes; other nodes in the tree are internal nodes. Root node and the internal nodes encode specially-designed rules to guide the search process. They do not contain any log groups. Each path in the parse tree ends with a leaf node, which stores a list of log groups, and we only plot one leaf node here for simplicity. Each log group has two parts: log event and log IDs. Log event is the template that best describes the log messages in this

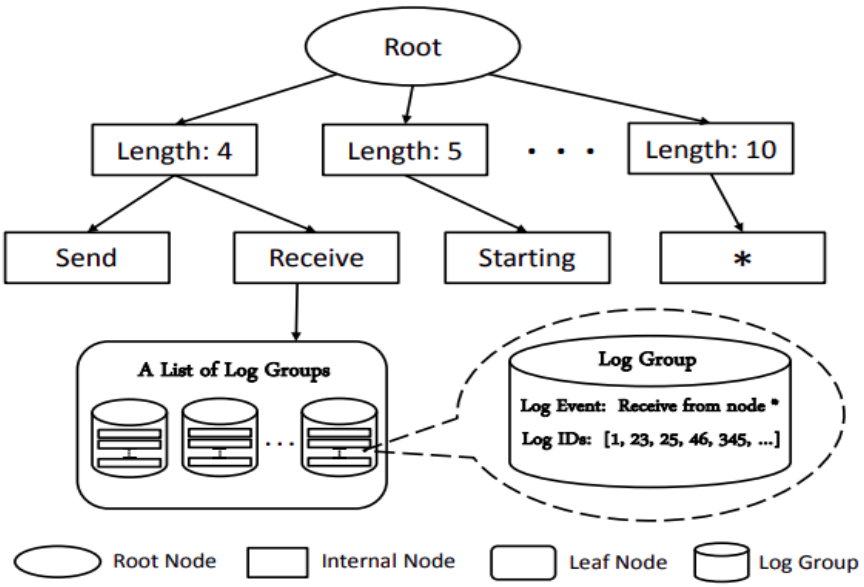

Figure 4: Structure of a Parse Tree in Drain (depth = 3)

group, which consists of the constant part of a log message. Log IDs record the IDs of log messages in this group. One special design of the parse tree is that the depth of all leaf nodes are the same and are fixed by a predefined parameter depth. For example, the depth of the leaf

nodes in Figure 4 is fixed to 3. This parameter bounds the number of nodes. Drain visits during the search process, which greatly improves its efficiency. Besides, to avoid tree branch explosion, we employ a parameter maxChild, which restricts the maximum number of children of a node. In the following, for clarity, we define an n-th layer node as a node whose depth is n. Besides, unless otherwise stated, we use the parse tree in Figure 4 as an example in the following explanation.

B. Step 1: Preprocess by Domain Knowledge

According to our previous empirical study on existing log parsing methods, preprocessing can improve parsing accuracy. Thus, before employing the parse tree, we preprocess the raw log message when it arrives. Specifically, Drain allows users to provide simple regular expressions based on domain knowledge that represent commonly-used variables, such as IP address and block ID. Then Drain will remove the tokens matched from the raw log message by these regular expressions. For example, block IDs in Figure 2 will be removed by “blk [0-9]+”.

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

C. Step 2: Search by Log Message Length

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

Drain starts from the root node of the parse tree with the preprocessed log message. The 1-st layer nodes in the parse tree represent log groups whose log messages are of different log message lengths. By log message length, we mean the number of tokens in a log message. In this step, Drain selects a path to a 1-st layer node based on the log message length of the preprocessed log message. For example, for log message “Receive from node 4”, Drain traverse to the internal node “Length: 4” in Figure 2. This is based on the assumption that log messages with the same log event will probably have the same log message length. Although it is possible that log messages with the same log event have different log message lengths, it can be handled by simple post processing.

D. Step 3: Search by Preceding Tokens

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

to the leaf node linked with the internal node “Receive”, and go to step 4.

The number of internal nodes that Drain traverses in this step is (depth − 2), where depth is the parse tree parameter restricting the depth of all leaf nodes. Thus, there are (depth− 2) layers that encode the first (depth − 2) tokens in the log messages as search rules. In the example above, we can consider the parse tree in for simplicity, whose depth is 3, so we search by only the token in the first position. In practice, Drain can consider more preceding tokens with larger depth settings. Note that if depth is 2, Drain only considers the first layer used by step 2. In some cases, a log message may start with a parameter, for example, “120 bytes received”. These kinds of log messages can lead to branch explosion in the parse tree because each parameter (e.g., 120) will be encoded in an internal node. To avoid branch explosion, we only consider tokens that do not contain digits in this step. If a token contains digits, it will match a special internal node “*”. For example, for the log message above, Drain will traverse to the internal node “*” instead of “120”. Besides, we also define a parameter maxChild, which restricts the maximum number of children of a node. If a node already has maxChild children, any non-matched tokens will match the special internal node “*” among all its children

E. Step 4: Search by Token Similarity

Before this step, Drain has traversed to a leaf node, which contains a list of log groups. The log messages in these log groups comply with

the rules encoded in the internal nodes along the path. For example, the log group in Figure 2 has the log event “Receive from node *”, where the log messages contain 4 tokens and start with the token “Receive”. In this step, Drain selects the most suitable log group from the log group list. We calculate the similarity $simSeq$between the log message and the log event of each log group. $simSeq$is defined as follows:

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

where seq1 and seq2 represent the log message and the log event respectively; seq(i) is the i-th token of the sequence; n is the log message length of the sequences; functionequ is defined as following:

$$equ(t1, t2) = \begin{cases} 1 \; if \; t1 = t2 \\ 0 \; otherwise \end{cases}$$

where $t1$ and $t2$ are two tokens. After finding the log group with the largest $simSeq$, we compare it with a predefined similarity threshold st. If $simSeq \geq st$, Drain returns the group as the most suitable log group. Otherwise, Drain returns a flag to indicate no suitable log group.

F. Step 5: Update the Parse Tree

If a suitable log group is returned in step 4, Drain will add the log ID of the current log message to the log IDs in the returned log group. Besides, the log event in the returned log group will be updated. Specifically, Drain scans the tokens in the same position as the log

message and the log event. If the two tokens are the same, we do not modify the token in that token position. Otherwise, we update the token in that token position by wildcard (i.e., *) in the log event. If Drain cannot find a suitable log group, it creates a new log group based on the current log message, where log IDs contain only the ID of the log message and the log event is exactly the log message. Then, Drain will update the parse tree with the new log group. Intuitively, Drain traverses from the root node to a leaf node that should contain the new log group and adds the missing internal nodes and leaf node accordingly along the path. For example, assume the current parse tree is the tree on the left-hand side of Figure 5, and a new log message “Receive 120 bytes” arrives. Then Drain will update the parse tree to the right-hand side tree in Figure 5. Note that the new internal

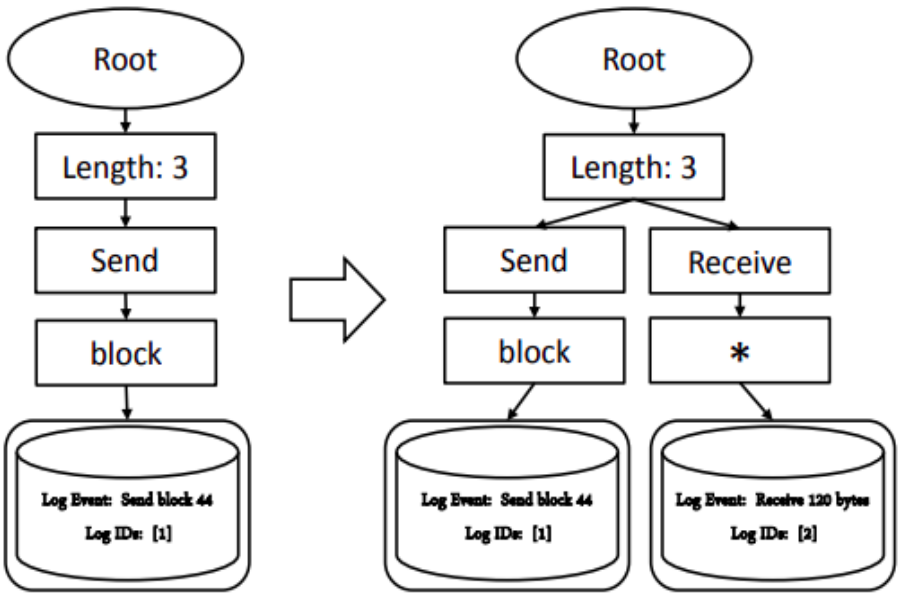

Figure 5: Parse Tree Update Example (depth = 4)

node in the 3-rd layer is encoded as “*” because the token “120” contains digits.

Evaluation Metric and Experimental Setup

We use F-measure, which is a typical evaluation metric for clustering algorithms, to evaluate the accuracy of log parsing methods. The definition of accuracy is as the following.

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

where Precision and Recall are defined as follows:

$$Precision = \frac{TP}{TP + FP}, Recall = \frac{TP}{TP + FN}$$

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

Time profiling using cProfile

cProfile is a built-in python module that can perform profiling. It is the most commonly used profiler currently. The preference to profiling is due to the following reasons-

1. It gives you the total run time taken by the entire code.

2. It also shows the time taken by each step. This allows you to compare and find which parts need optimization
3. cProfile module also tells the number of times certain functions are being called.
4. The data inferred can be exported easily using pstats module.
5. The data can be visualized nicely using snakeviz module.

```
import cProfile
```

cProfile's run() function. The syntax is:

`cProfile.run (statement, filename=None, sort=-1)`.

We can pass python code or a function name that you want to profile as a string to the statement argument. To save the output in a file, it can be passed to the filename argument. The sort argument can be used to specify how the output has to be printed. By default, it is set to -1(no value). Calling cProfile.run() on a simple operation,

```
import numpy as np
cProfile.run("20+10")
```

```
3 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'
```

Figure 6: Function calls in Cprofile

The different components of Cprofile's run function are-

- ncalls : Shows the number of calls made
- tottime: Total time taken by the given function. Note that the time made in calls to sub-functions are excluded.
- percall: Total time / No of calls. (remainder is left out)
- cumtime: Unlike tottime, this includes time spent in this and all sub-functions that the higher-level function calls. It is most useful and is accurate for recursive functions.
- The percall following cumtime is calculated as the quotient of cumtime divided by primitive calls. The primitive calls include all the calls that were not included through recursion.

A best tool available for visualizing data obtained by cProfile module is SnakeViz.

```
#installing Snakeviz
!pip install snakeviz
%load_ext snakeviz
```

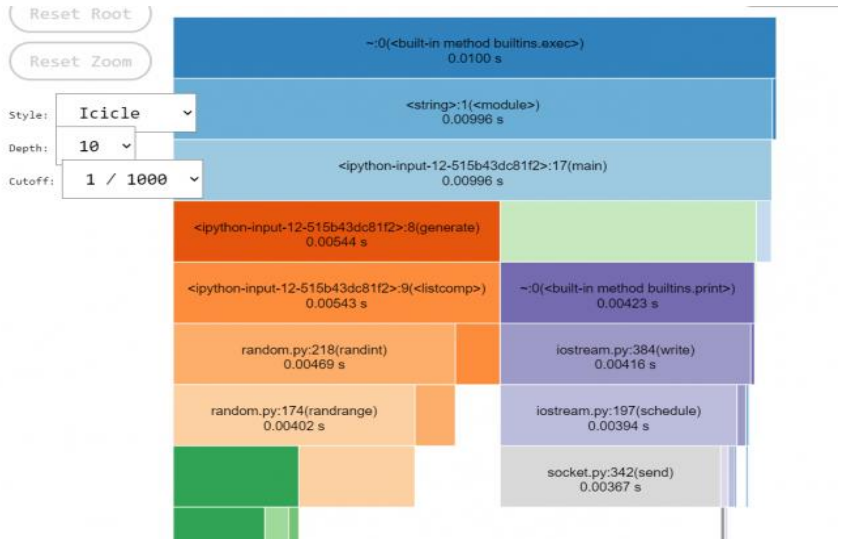

Figure 7: Snakeviz, Style Icicle

```
%snakeviz main()
```

SnakeViz has two visualization styles, ‘icicle’ and ‘sunburst’. By default, it’s an icicle. icicle, the fraction of time taken by a code is represented by the width of the rectangle. Whereas in Sunburst, it is represented by the angular extent of an arc. You can switch between the two styles using the “Style” dropdown. For the same code, Sunburst style visualization is as below –

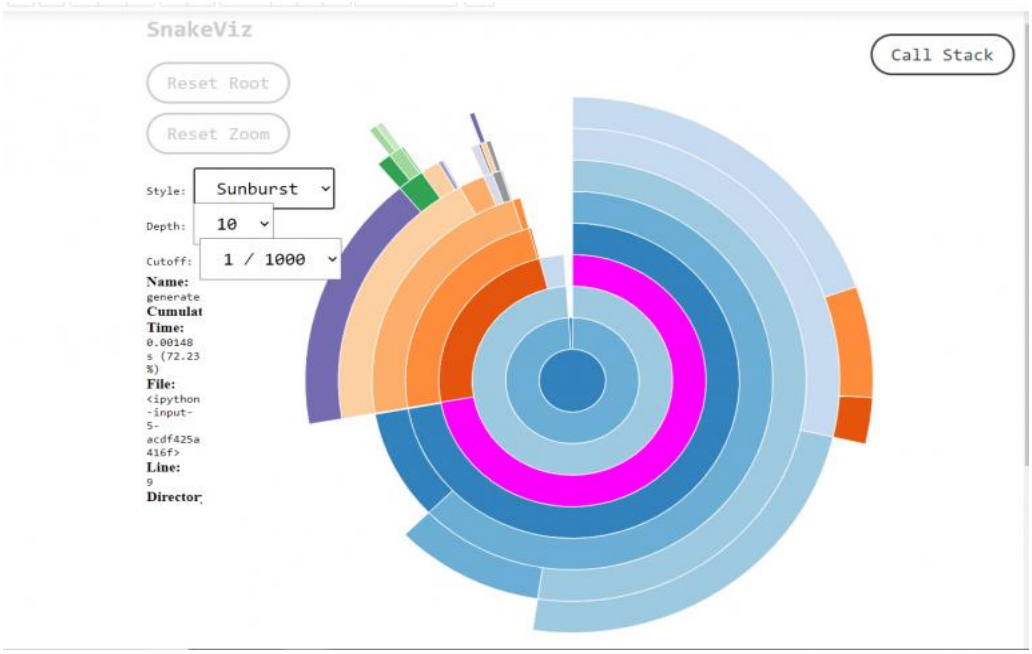

Figure 8: Snakeviz, Style Sunburst

Work done

- Learnt basics of Linux terminal, study on Elasticsearch, Visualizations on Kibana, Advanced topics on Python.
- Understood the overall architecture of how the Log parsing and Transaction Monitoring and aided in making it an optimized code. Also, worked to understand the method of Drain and helped optimize the algorithm such that it is faster to have different types of log files added at once.
- Learned to capture valuable information using regex expressions, and have a good grasp on lexicon analysis which is required for logs.
- Performed tests on different types of logs available to get a hang of how the module performs and the output from the same.
- Time and Memory Profile the codes, along with its visualizations, to understand and perform optimizations where possible.

Proposed work

- Time optimized, online method of parsing the logs. Transaction Monitoring in real-time, i.e., when the logs arrive, it has to be processed and the results are displayed as dashboards in real time.
- Automatic Timestamp Analysis, to gather all kinds of timestamp in popular usage, using regex expressions and datetime module.
- Explore the possibilities using deep learning based NLP techniques in log analysis

Bibliography

- Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu (2017): Drain: An Online Log Parsing Approach with Fixed Depth Tree
  https://jiemingzhu.github.io/pub/pjhe_icws2017.pdf

- NLP, Log Analysis, AIOps (for the introduction page)
  https://en.wikipedia.org/wiki/Natural_language_processing
  https://en.wikipedia.org/wiki/Log_analysis
  https://en.wikipedia.org/wiki/Artificial_Intelligence_for_IT_Operations

- Drain: Log Parsing Documentation
  https://logparser.readthedocs.io/en/latest/tools/Drain.html
- Cprofiling
  https://www.machinelearningplus.com/python/cprofile-how-to-profile-your-python-code/