IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 18, Issue 5, Ver. II (Sept - Oct. 2016), PP 61-64
www.iosrjournals.org

Data Analytics on Application Logs for Managing API-related Dependencies

Sreejita Dutta[1], Subash Prabanantham[2], Abhas Tandon[3], Saumya Garg[4] , Arshia Garg[5]

[1](School of Information Technology/ VIT University, India)
[2](Department of Information Technology/PSG Tech, India)
[3](School of Information Technology / VIT University, India)
[4](School of Information Technology / VIT University, India)
[5](School of Information Technology / VIT University, India)

Abstract: *There are innumerable APIs (Application Programming Interface) that are being created every single day as they serve as a rudimentary tool in developing software applications in today's technology-driven industry. Any organization in this industry could own several APIs and managing them could be a cumbersome task. In this paper, we present an end to end solution that takes the example of one such organization and collects API -related data from application logs to solve two problems – first, maintaining and visualization the hierarchy and dependency between APIs and second, detecting failures and anomalies by efficiently traversing through the complete dependency chain.*

Keywords: *Anomaly Detection, API dependencies, Graph Database, Log Analysis, Time series data analysis*

I. Introduction

In managing APIs, a key problem is to identify the dependencies between the APIs. The large number of APIs present makes it next to impossible to determine their entire hierarchy. Solving this problem will enable any user to identify the upstream and downstream services for a particular API. This helps in efficient management of the APIs. Often times, creating this link between APIs could be difficult to achieve. Each API could be used by a number of other services and it could also be using other APIs. This generates an n-n mapping between APIs.

Also, visualization of such a complex hierarchy between services could be a challenge. Showing the complete chain of services causes information overload for the user and is also unnecessary. It is a rare scenario for one user to be interested in all the APIs. Hence we have applied the concept of single level upstream and downstream traversal of the required API. Knowing and representing the order of APIs in a user-friendly form can also be beneficial to detect failures. If the metrics related to the performance of each API is logged, this information can be utilized to detect any kind of spikes and dips for a particular API.

In this paper, our proposed system grabs API-related data (real-time) from the organization's centralized logging system. From this data, the dependencies between API (i.e. upstream and downstream APIs for each) is generated. This data was leveraged to create a user friendly search UI where the user can easily monitor his service and other services directly dependent on it. He can also deduce which services his API depends on. Furthermore, the API-related data evinced certain thresholds and parameters for each service. This was used to detect anomalies in APIs and the dependencies helped extract the immediate downstream services impacted by failures in a particular service.

II. System Architecture

Fig. 1 describes our system architecture that represents the various modules and components. The logs obtained from various application servers are stored in form of reports in Hadoop and downloadable csv.gz files. Logs containing information apropos to the different APIs are scraped and stored in MongoDB [1]. This information is used to create the hierarchy between APIs in a Graph database - Neo4j [2]. The data stored in MongoDB also contains metrics that can be used by the anomaly detection algorithm to detect failures. The API lineage is visualized in an interactive dashboard and Twitter anomaly detection R- package [3][4] is used to detect spikes and dips in API metrics. Alerts are generated in case of failures.

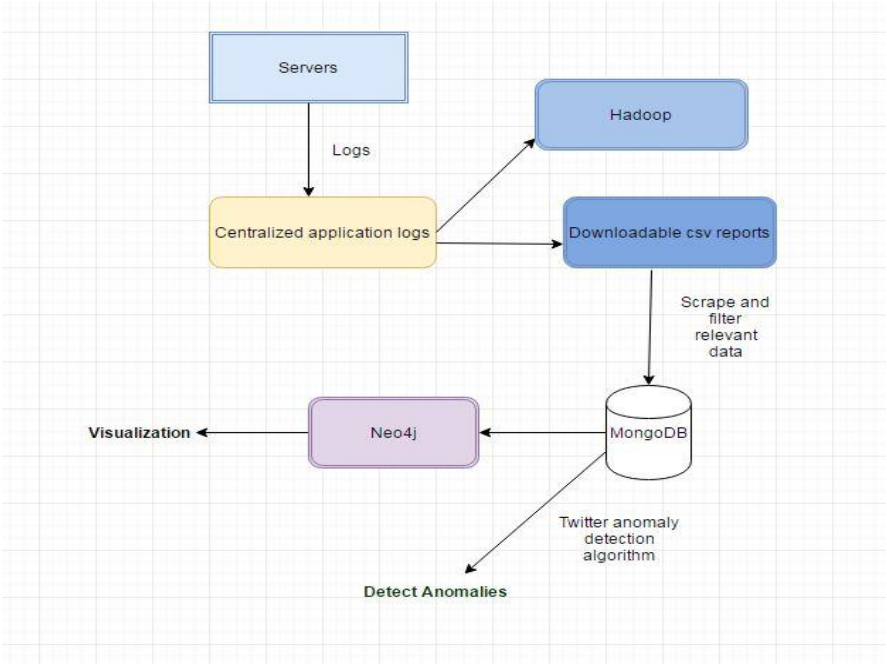

Fig.1

III. Implementation

The data pertinent to the APIs was available in the centralized application logs of the organization. It was available in two forms – through downloadable csv.gz files or reports in Hadoop that can be obtained using Pig Latin scripts [5]. This data reflected two kinds of information:

- Data related to source API which would in turn help in finding the dependency chain between all the relevant APIs
- Metrics related to each API – such as performance parameters, time to connect to each API, no. of times an API was getting hit – which helps in analyzing the performance of APIs and detect anomalies when these metrics are not in their normal range of values.

This source data was gathered from the logs on an hourly basis using Python scripts [6] and stored in a NoSQL database MongoDB. The data stored in MongoDB indicated the source and target API information. Also, a separate collection in MongoDB contained the performance metrics for each API.

A. Neo4j

Representing the complex API dependencies is the most challenging problem. Any user may want to know about its upstream and downstream services. But it will hardly be the case when the user will want to know all the possible dependencies between APIs. This kind of information will only confuse the user. In order to get the entire chain of API dependencies, a graph database – Neo4j – was used.

Graph Database which stores data in the form of graph structures i.e. in terms of nodes, relationships and properties. Neo4j being one of the world's leading open source graph database was preferred by us. It has a Cypher Query Language (CQL). All APIs formed nodes and relationships between them were formed based on the source-target information stored in MongoDB. This created the entire lineage of APIs. Fig. 2 indicates how the relationships were formed between each source API and its target APIs. The direction of the arrow goes from source to target.

Using a graph database helped in maintaining the hierarchy of the APIs and cypher queries were written to identify the upstream and downstream services of each particular API. This data was visualized in an interactive user interface where a user could search for a particular API and get an idea about the services that it is affecting and services getting affected by it.

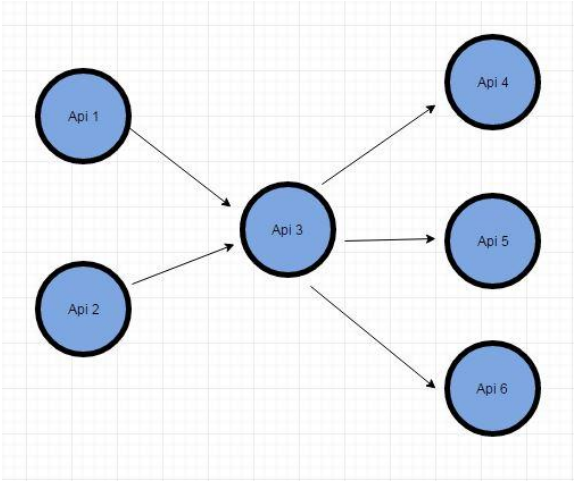

Fig. 2

B. Seasonal Hybrid ESD (S-H-ESD) algorithm

The performance metrics for each API were collected from the logs on an hourly basis. This could be helpful to detect spikes and falls for threshold parameters in order to detect anomalies. Seasonal Hybrid ESD (S-H-ESD) algorithm [7] can be used to detect local anomalies caused by variations in seasonal patterns as well as global anomalies caused by factors that can't be explained by seasonal variations.

We used the open source R-package released by Twitter that is based on this algorithm and can be used for working with time series data. Fig. 3 shows how global anomalies extend beyond normal seasonal variations but local anomalies do not go beyond the natural range of values.

Generalized Extreme Studentized Deviates (ESD) is a statistical procedure that is used for detecting outliers with the assumption that the inliers are normal distributed. The generalized ESD method introduces a procedure for identifying from 1 to k outliers in a dataset simultaneously thus introducing a robustness for the particular number of outliers present. The ESD procedure computes statistics $R_1, \ldots, R_k$ for k data points as the extreme studentized deviates. The first statistic is the largest deviation from the mean,

$$R_1 = \max_i \frac{|x_i - \bar{x}|}{s}, \quad s^2 = \frac{1}{n-1} \sum (x_i - \bar{x})^2$$

The next statistic is calculated on the reduced sample size after removing the sample with the largest deviation, and so on. Critical values λ_i for each test statistic are determined based on a transformed 't' distribution for a specified confidence level. The decision rule specifies that: if all of the test statistics are lower than the critical values, then there are no outliers. If any of the test statistics are greater than the critical value, then the largest number of points such that the associated test statistic is greater than the critical value are removed as outliers. Twitter's R Package [8] is built upon this procedure by working with seasonality and piecewise approximation.

This R-package provides visualization support. The direction of anomalies and the time window (e.g. last day, last hour) can be mentioned by the user. The returned value is a list with the following components:

- Data frame containing index, values, and optionally expected values.
- A graphical object if plotting was requested by the user. The plot contains the estimated anomalies annotated on the input time series.

Hence, this was used for anomaly detection amongst the APIs based on certain thresholds and performance parameters.

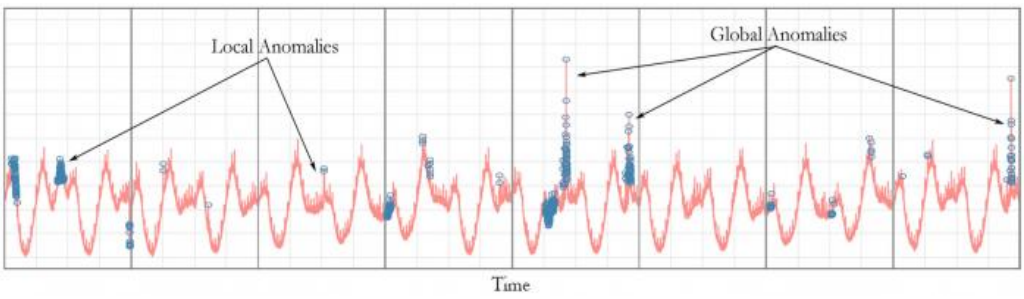

Fig. 3

IV. Results And Future Improvements

Maximum time associated with this solution is spent while scraping the log files. It depends on size of logs as well as programming paradigms used for log analysis. Multithreading can reduce the data collection time significantly. Time to deliver (TTD) is also dependent on how much close to real time are the logs getting generated. In our case, getting the reports from Hadoop via Pig scripts reduced the TTD significantly below 10 minutes.

Overall, the use of Graph DB solves the problem of visualizing the hierarchy of APIs. Also anomaly detection helps to identify faults in services. Once fault is detected for a particular API, the Graph DB helps to identify the immediate downstream services that would get impacted by that API. This could be used to issue alerts and immediately notify concerned teams. The availability and performance of the APIs can be maintained in this way.

The Twitter anomaly detection R-package generally requires two time periods of data i.e two hours of data in our case to start detecting anomalies. Thus in our solution, there was no detection for the first two hours. We proposed a solution for this. In order to identify outliers in the first two hours, instead of using advanced statistical methods, we suggested that box and whisker plots can be used. In a boxplot, an outlier is a data point that is located outside the "whiskers" of the boxplot (for example, beyond 1.5 times the interquartile range above the upper quartile and below the lower quartile). These outliers can be viewed as the anomalies for the first two hours.

V. Conclusion

The goal of managing the API dependencies was achieved. The logs were analyzed and data related to source and target APIs was gathered. This data was used to create the lineage of API dependencies using Neo4j. An interactive search UI was built based on this hierarchical data which allowed upstream and downstream traversal for any API. Also, performance metrics were collected from the log analysis for each API on an hourly basis. Twitter's R package for anomaly detection was used to successfully detect spikes or dips beyond expected range of values. This information was used to generate alerts that identify anomalous APIs and all the downstream services getting affected by it.

References

[1]. K.Chodorow, *MongoDB: The Definitive Guide* (O'Reilly Media,2013).
[2]. H. Huang and Z. Dong, Research on architecture and query performance based on distributed graph database Neo4j, *IEEE CECNet,*, 2013, 10.1109/CECNet.2013.6703387.
[3]. B.Schwartz and P. Jinka, *Anomaly Detection for Monitoring* (O'Reilly Media,2015).
[4]. J. Guzman and B. Poblete, On-line relevant anomaly detection in the Twitter stream: an efficient bursty keyword detection model, *ODD '13 Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description*, 2013, 10.1145/2500853.2500860.
[5]. A.F. Gates, *Programming Pig* (O'Reilly Media,2011).
[6]. W. McKinney, *Python for Data Analysis* (O'Reilly Media,2013).
[7]. S. Kelly and K. Ahmed, Propagating Disaster Warnings on Social and Digital Media, *IDEAL 2015*, 4 (Switzerland: Springer-International Publishing, 2015) 475-484.
[8]. S.K. Ravindran and V. Garg, *Mastering Social Media Mining with R* (Packt Publishing, 2015).