

Big Data and Cloud Computing

A Study & implementation of big data solution for ip scan project

Haihao Yan - 1099966

Table of Contents

[Introduction 2](#_Toc492393245)

[Big data analysis system 3](#_Toc492393246)

[Solution Overview 3](#_Toc492393247)

[Architecture Q&A 5](#_Toc492393248)

[Big Data System 6](#_Toc492393249)

[Assumptions 6](#_Toc492393250)

[Solution Overview 6](#_Toc492393251)

[Data Ingestion 7](#_Toc492393252)

[Batch Layer 7](#_Toc492393253)

[Serving Layer 7](#_Toc492393254)

[Speed Layer 7](#_Toc492393255)

[API Fabric 8](#_Toc492393256)

[Presentation Layer (Dashboard) 8](#_Toc492393257)

[Architecture Q&A 8](#_Toc492393258)

[Reference 11](#_Toc492393259)

[Appendix – 1 Experiment Execution Report 12](#_Toc492393260)

[Q1 – How many records are there in the dataset? 12](#_Toc492393261)

[Q2 – How many records for each country code? List top 10 countries 12](#_Toc492393262)

[Q3 – Percentage of HTTP to Https? 13](#_Toc492393263)

[Q4 – What percentage of servers that have heartbleed bug? 14](#_Toc492393264)

[Q5 – Identify the 10 countries with the biggest percentages of heartbleed bugs 15](#_Toc492393265)

[Q6 – Percentage of Windows vs Ubuntu 16](#_Toc492393266)

[Q7 – Identify the 10 most used cipher suite 17](#_Toc492393267)

[Q8 – Identify the most used software version 18](#_Toc492393268)

[Appendix 2 – Flintrock Configuration 19](#_Toc492393269)

# Introduction

The university of Michigan scans the IP space daily with a variety of tools. It publishes snapshots daily over Alex Top Million website and known X.509 certificates. The dataset provides the technical specification of each server (by IP address) such as the spec for each open port and OS. The size of the dataset is around 900GB of JSON. A big data solution is required to process such volume. This report will present a cloud based big data analysis system and proposing a big data solution to address the key challenges proposed.

The report will be divided into 2 section:

1. To analyze the sample dataset, we implemented a big data analysis platform. In this section of the report, we will present an overview of the implementation as well as demonstrate some of the key concerns behind each design decisions.
2. In the second section, we proposed a big data system for handling this dataset. It will be responsible for generating the regular report and integrate with real-time events. We will present a high-level solution architecture for the system and explain our strategy of handling each proposed challenge.

Answers to each question and execution summary of part 2 will be included in Appendix – 1. Source code can be found on the git repository.

<https://github.com/hyan36/oxclo-assignment>

# Big data analysis system

The Censys dataset [1] is around 900GB and the given sample dataset is about 900MB. Given the fact, the average memory size of a personal computer is around 8GB. It is rather slow to analyze the sample dataset and nearly impractical for processing the full dataset on a single machine. (It is only possible when a programmer implemented a solution which handles memory effectively, however, even that the program will still take days to execute a single query due to the nature of data). Thus, we implemented a cloud based cluster to analyze given dataset.

## Solution Overview

Figure 1 illustrates the architecture overview of this data analysis system. The implementation is based/deployed on AWS.

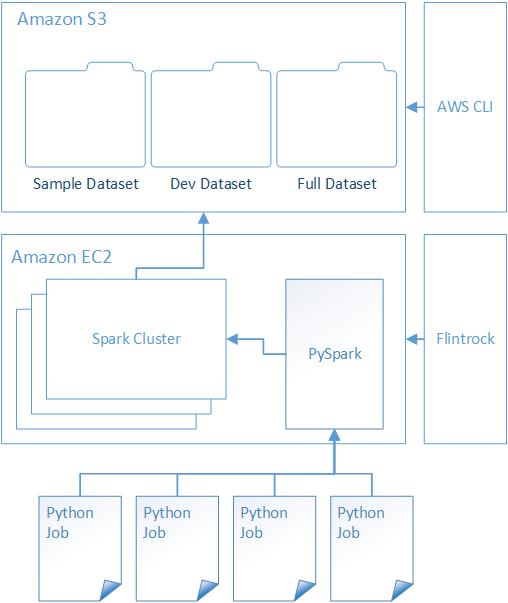


Figure 1 Solution Overview

**Data Repository** (Amazon S3)

We choose Amazon S3 as data repository. There are a few reasons behind this choice.

* Network – the analysis component of the solution is based on AWS. To load the data on to cluster with the lowest latency, S3 is the most reasonable solutions. On the other hand, source data is provided on S3. It is a lot faster to transfer huge dataset within AWS’s internal network.
* Easy to use – S3 provides both GUI and CLI interface to manage it. There is no need to setup another cluster in compare with Hadoop. The overall task required for this solution is very limited.
* Pricing – S3 storage is relatively cheaper compared to other services.

**Data Analysis Cluster** (Flintrock + Spark + EC2)

We use spark cluster to analyze the given dataset. There are a few technologies behind this.

* Flintrock - It is a command-line tool for launching Apache Spark clusters [2]. It provides a fast way for us to manage spark cluster on EC2.
* EC2 – elastic clouds provided a scalable and economical platform for developers to launch experiment / production cluster.
* Spark Cluster – Distributed cluster to process large volume data via MapReduce.
* PySpark - The Spark Python API (PySpark) exposes the Spark programming model to Python [3].

There are a few motivations behind this design.

1. **Performance** – as mentioned previously, the given dataset is nearly impossible/impractical to be processed on a single computer. It needs a distributed system to increase the performance.
2. **Dedication** – Spark is designed to handling large scale MapReduce problem for big data in comparison with MPI (Message Passing Interface) [4] and Globus [5].
3. **Scalability** – Developer can easily add a new/remove slave node to boost the performance of the cluster on EC2. For example, the development cluster for this exercise only contains 5 slave’s nodes while the experiment cluster contains 19 (overall 20 working nodes plus master).
4. **Easy to use** – with the help of tools such as Flintrock, launching/destroying analysis cluster becomes relatively easy.
5. **Economical** – Developer can destroy the whole cluster if no longer used. It is a lot cheaper than buying physical computing equipment.

**Job / Query** (Python)

We use Python/Pyspark to interact with Spark cluster. There are a lot of other options such as Scala/Java/C#. We choose Python due to its popularity and simplicity.

**Cluster Spec**

Here is the key spec of the cluster that has been used for experiments. We attached detail configuration in the appendix.

|  |  |
| --- | --- |
| Name | oxclo-sc |
| Operation System | ami-30041c53 # Amazon Linux, ap-southeast-2 |
| Region | ap-southeast-2 # Sydney |
| Instance Type | m3.medium #1 Core – 4 GB memory |
| No of Slaves | 19 |
| Spark | 2.2.0 |

## Architecture Q&A

1 – **What big data framework/algorithm did we choose?**

As mentioned in the previous section, the system adapts spark as its foundation of processing given dataset. MapReduce is the algorithm backbone of the implementation.

2 – **Which cloud infrastructure?**

We choose to use AWS to hosting our solution due to following reasons:

* **Simplicity** – Launching spark cluster on AWS is relatively easy with the help of developer tools such Flintrocks.
* **Performance** – AWS provides decent infrastructure for data transferring, CPU utilization.
* **Economical** – Developer can switch off the instance when no longer needed.

3 – **How we are going to scale this?**

To answer this question, we need to analyze with the core technologies that have been chosen.

* S3 Bucket – S3 is the on-demand storage services provided by AWS. It scales up naturally if developer is willing to pay.
* Spark + EC2 – Spark cluster solves the scalability issue by enabling adding/removing slave nodes into cluster. The cluster becomes extremely scalable with the help of EC2. A developer can add/remove a new slave instance to the cluster with very minimum amount of work.

4 – **How are we going to efficiently process the data?**

MapReduce is a decent algorithm which solves the performance issue in a distributed system. Spark compiles the code into a distributed execution plan and coordinates the execution automatically. In our example, the lambda calculations for each task are shared across the cluster. Thus, in theory, the more slave nodes that we add into cluster the better performance we will achieve.

5 – **What language did we use to process data?**

We choose Python because of its simplicity in handling data compare to traditional procedure oriented language such as Java / C#. We also choose Python because it has better support for data analysis and AI system. For example, there are more options for Python when dealing with Machine Learning comparing to R or Scala.

# Big Data System

We created a cloud based data analytics engine in the previous chapter. In this chapter, we would extend the platform so it can provide continuous analytics. The new system will need to support following features.

1. Capable of consuming the stream of real-time updated data.
2. Identify changes in the system, E.g. detecting a new port of an IP address
   1. System need to notify customer/dashboard when there is update for the IP addresses
   2. System need to record a summary of all address to represent the most up-to-date status
3. Need to provide to have a dashboard to interact with given report
   1. System needs to be able to retrieve report for current data as well as historical report

## Assumptions

* We have a development team with different skill set, e.g. Python / Java
* We will keep adding new report into the system when there is a new need
* Geolocation of data center is not our concern, we assume network on the cloud works perfectly fine.
* User authentication system such as Active Directory is not a part of this design.

## Solution Overview

Figure 2 illustrates a high-level design of our proposed system. The system follows Lambda architecture to handle massive quantities of data. There are 3 major parts of this architecture, batch layer, speed layer and serving layer.



Figure 2 Solution Overview

### Data Ingestion

As a part of lambda architecture, real-time data will be ingested into two places. We use Kafka [6] to dispatch the event into two systems.

* **Master Dataset** – a complete record of all data has been generated in time. In our design, we decided to use Amazon S3 which has been proven as a cost-effective solution for handling raw data set.
* **Real-time Dataset** – Spark stream [7] framework provides an easy way to processing real-time data and will produce a real-time view.

### Batch Layer

Once data is ingested to the system. We will use our analytical engine from previous section. Data will be aggregated and analyzed via batch process and output into another S3 bucket with formatted/analyzed data.

* **Spark Cluster** – We use spark cluster to provide highly scalable data analytical platform for the system. As explained in the previous chapter, the cluster will be deployed on EC2. This enables the system admin to scale the cluster easily.
* **Batch Process** – A batch process is the script we created for analyzing data. They read the data from the master dataset and publish them into a shareable data repository. In our example, we use Amazon S3 as it is the most convenient option. However, one thing is different from the previous chapter. In our previous design, we choose Python as the main language for batch process. We need to open the options for developers. All we need to standardize is they use Spark cluster and can output result file into a shared data store.

### Serving Layer

Once batched data has been generated. A list of data processors will consume the output and serve them into application data store. Each process contains business logic for a requirement, e.g. heartbleed country percentage report. Triggers of the processes will be determined by actual business needs. All processors need to be containerized as microservices/processes which decrease the difficulty of deployment and development.

* **Data Processor** – self-contained process which extracts the data from one or multiple batched data source and output them into presentation data store (No-SQL database). Each process will be containerized using Dockerwith following behaviors.
  + Extract data from batched data repository
  + Translate Data (In any programming languages)
  + Output data into No-SQL data store
* **Batched / Report Data Store** – A batched data store provides a complete history of reports. The data stored in this database is structured by the report that required in the system.

### Speed Layer

We use Siddhi to provide a view of real-time stream data. It generates a real-time view for incoming data. The output data will be input into two places.

* **Message Queue** – for all changes in data, we will push the change into message queue which will be responsible of notifying client. In our example, we chose Amazon SQS since it is a native out of box solution from AWS server.
* **Real-time Data Store** – this is a no-sql database which provides a snap-shot of most recent status and commonly requested information of each IP as well as the changes of them history up to one month. Certain fields may not be accessible if they are not commonly used. It provides a quick way of user to lookup the information of a specific IP address.

### API Fabric

Business service layer which interact with both Real-time data store and report / batched data store. It can be implemented in any language. However, we would recommend chose Microservice platform such as AWS lambda.

### Presentation Layer (Dashboard)

It can use any web based technology such as PHP + MySQL + Nginx. It interacts with API layer and listen to the message queue.

## Architecture Q&A

**How data is ingested into system and fed around?**

The incoming message from real-world is handling by Kafka. As explained previously, it will distribute the message into two part: Master dataset and Stream data processing engine.

**How current data is compared with historical data?**

In our system, we define current data store as the data which is commonly used and reflect the status of an IP for now. It is mainly for user interaction purpose. However, historical data exists solely for reporting purpose. There might be multiple data store for different report.

**How to add a new datasets and queries to the system?**

Following item need to be added into system

1. **Batched Process** – a new set of processes for analyzing this dataset need to be added into system
2. **Data Processor** – a new processor need to be introduced to translate the batched view to presentation data store (no-sql)
3. **Real-time View** – a new process need to be added to Spark Stream to provide realtime view of the new dataset

**How a live dashboard can be built that displays data from analytics?**

A live dashboard is a web application which interacts with API fabric. All reporting data will be expose to live dashboard with microservices. Each service may query from both current / report data store. The data from report data store is fed by batch process while the data from real-time data store is provided by stream analysis cluster (Spark Stream).

**How the system is deployed, maintained and monitored?**

There are two major parts need to be discussed to answer the question.

* **Infrastructure** – infrastructure indicates the cluster software such as Spark, Cassandra etc. It also means the infrastructure to host the services such as EC2. The frequency of deployment of infrastructure is low. Ideally, we will develop script based system to easily add / remove instance of cluster.
  1. **Spark** – Spark provides a web interface to manage the cluster. This needed to be managed manually by internal system admin team.
  2. **Cassandra** – We will use Cassandra Administration tool to manage the cluster [8] This need to be managed by internal system admin team.
  3. **EC2** – We can be monitored via default web interface via AWS. Physical infrastructure will be managed by Amazon.
  4. **S3** – We can monitor via AWS portal. Physical infrastructure will be managed by Amazon.
  5. **Amazon Lambda** – Microservices framework will be managed by AWS.
* **Processes** – processes will be regularly updated, deployed. They need to be developed and deployed whenever there is a business requirement. The source code will be managed via bitbucket and get deployed via CICD process. They will most likely be managed / owner by the process owner / developer.
  1. **Batch Process** – Python/Scala/R script which can be submit to spark cluster. CICD server will watch the updates of each process and update the production process whenever needed.
  2. **Data Processor –** Pre-compiled docker containers which translate data from batched process result to data store. Developer will write script to enable CICD server to compile the solution and build the docker image and publish to docker repository. Process/Services will be triggered per business requirement via a common scheduler.
  3. **Microservices –** CICD server will automatically build / deploy the service to AWS Lambda via automation script.
  4. **Dashboard** – Traditional web development / deployment. We will containerize the application. However, we won’t discuss this item in detail because it is not the focus of this report.

**How system is scaled?**

Most part of the platform can be scaled automatically.

* Spark cluster – By adding / removing slave nodes
* Cassandra – Can be scaled up / down in run time
* EC2 – Enabling adding / removing instance with reasonable costs
* S3 – Virtualized solution can scale automatically
* Amazon Lambda – Virtualized cloud fabric can be scaled automatically
* AWS SQS – Cloud based solution can be scaled automatically
* Kafka – Distributed solution designed for scaling problem
* Data Processors – Can be add / removed with help of EC2 services per business needs

**How system is secured**

Our system is mostly based on AWS platform. Thus, most communication occurs within the cloud infrastructure (internal network). They would be secured by either SSH tunneling or AWS certificate (for those who interacts with AWS services).

|  |  |  |
| --- | --- | --- |
| System | Security Strategy | Description |
| Spark | Internal network | Only internal system can interact with Spark |
| S3 | AWS KEY Pair |  |
| Kafka | Internal network |  |
| Cassandra | Hardware: Open port  Software: Password/Username | External system can access Cassandra via Port 7199 with correct acces |
| Data Processors | Docker instance, protected by provide docker repository |  |
| API Fabric | Open to public  With proper OAUTH token |  |
| Dashboard | Port 80 / Port 443 (SSH)  Protected via SSH keys | Open port 80 web interface to public  Open port 443 to access the box via SSH  Infrastructure protected via SSH keys |

# Reference

[1] Internet-Wide Scan Data Repository <https://scans.io/>

[2] Flintrock <https://github.com/nchammas/flintrock>

[3] PySpark <https://spark.apache.org/docs/0.9.0/python-programming-guide.html>

[4] MPI <https://en.wikipedia.org/wiki/Message_Passing_Interface>

[5] Globus Toolkit <http://toolkit.globus.org/toolkit/about.html>

[6] Kafka <https://kafka.apache.org/>

[7] Spark Streaming <https://spark.apache.org/streaming/>

[8] Cassandra Administration Tools <https://wiki.apache.org/cassandra/Administration%20Tools>

[9] Shane Johnson, January 13, 2015, Lambda Architecture with NoSQL, <https://blog.couchbase.com/lamda-architecture-and-beyond-with-nosql/>

# Appendix – 1 Experiment Execution Report

All code is executed on the cluster with 19 slave nodes inside.

## Q1 – How many records are there in the dataset?

**Answer**

151613

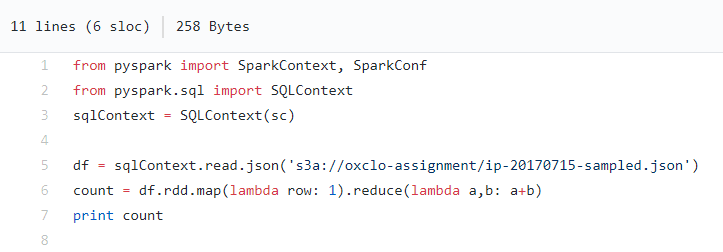
**Analysis**

We will create a map with a counter and reduce after execution

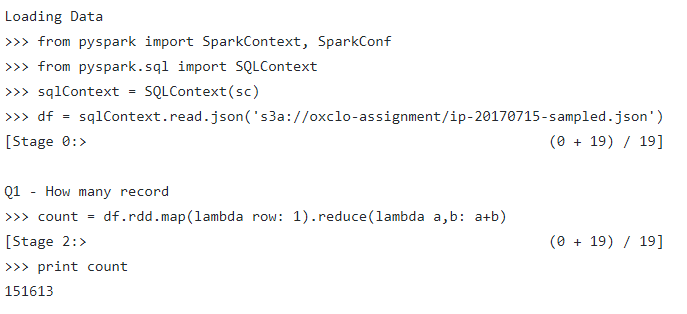
**Source Code**

<https://github.com/hyan36/oxclo-assignment/blob/master/assignment-q1.py>

**Code Snapshot**



**Execution Result**



## Q2 – How many records for each country code? List top 10 countries

**Answer**

US, CN, MX, DE, GB, KR, RU, IT, BR, IN

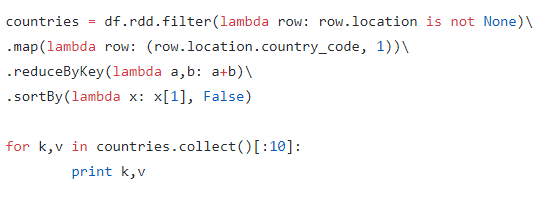
**Analysis**

We need to filter countries which doesn’t have a location object and create a map with country code and counter. We then sort the result by count and get top 10 result in the list.

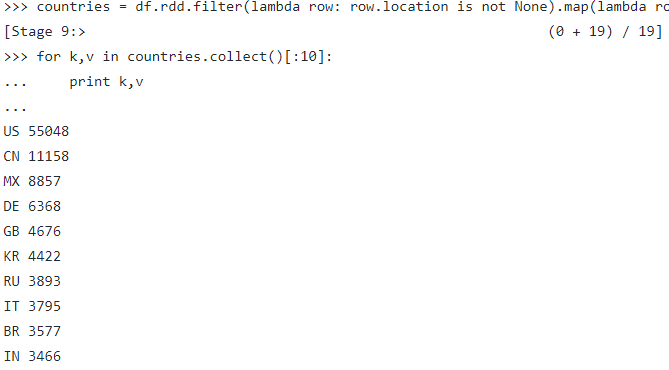
**Source Code**

<https://github.com/hyan36/oxclo-assignment/blob/master/assignment-q2.py>

**Code Snapshot**

****

**Execution Result**



## Q3 – Percentage of HTTP to Https?

**Answer**

|  |  |
| --- | --- |
| Protocol | Percentage |
| Https | 53.92% |
| Http | 35.30% |
| Both | 20.39% |

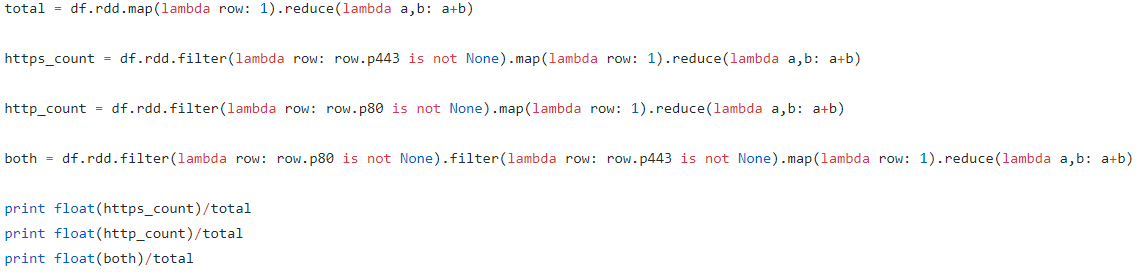
**Analysis**

Like Q2, we need to count the total number of all records, https, http and record which contains both.

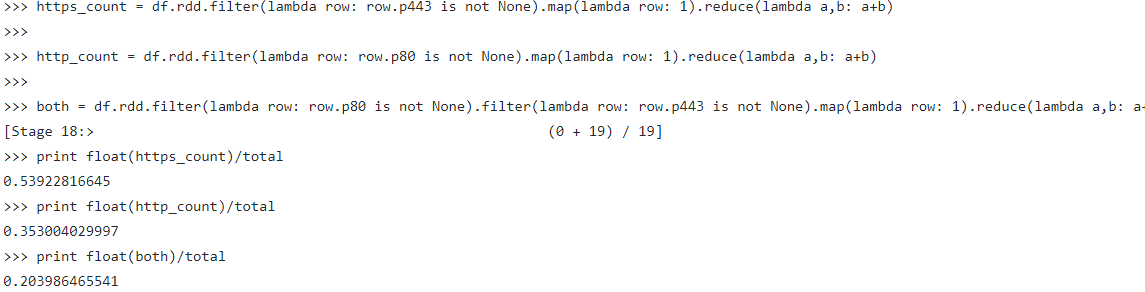
**Source Code**

<https://github.com/hyan36/oxclo-assignment/blob/master/assignment-q3.py>

**Code Snapshot**



**Execution Result**



## Q4 – What percentage of servers that have heartbleed bug?

**Answer**

|  |  |
| --- | --- |
| Protocol | Percentage |
| Https | 0.08% |

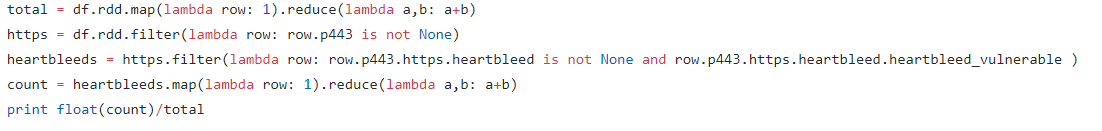
**Analysis**

We need to count the total number of server than get the total number of heartbleed instance

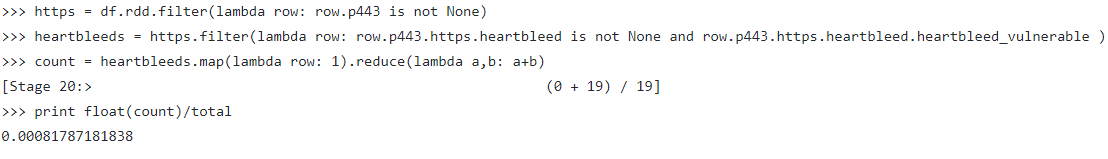
**Source Code**

<https://github.com/hyan36/oxclo-assignment/blob/master/assignment-q4.py>

**Code Snapshot**



**Execution Result**



## Q5 – Identify the 10 countries with the biggest percentages of heartbleed bugs

**Answer**

|  |  |
| --- | --- |
| Country | Percentage |
| HN | 25% |
| BD | 4.7% |
| TT | 4.3% |
| LU | 2.17% |
| IL | 1.9% |
| MD | 1.75% |
| HU | 1.70% |
| PH | 1.28% |
| AE | 1.13% |
| CZ | 0.9% |

**Analysis**

We will create three map.

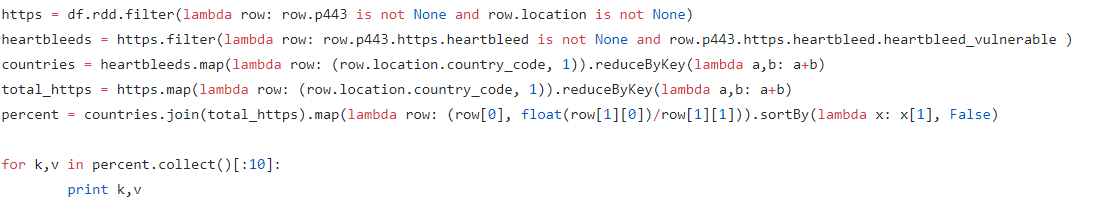
1. A map of total count of https server for each country
2. A map of total count of https server who has heartbleed issue of each country
3. Join 1 and 2, reduce them by key, generate a new set of data which calculates the percentage

Sort the map 3 and pick top 10 from the list

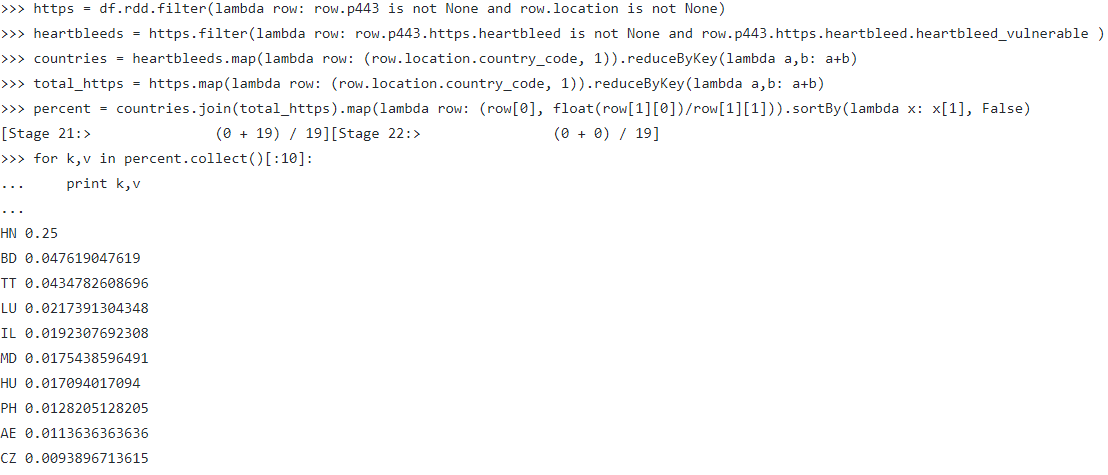
**Source Code**

<https://github.com/hyan36/oxclo-assignment/blob/master/assignment-q5.py>

**Code Snapshot**



**Execution Result**



## Q6 – Percentage of Windows vs Ubuntu

**Answer**

|  |  |
| --- | --- |
| OS | Percentage |
| Windows | 21.02% |
| Ubuntu | 21.14% |

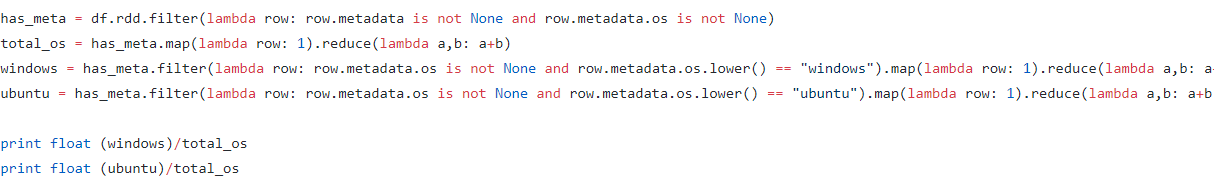
**Analysis**

1. Count Total record which contains OS information
2. Count Windows
3. Count Ubuntu

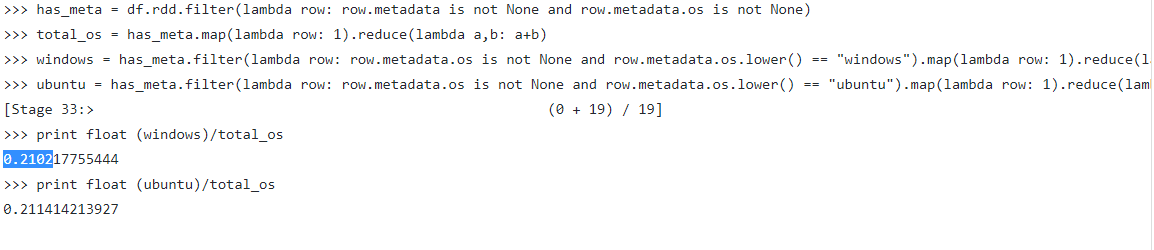
**Source Code**

<https://github.com/hyan36/oxclo-assignment/blob/master/assignment-q6.py>

**Code Snapshot**



**Execution Result**



## Q7 – Identify the 10 most used cipher suite

**Answer**

|  |  |
| --- | --- |
| Country | Percentage |
| TLS\_ECDHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256 | 17906 |
| TLS\_RSA\_WITH\_AES\_256\_CBC\_SHA | 6053 |
| TLS\_RSA\_WITH\_AES\_128\_GCM\_SHA256 | 4523 |
| TLS\_ECDHE\_RSA\_WITH\_AES\_256\_CBC\_SHA | 4092 |
| TLS\_DHE\_RSA\_WITH\_AES\_256\_CBC\_SHA | 2874 |
| TLS\_ECDHE\_ECDSA\_WITH\_AES\_128\_GCM\_SHA256 | 1205 |
| TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA | 1170 |
| TLS\_RSA\_WITH\_RC4\_128\_SHA | 1083 |
| TLS\_DHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256 | 674 |
| TLS\_RSA\_WITH\_RC4\_128\_MD5 | 652 |

**Analysis**

We need to filter the result based on following conditions

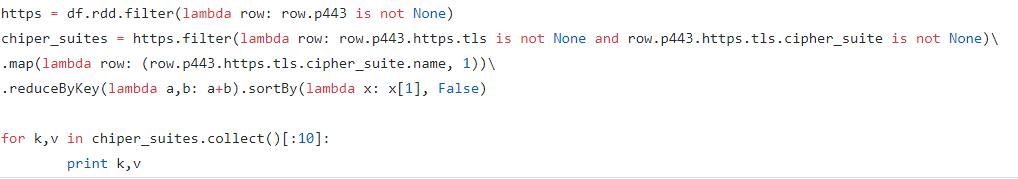
1. A filter to get all https record
2. A filter to remove all record who doesn’t have cipher\_suite attribute

Then create a map with cipher suite name and counter.

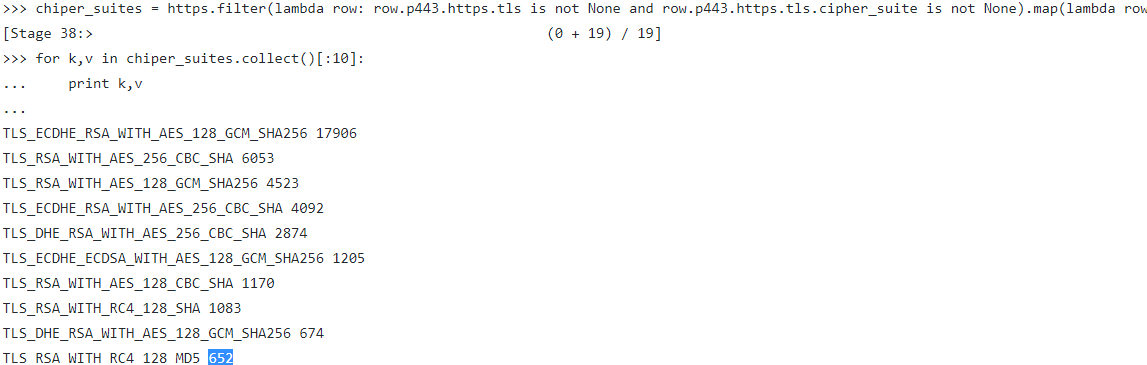
**Source Code**

<https://github.com/hyan36/oxclo-assignment/blob/master/assignment-q7.py>

**Code Snapshot**



**Execution Result**



## Q8 – Identify the most used software version

**Answer**

|  |  |
| --- | --- |
| Software | Version |
| OpenSSH\_5.3 | 2.0 |

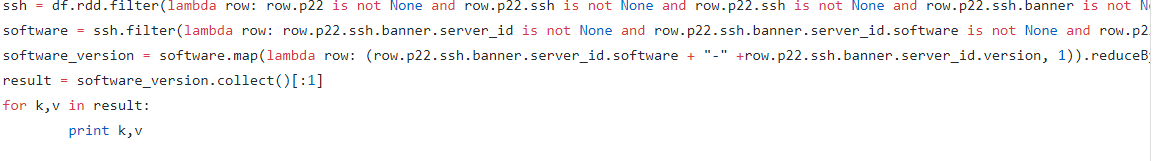
**Analysis**

1. Create a data frame with all ssh server
2. Filter those who doesn’t have software information
3. Create a map based on software name + it’s version number (because a software may have different version)
4. Sort it and get the first result

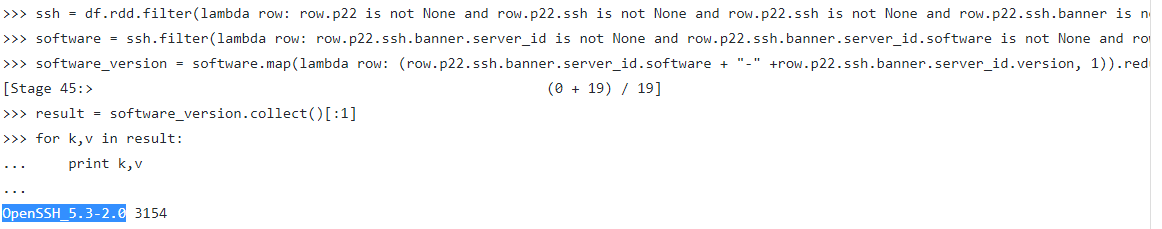
**Source Code**

<https://github.com/hyan36/oxclo-assignment/blob/master/assignment-q8.py>

**Code Snapshot**



**Execution Result**



# Appendix 2 – Flintrock Configuration

