Experiment 10

Shashwat Shah
60004220126
TY Btech Comps B

Aim: Perform Sentiment analysis using Kafka

Theory: Data streaming is the process of transmitting a continuous flow of data (also known as streams) typically fed into stream processing software to derive valuable insights.

Apache Kafka is an open source, distributed streaming platform that enables the development of real-time, event driven applications.

Components of Apache Kafka
- → Producers
- → Brokers
- → Topics & Partitions
- → Replicas
- → Consumers
- → Followers & Leaders

Producers - Producers in Kafka publish messages to one or more topics.

Brokers - A Kafka cluster comprises one or more servers that are known as brokers. Brokers works as containers that can hold multiple topics.

Topics - A stream of message that are a part of a specific category or feed name is referred to as a kafka topic

Partitions - Topics in kafka are divided into a configurable number of parts, which are known as parameter.

Replicas - Replicas are like backups for partitions in ~~kafka~~ kafka

Leader & followers - Every partition will have one server that plays the role of a leader for that partition Leave will perform read and write operations. Followers will replicate the data of the leaders.

Conclusion: Thus, we performed sentiment analysis using Apache Kafka.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

EXPERIMENT NO. 10
Build sentiment analytics application using Spark Streaming

Name: Shashwat Shah

SAP ID: 60004220126

Batch: C2_2

AIM: Case Study to build sentiment analytics application using Spark Streaming.

THEORY:

Build Log Analytics Application using Apache Spark

Why Apache Spark Architecture if we have Hadoop?

The Hadoop Distributed File System (HDFS), which stores files in a Hadoop-native format and parallelizes them across a cluster, and applies MapReduce the algorithm that actually processes the data in parallel. The catch here is Data Nodes are stored on disk and processing has to happen in Memory. Thus we need to do lot of I/O operations to process and also Network transfer operations happen to transfer data across the data nodes. These operations in all may be a hindrance for faster processing of data.

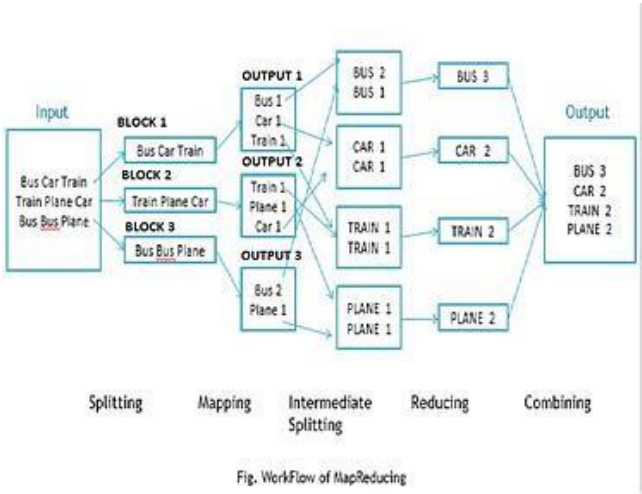
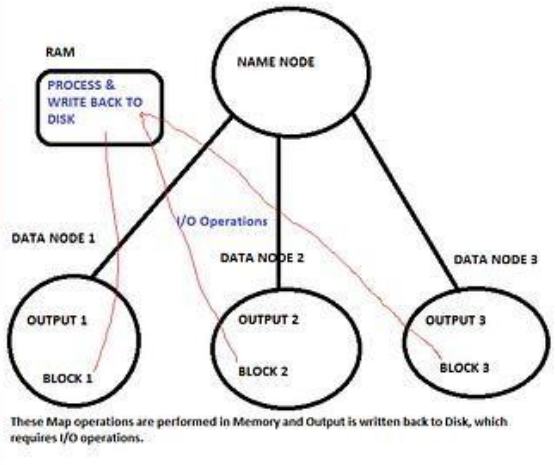


Fig. WorkFlow of MapReducing

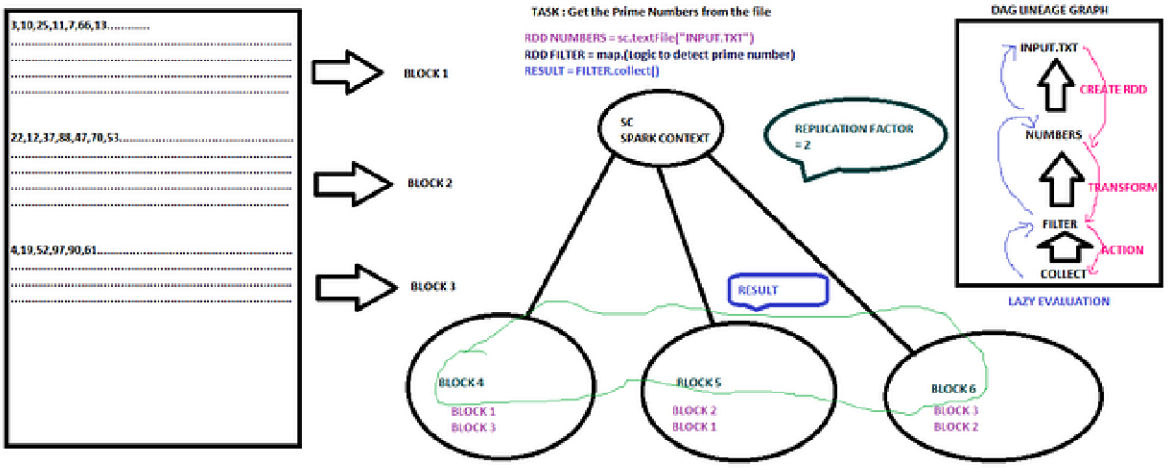


Above image describes, blocks are stored on data notes which reside on disk and for Map operation or other processing has to happen in RAM. This requires to and fro I/O Operation which causes a delay in overall result.

Apache Spark: Official website describes it as : "Apache Spark is a **fast** and **general-purpose** cluster computing system".

Fast: Apache spark is fast because computations are carried out in memory and stored there. Thus there is no picture of I/O operations as discussed in Hadoop architecture.

General-Purpose: It is an optimized engine that supports general execution graphs. It also supports a rich SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming for live data processing.



Entry point to Spark is Spark Context which handles the executors nodes. The main abstraction data structure of Spark is Resilient Distributed Dataset (RDD), which represents an immutable collection of elements that can be operated on in parallel.

Lets discuss the above example to understand better: A file consists of numbers, task is find the prime numbers from this huge chunk of numbers. If we divide them into three blocks B1,B2,B3. These blocks are immutable are stored in Memory by spark. Here the replication factor=2, thus we can see that a copy of other node is stored in corresponding other partitions. This makes it to have a fault-tolerant architecture.

Step 1 : Create RDD using Spark Context

Step 2 : Tranformation: When a map() operation is applied on these RDD, new blocks i.e B4, B5, B6 get created as new RDD's which are immutable again. This all operations happen in Memory. Note: B1,B2,B3 still exist as original.

Step 3: Action: When collect(), this when the actual results are collected and returned.

LAZY EVALUATION: Spark does not evaluate each transformation right away, but instead batch them together and evaluate all at once. At its core, it optimizes the query execution by planning out the sequence of computation and skipping potentially unnecessary steps.

Main Advantages : Increases Manageability**,** Saves Computation and increases Speed, Reduces Complexities, Optimization.

How it works ? When it we execute the code to create Spark Context, then create RDD using sc, then perform tranformation using map to create new RDD. In actual these operations are not executed in backend, rather a **Directed Acyclic Graph(DAG) Lineage** is created. Only when the **action** is performed i.e. to fetch results, example : **collect()** operation is called then it refers to DAG and climbs up to get the results, refer the figure, as climbing up it sees that filter RDD is not yet created, it climbs up to get upper results and finally reverse calculates to get the exact results.

RDD — **Resilient :** i.e. fault-tolerant with the help of RDD lineage graph. RDD's are a deterministic function of their input. This plus immutability also means the RDD's parts can be recreated at any time. This makes caching, sharing and replication easy.

Distributed : Data resides on multiple nodes.

Datasets : Represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

In this experiment, we will create a Apache Access Log Analytics Application from scratch using **pyspark and SQL** functionality of Apache Spark. Python3 and latest version of pyspark.

Data Source: **ApacheAccessLog**

Prerequisite Libraries

```
pip install pyspark
pip install matplotlib
pip install numpy
```

Step 1 : As the Log Data is unstructured, we parse and create a structure from each line, which will in turn become each row while analysis.

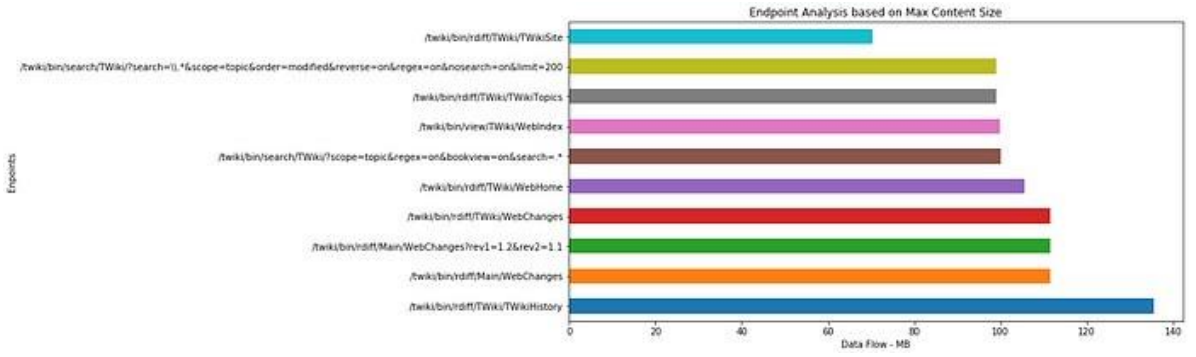
```
import re
from pyspark.sql import Row
# This is the regex which is specific to Apache Access Logs parsing, which can be modified according to different Log formats as per the need
# Example Apache log line:
#   127.0.0.1 - - [21/Jul/2014:9:55:27 -0800] "GET /home.html HTTP/1.1" 200 2048
#   1:IP  2:client 3:user 4:date time           5:method 6:req 7:proto   8:respcode 9:size
APACHE_ACCESS_LOG_PATTERN = '^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+) (\S+)" (\d{3}) (\d+)'

# The below function is modelled specific to Apache Access Logs Model, which can be modified as per needs to different Logs format
# Returns a dictionary containing the parts of the Apache Access Log.
def parse_apache_log_line(logline):
    match = re.search(APACHE_ACCESS_LOG_PATTERN, logline)
    if match is None:
        raise Error("Invalid logline: %s" % logline)
    return Row(
        ip_address    = match.group(1),
        client_identd = match.group(2),
        user_id       = match.group(3),
        date = (match.group(4)[:-6]).split(":", 1)[0],
        time = (match.group(4)[:-6]).split(":", 1)[1],
        method        = match.group(5),
        endpoint      = match.group(6),
        protocol      = match.group(7),
        response_code = int(match.group(8)),
        content_size  = int(match.group(9))
    )
```

Step 2: Create Spark Context, SQL Context, DataFrame (is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database)

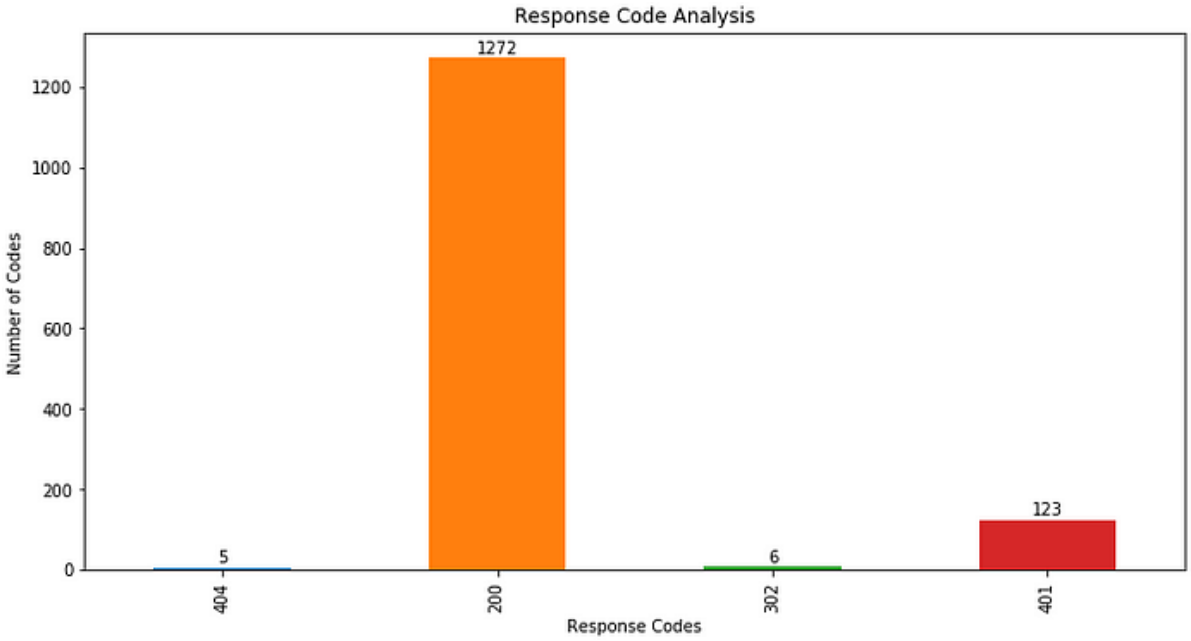
```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SQLContext
import apache_access_log # This is the first file name , in which we created Data Structure of Log
import sys

# Set up The Spark App
conf = SparkConf().setAppName("Log Analyzer")
# Create Spark Context
sc = SparkContext(conf=conf)
#Create SQL Context
sqlContext = SQLContext(sc)

#Input File Path
logFile = 'Give Your Input File Path Here'

# .cache() - Persists the RDD in memory, which will be re-used again
access_logs = (sc.textFile(logFile)
               .map(apache_access_log.parse_apache_log_line)
               .cache())

schema_access_logs = sqlContext.createDataFrame(access_logs)
#Creates a table on which SQL like queries can be fired for analysis
schema_access_logs.registerTempTable("logs")
```

Step 3 : Analyze Top 10 Endpoints which Transfer Maximum Content in MB

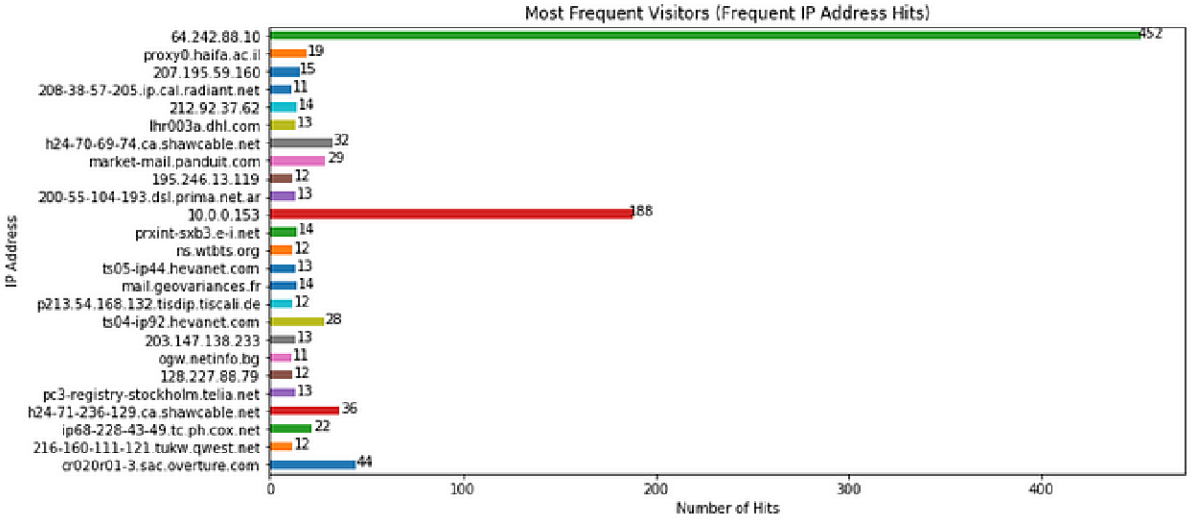
```
#Top 10 Endpoints which Transfer Maximum Content
#.rdd.map() - Will convert the resulted rows from SQL query into a map
# .collect() - actually executes the DAG to get the overall results
topEndpointsMaxSize = (sqlContext
                .sql("SELECT endpoint,content_size/1024 FROM logs ORDER BY content_size DESC LIMIT 10")
                .rdd.map(lambda row: (row[0], row[1]))
                .collect())
# Plot Analysis Code
bar_plot_list_of_tuples_horizontal(topEndpointsMaxSize,'Data Flow - MB','Enpoints','Endpoint Analysis
    based on Max Content Size')
```



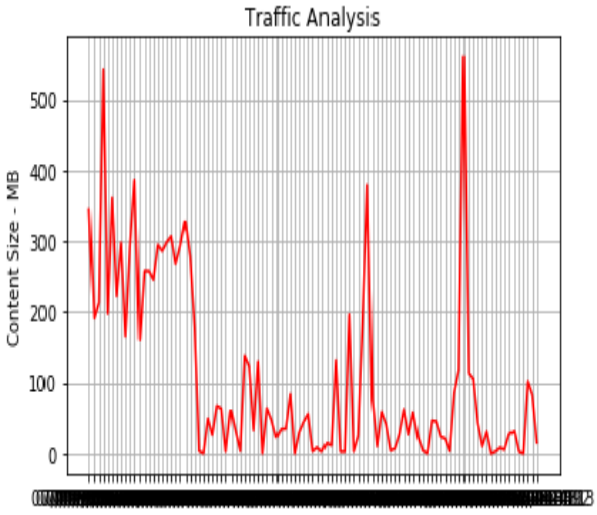
```
# Response Code Analysis
responseCodeToCount = (sqlContext
                       .sql("SELECT response_code, COUNT(*) AS theCount FROM logs GROUP BY
                           response_code")
                       .rdd.map(lambda row: (row[0], row[1]))
                       .collect())
bar_plot_list_of_tuples(responseCodeToCount,'Response Codes','Number of Codes','Response Code Analysis'
    )

# Code to Plot the results
def bar_plot_list_of_tuples(input_list,x_label,y_label,plot_title):
    x_labels = [val[0] for val in input_list]
    y_labels = [val[1] for val in input_list]
    plt.figure(figsize=(12, 6))
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(plot_title)
    ax = pd.Series(y_labels).plot(kind='bar')
    ax.set_xticklabels(x_labels)
    rects = ax.patches
    for rect, label in zip(rects, y_labels):
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width()/2, height + 5, label, ha='center', va='bottom')
```

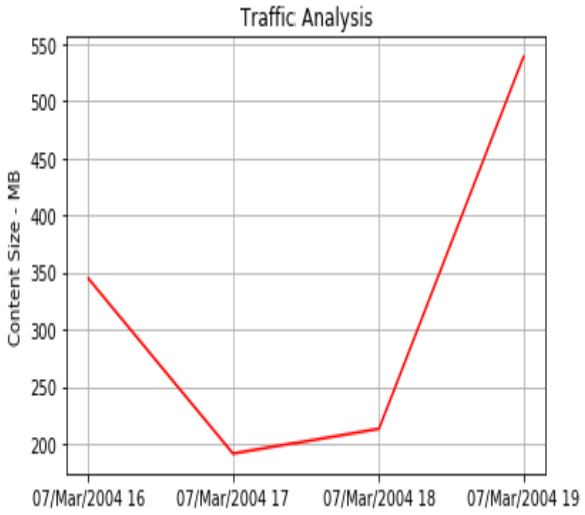



```
# Most Frequent Visitors (Most Frequent IP Address visits).
frequentIpAddressesHits = (sqlContext
                .sql("SELECT ip_address, COUNT(*) AS total FROM logs GROUP BY ip_address HAVING total >
                    10")
                .rdd.map(lambda row: (row[0], row[1]))
                .collect())
bar_plot_list_of_tuples_horizontal(frequentIpAddressesHits,'Number of Hits','IP Address','Most Frequent
    Visitors (Frequent IP Address Hits)')
```

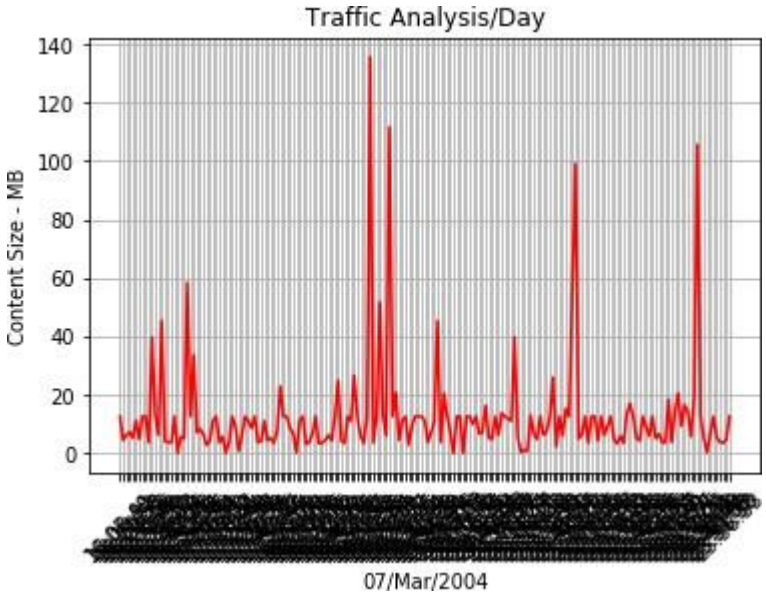



```
# Traffic Analysis for past One Week
trafficWithTime = (sqlContext
                   .sql("SELECT date, content_size/1024 FROM logs")
                   .rdd.map(lambda row: (row[0], row[1]))
                   .collect())
time_series_plot(trafficWithTime)

```

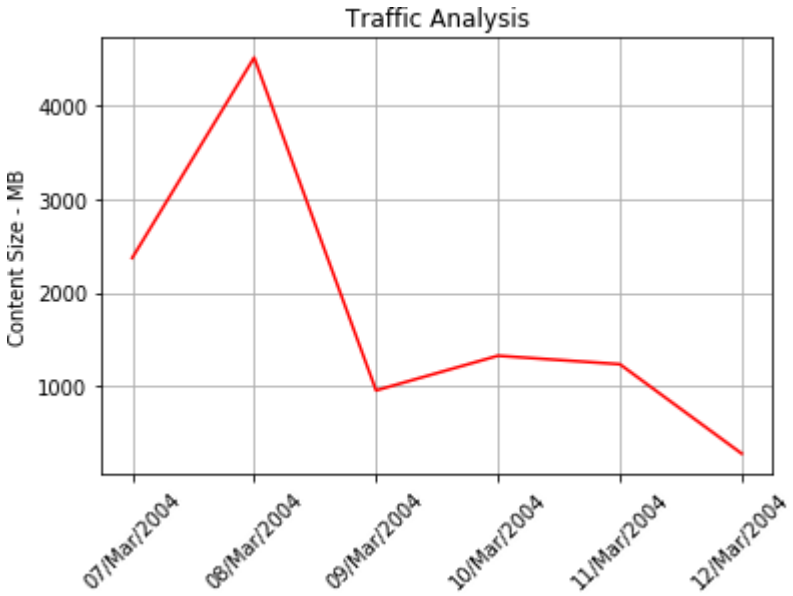





```
# Overall Traffic Analysis for a Day
Day = '07/Mar/2004'
trafficperDay = (sqlContext
                    .sql("SELECT time,content_size/1024 FROM logs where date='07/Mar/2004'")
                    .rdd.map(lambda row: (row[0], row[1]))
                    .collect())
time_series_plot(trafficperDay,Day,'Content Size - MB','Traffic Analysis/Day')
```



Outliers can be clearly detected by analysis the spikes and which end points were been hit at time by what IP Addresses.



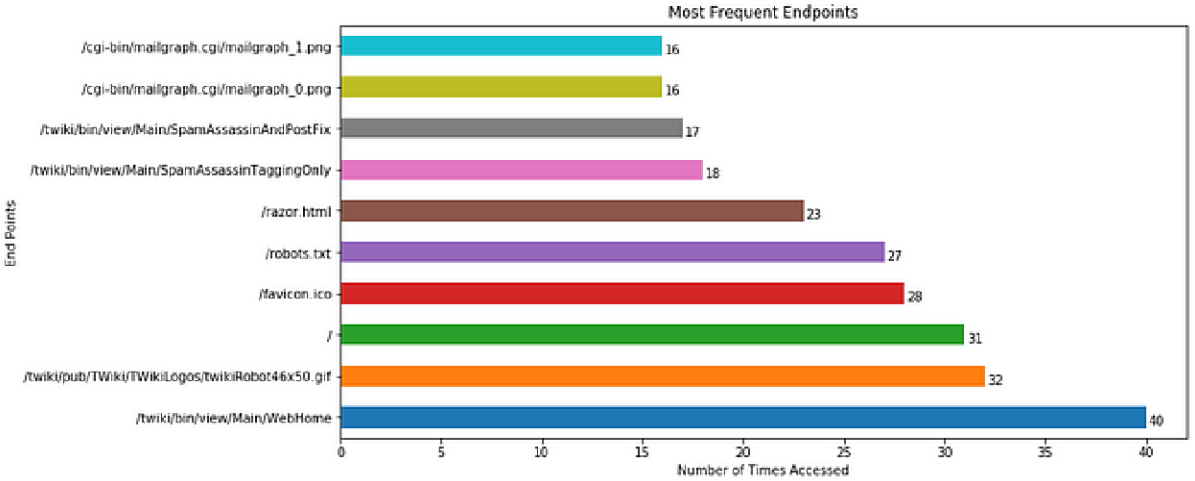
Here, we can see an unusual spike on 8th March, which can be analyzed further for identifying discrepancy.

Code for Plot Analysis:

```
def time_series_plot(input_list,x_label,y_lablel,plot_title):
    x_labels = [val[0] for val in input_list]
    y_labels = [val[1] for val in input_list]
    dict_plot = OrderedDict()
    for x,y in zip(x_labels,y_labels):
       # cur_val = x.split(":", 1)[0]
        cur_val = x.split(" ")[0]
        #print(cur_val)
        dict_plot[cur_val] = dict_plot.get(cur_val, 0) + y
    input_list = list(dict_plot.items())
    x_labels = [val[0] for val in input_list]
    y_labels = [val[1] for val in input_list]
    plt.plot_date(x=x_labels, y=y_labels, fmt="r-")
    plt.xticks(rotation=45)
    plt.title(plot_title)
    plt.xlabel(x_label)
    plt.ylabel(y_lablel)
    plt.grid(True)
    plt.show()
```

```

    def bar_plot_list_of_tuples_horizontal(input_list,x_label,y_label,plot_title):
    y_labels = [val[0] for val in input_list]
    x_labels = [val[1] for val in input_list]
    plt.figure(figsize=(12, 6))
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.title(plot_title)
    ax = pd.Series(x_labels).plot(kind='barh')
    ax.set_yticklabels(y_labels)
    for i, v in enumerate(x_labels):
        ax.text(int(v) + 0.5, i - 0.25, str(v),ha='center', va='bottom')
```

```

   # Frequent End Points
topEndpoints = (sqlContext
                .sql("SELECT endpoint, COUNT(*) AS total FROM logs GROUP BY endpoint ORDER BY total
                    DESC LIMIT 10")
                .rdd.map(lambda row: (row[0], row[1]))
                .collect())
bar_plot_list_of_tuples_horizontal(topEndpoints,'Number of Times Accessed','End Points','Most
    Frequent Endpoints')
```



CONCLUSION: Hence, with the help of the above experiment, we successfully built a log analytics application as a case study for Spark streaming. We visualized most frequent visitors, traffic analysis for a day and past one week through our implemented application.