ANALYZING ACCESS LOGS DATA USING STREAM BASED ARCHITECTURE

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Nitendra Gautam

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

February 2018

Fargo, North Dakota

North Dakota State University
Graduate School

Title

ANALYZING ACCESS LOGS DATA USING STREAM BASED ARCHITECTURE

By

Nitendra Gautam

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Anne Denton
Chair

Dr. Saeed Saleem

Dr. Kambiz Farahmand

Approved:

02/16/2018
Date

Dr. Kendall Nygard
Department Chair

ABSTRACT

Within the past decades, the enterprise-level IT infrastructure in many businesses have grown from a few to thousands of servers, increasing the digital footprints they produce. These digital footprints include access logs that contain information about different events such as activity related to usage patterns, networks and any hostile activity affecting the network. Apache Hadoop has been one of the most standardized frameworks and is used by many Information Technology (IT) companies for analyzing these log files in distributed batch mode using MapReduce programming model. As these access logs include important information related to security and usage patterns, companies are now looking for an architecture that allows analyzing these logs in real time. To overcome the limitations of the MapReduce based architecture of Hadoop, this paper proposes a new and more efficient data processing architecture using Apache Spark, Kafka and other technologies that can handle both real-time and batch-based data.

ACKNOWLEDGEMENTS

I would like to thank and express my sincere gratitude to my advisor Dr. Denton for introducing me to the field of Big data processing in real time and her guidance during this research. I would like to thank her for addressing all my underdeveloped questions and explaining my doubts with enthusiasm.

I am grateful to my committee members Dr. Saeed Salem and Dr. Kambiz Farahmand for their invaluable time and support. I would also like to thank my friends at North Dakota State University for making my stay memorable at NDSU.

DEDICATION

This work is dedicated to my parents whose unconditional love and hard work have taught me to success in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API .....Application Programming Interface

HDFS .....Hadoop Distributed File System

RDD .....Resilient Distributed Datasets

DStream .....Discretized Stream

IT.....Information Technology

RDBMS.....Relational Database Management System

IP .....Internet Protocol

HTTP.....Hyper Text Transfer Protocol

CLF .....Combined Log Format

SFTP .....Secure File Transfer Protocol

DAG.....Direct Acyclic Graph

URL.....Uniform Resource Locator

XML.....Extensible Markup Language

IoT.....Internet of Things

1. INTRODUCTION

Within last decade, there has been a dramatic increase in the number of people having internet access and number of customers accessing internet based applications. Most of these applications are hosted on multiple web servers that store the activity of customers in their servers as digital footprints. As the number of customers have increased from millions to billions, these digital footprints have also grown from millions to billions of transactions per day. The footprints consist of web usage data that are recorded in the form of access logs on the web servers. These access logs contain a variety of important pieces of information such as web server details, source or Internet Protocol (IP) address of the request from the client, user activities detail during that session, time spent on these activities and other relevant information. These logs help to find out the total number of people who have visited the website along with the total time spent on the website. It can be also used to detect security threats or abnormalities in the logs that can be used to protect customers from those threats. This, in turn, increases trust among customers that can be used as an advantage against competitors.

1.1. Problem Statement

Increase in the number of customers that use internet-based application have led to a surge in the amount and size of access logs being produced on web servers. Most of the companies which develop these applications, spend a large part of their yearly budget on the periodic analysis of the access logs as the size of these logs have increased from Terabytes to Petabytes scale falling in the category of big data. Traditional data analytics stacks such as Hadoop provide capabilities for handling the event logs in a batch-oriented fashion in offline mode using the MapReduce algorithm.

Although Hadoop can efficiently handle the batch or offline processing of event logs after the events have occurred, it cannot reflect the unbounded and continuous/streaming nature of log data as they are produced in many real-world applications. One of the main problems with this approach is that once an event occurs, some contexts of the event logs and information collected from them becomes irrelevant and useless.

Modern streaming analytics applications such as Spark support the “data in motion” paradigm by processing real-time data in addition to the capability of handling static data when it becomes available. An implementation of streaming based data analytics approach helps to decrease the total time spent per data unit in the processing pipeline as data are continuously ingested into the streaming platform and analyzed in real time.

There are many open source and commercial applications like Apache Storm or International Business Machine (IBM) streams that can handle either stream or batch data efficiently. However, many of them do not provide a scalable and economically viable unified platform to handle both data types. This paper aims to evaluate two of the open source platform namely Hadoop and Spark with respect to their capability for handling both batch and stream log data. This paper proposes a more efficient and scalable data analytics architecture based on open source tools like Spark that can handle both data in real time and batch data [37].

2. LITERATURE REVIEW

Most of the modern applications in today's market are deployed on multiple production web servers in different data centers or in the cloud-based web servers that are distributed geographically. These web servers which, host various applications, record different user and application activity and save it in the form of messages in a file called log file [1].

Log data can be recorded in different formats as their sources on the production servers are hosted on different systems. Some of these sources can include server access log, proxy server log, browser log or any other data source generated by users coming from a web server. Even though the sources of logs can be in different formats, some of the basic information contained in the log file is common to most of them. According to Verma and Bhatia [2], some of this common information consists of various attributes like IP address, access date and time, request method (GET/POST), Uniform Resource Location (URL) of the accessed page, transfer protocol, success return code etc. Different data preprocessing activities like data cleaning, data reduction, and related algorithms must be performed to extract the data from the log file [3].

The rate at which log files are produced in modern distributed applications ranges from several terabytes to petabytes per day. Among the different types of logs recorded on the application servers, access logs contain the information related to user navigational behavior and user access patterns. Mavridis and Karatza [4] have used the MapReduce algorithm to identify the user's browsing patterns by analyzing the user's navigational behavior in the access logs. In addition to identifying the navigational behavior, it is also used in identification of system threats and problem identification based upon these access log.

The Hadoop Distributed File System (HDFS) is the backbone for the Hadoop framework that is designed for high throughput data input/output rate rather than a high-performance

input/output rate. Data needs to be copied first to HDFS before starting any computation or running a Hadoop job. This makes it difficult to use the Hadoop framework for analyzing data streams like log streams and clickstreams or any continuous data. Most MapReduce jobs have some amount of waiting time once the computation is started. This waiting time ranges from a couple of minutes to a couple of hours until the whole computation is completed. Most of today's applications produce data continuously and require it to be evaluated when it arrives in real time. As not all data is available up front, the computation must happen on the fly when new data arrives. An important example would be to detect anomalies continuously from incoming events in real time [5][6].

Shahrivari [7] describes the advantages of Spark to Hadoop in terms of latency period and hard disk speed. Spark provides an in-memory computation framework that allows use of distributed main memory for processing the data in real time. Use of in-memory computation in Spark allows computing more than 10 GB/second in comparison to the 200 MB/seconds hard disk speed in the Hadoop framework. Spark provides nanosecond latency periods when computed in-memory, whereas it provides millisecond latency periods when using local hard disks for computation. The price of memory has decreased so much that in-memory computing is feasible in comparison to disk-based computation. In-memory computation also supports caching of data, which allows for faster retrieval of frequently used data from memory [7].

Wingerath, Wolfram, et al [8] have noted that Spark is moving away from the batch-based paradigm to meet the real-world requirements of working with data that arrives continuously and must be analyzed in streams. Spark Streaming splits the incoming stream of data into small micro batches, which are further converted into Resilient Distributed Datasets(RDD). A DStream contains the number of RDDs that are processed in order whereas

data inside an RDD are processed parallelly without order guaranteed. Using this paradigm Spark provides a faster streaming-based platform challenging batch-based data processing platform using Apache Hadoop. These RDDs are then processed using the actions and transformation as provided by Spark API [8].

3. CONCEPTS

This chapter gives a detail introduction to the concepts needed to understand this paper.

3.1. Log Files

Log file consist of different messages stored in the form of log message or log. A log message can be defined as a message that a computer, system, server, device or any software generates upon any activity. Most of these log file include information such as the timestamp at which the log message was generated, information about the system source which produces the logs and other data relevant to the activity. There is not a single de-facto standard for the log format as different system can implement the format in different ways. Logs have been mainly categorized into four different types [1][4].

- **Access Log file (Transfer):** This log file stores data about all incoming requests, information about the client and records all request that is processed by the web server.
- **Error Log file:** This file contains a list of all internal errors that occur when there is a connection problem between client and server. Whenever a client requests a web page from a server and an error occurs, it is recorded in an error log file.
- **Agent Log File:** This log file contains data and meta-data related to client browser and browser version.
- **Referrer Log File:** This file contains information about links and redirects visitor information to the site.

3.1.1. Access Logs

Access Logs are the server logs that record all HTTPS (Hyper Text Transfer Protocol Secure) request processed by a web server. It maintains the history of page request made to the

server along with the other useful information like type of request, time of request and a status code of the request. World Wide Web Consortium or W3C maintains a standard common log format for web server access Log files. This paper focuses on the Combined Log Format (CLF), which is the most popular access logs format among the four. The entries of a Log give details about the client that made the request to the server [22] [20] [23]. Below is the configuration for the combined log formats.

LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-agent}i\""

The file entries, which is produced in CLF, is in below format.

97.63.130.73 - - [03/May/2017:18:16:33 -0500] "DELETE /apps/cart.jsp?appID=7978 HTTP/1.0" 200 4902 "http://pitts-jackson.com/home/" "Mozilla/5.0 (Windows NT 5.1) AppleWebKit/5320 (KHTML, like Gecko) Chrome/13.0.817.0 Safari/5320".

96.63.130.73(%h): This is the IP address of the client which made the request to the server. Generally, IP address assigned by the Internet Service Provider would be the identification of the user or client name. If we can track the client IP address, we can identify the user who visited website.

- (%l) : Hyphen present in the log file entry means requested is not available.

[03/May/2017:18:16:33 -0500](%t) : It is the time at which the request was received. Format of the time resembles like [day/month/year:hour:minute:second zone].

"DELETE /apps/cart.jsp?appID=7978 HTTP/1.0"(\"%r\") : It represents the request which is coming from the client as given by double quotes. DELETE is the type of REST(Representational State Transfer) method used. Instead of DELETE we can have GET, POST, PUT and PATCH. *'/apps/cart.jsp?appID=7978'* represents the information requested by the client. Here *'HTTP/1.0'* represents the protocol used.

200(%>s): It is the status code which web server sends back when the request is received. Status codes beginning with 2 is for a successful response, 3 is for redirection, 4 is for error caused by the client and 5 for error related to a server.

4902(%b): It refers to the size of the object (in bytes) returned to the client by the web server but does not include the response headers. If the server returns an empty content to the client, it will logged as a "-".

"http://pitts-jackson.com/home/"(\"%{Referer}i\"): It gives the information about the website or the URL from which referred to the current page.

***"Mozilla/5.0 (Windows NT 5.1) AppleWebKit/5320 (KHTML, like Gecko) Chrome/13.0.817.0 Safari/5320"*:** It has the information about the browser from where the user sends the request to the web server. It is a string which describes the type and version of the browser which was being used.

3.1.2. Clickstream Analysis

Clickstream analysis is the process of collecting and studying the user's behaviors from the data that gets generated in the access log by clicking on a web page. Raw form of this data gets saved in the form of access logs that gives different user behavior information and helps the company in making a strategic decision regarding their business [20] [21].

- Duration of their visits
- Page views frequency
- Types of item viewed on the web page.
- Geographical location of the user
- Browser type of User

3.2. Big Data

Big data is a problem of dealing with structured, semi-structured and unstructured data sets so large that it cannot be processed by using conventional relational database management systems. It includes different challenges such as storage, search, analysis, and visualization of the data, finding business trends, determining the quality of scientific research, combatting crime and other use cases that would be difficult to derive from smaller datasets [10].

Structured data have a predefined schema and represents data in row and column file format. Though many examples of structured data exist, some of the important examples can be taken as Extensible Markup Language (XML) Data warehousing data, databases data, Enterprise Resource Planning (ERP) data, customer relationship management (CRM) data.

Semi-structured is a type of self-describing structured data which does not conform with the data types as in relational data but contains some tags or related information which separate it from unstructured data. Some examples of semi-structured data are XML and JSON data format.

Unstructured data are data types which do not have a predefined schema or data model. With the ambiguity in a formal pre-defined schema, traditional applications have hard time reading and analyzing the unstructured data. Some examples of unstructured data are video, audio, and, binary files. Big data can be categorized based on four properties which are volume, variety, velocity and veracity [9].

3.3. Four V's of Big Data

Volume: Data have grown at exponential growth in the last decade as the web evolution has brought more devices and users in internet grid. The storage capacity of the disk has increased from megabytes to terabytes and petabytes scale as enterprise level applications started producing data in large volumes.

Variety: Explosion of data has caused a revolution in the data formats types. Most of the data formats such as Excel, database, and Comma Separated Values (CSV), Tab Separated Value (TSV) files can be stored in a simple text file. There is no any predefined data structure for big data because of which it can be in either structured, unstructured or a semi-structured format. Unlike the previous storage medium like spreadsheets and databases, data currently comes in a variety of formats like emails, photos, Portable Document Format (PDF), audios, videos, and monitoring devices etc. Real world problems include data in a variety of formats that possess a big challenge for technology companies.

Velocity: With the explosion of social media platform over the internet, it caused explosion in the growth of data in comparison to data coming from traditional sources. There has been massive and continuous flow of big data from the sources like social media websites, mobile devices, businesses, machines data, sensors data, web servers and human interaction within the last decade. Modern people are hooked into their mobile devices all the time updating their latest happening in their social media profiles leaving a huge electronic footprint. These electronic footprints are collected every second at high speed at petabytes scale.

Veracity: It is not always guaranteed that all the data that gets produced and ingested into the big data platform contains clean data. Veracity deals with the biases, noise, and abnormality that might arrive with data. It reflects one of the biggest challenges among the analysts and engineers to clean the data. As the velocity and speed of data keep on increasing, big data team must prevent the accumulation of dirty data in the systems [11].

3.4. Apache Hadoop

Hadoop is an open source framework that is used for processing large data sets across clusters of low-cost servers using simple MapReduce programming models. It is designed to

scale up from one server to multiple servers, each of them offering computation and storage at the local level. Hadoop library is designed in such a way that high availability is obtained without solely relying on the hardware. Failures are detected at application layer using Hadoop and handled well.

Google was the first organization which dealt with the massive scale of data when they decided to index the internet data to support their search queries. In order to solve this problem, Google built a framework for large-scale data processing using the map and reduce model of the functional programming paradigm. Based on the technological advancement that Google made related to this problem, they released two academic papers in 2003 and 2004. Based on the readings of these papers Doug Cutting started implementing Google MapReduce platform as an Apache Project. Yahoo hired him in 2006 where he supported the Hadoop Project.

Hadoop mainly consists of two components, a distributed filesystem known as HDFS and the MapReduce programming model to process that data [12].

3.4.1. Hadoop Distributed File System (HDFS)

HDFS is a distributed file system that is designed for storing very large files with streaming data access patterns running on clusters of commodity hardware. It was originally created and implemented by Google, where it was known as the Google File System (GFS). HDFS is designed such that it can handle large amounts of data and reduces the overall input/output operations on the network. It also increases the scalability and availability of the cluster because of data replication and fault tolerance.

When input files are ingested into the Hadoop framework, they are divided into a block size of 64 MB or 128 MB and are distributed among Hadoop clusters. Block size can be pre-defined in the cluster configuration file or can be passed as a custom parameter while submitting

a MapReduce job. This storage strategy helps Hadoop framework store large files having bigger size than the disk capacity of each node. It enables HDFS to store data from terabytes to petabytes scale [12].

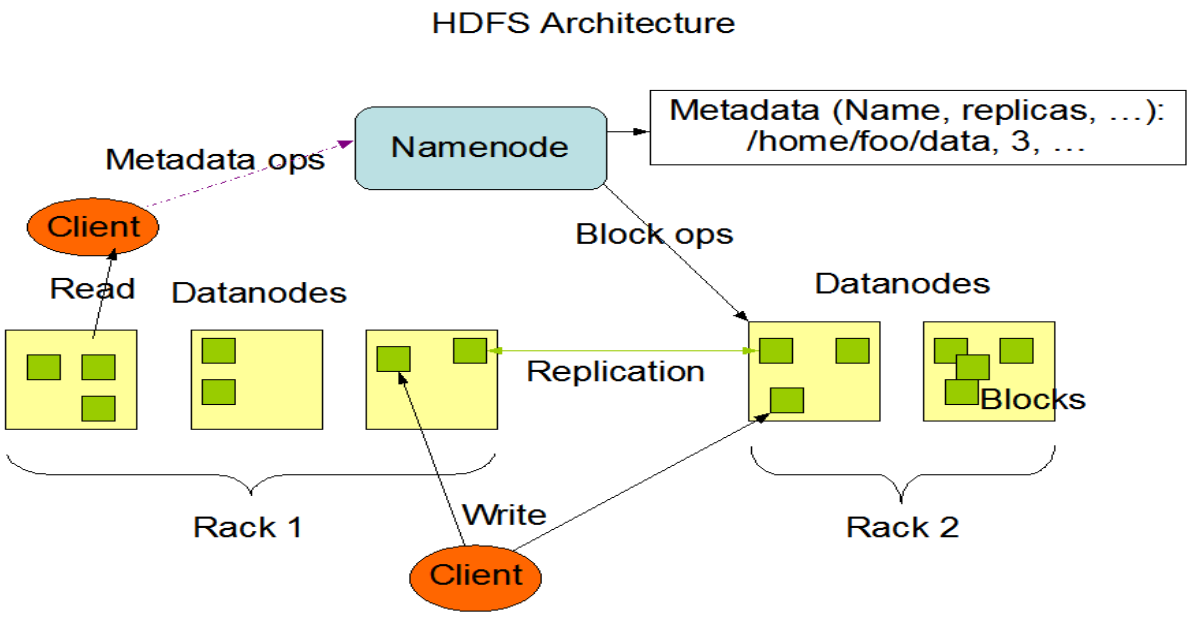

Figure 1. HDFS Architecture Diagram.

HDFS uses the concepts of a Namenode and Datanodes that are based on a master/slave architecture.

Namenode: It is the node that stores the file system metadata by maintaining it in an in-memory table. These tables include the information such as which file maps to what data block and what data blocks are stored on which data node. Whenever a data node reports a disk corruption or other problem, the first table gets updated and whenever the data node dies because of a node or network failure, both the table get updated.

Datanode: It is the node where the actual data resides and is responsible for reading and writing requests from clients. When any file is stored in data node it is converted to different storage blocks and replicated in different nodes.

3.4.2. MapReduce

MapReduce is the parallel programming model that is used for processing large chunks of data. A Map-Reduce job splits the input datasets from the disk into independent chunks if these data cannot be stored on one single node. MapReduce job first executes the mapping tasks to process the split input data in a parallel manner and sorts the output of the map function and sends the result to reduce tasks as their input. MapReduce framework is responsible for scheduling tasks, monitoring them and re-executing the failed tasks.

The MapReduce framework converts the given input datasets into <key, value> pairs and works on all input datasets in this format. To use this framework input data is first converted as a set of <key, value> pairs that in return produces a set of <key, value> pairs as output. Input datasets for this framework can be both the structured, semi-structured or unstructured data sets stored in an HDFS.

The MapReduce framework can be divided into following phases:

Split Phase (Input Splits): In this phase, given input data is divided into input splits based on the given input format. Splitting of Input data is equivalent to the map task that runs in a parallel fashion. In this phase, data stored on HDFS is split and broken up line-by-line and then sent to the mappers.

Mapper/Map Phase: In this phase, input data is split into key/value pairs based on the logic used by user in their mapper code. All the data is sent to the mapper based on the key values present in input data.

Shuffle and Sort Phase: This phase executes in the Datanode and consists of two steps shuffle and sort. This phase uses all the network bandwidth and the computing capacity of the

Datanode. In shuffle phase, output data from mapper phase is partitioned and grouped according to the given key. Sort phase consists of sorting the data and sending the results to the reducers.

Reduce Phase (Reducers): This phase aggregates the key/value pairs based on logic defined by the users. A reducer function receives an iterate input values from an output list and combines these values together to give a single value output [11] [13].

All these phases can be represented as follows.

(Input Data) -->map(K1,V1) --> list(K2,V2) -->Shuffle /Sort-->reduce (k2,list(v2))-->list(k3,v3)-->(Output Data)

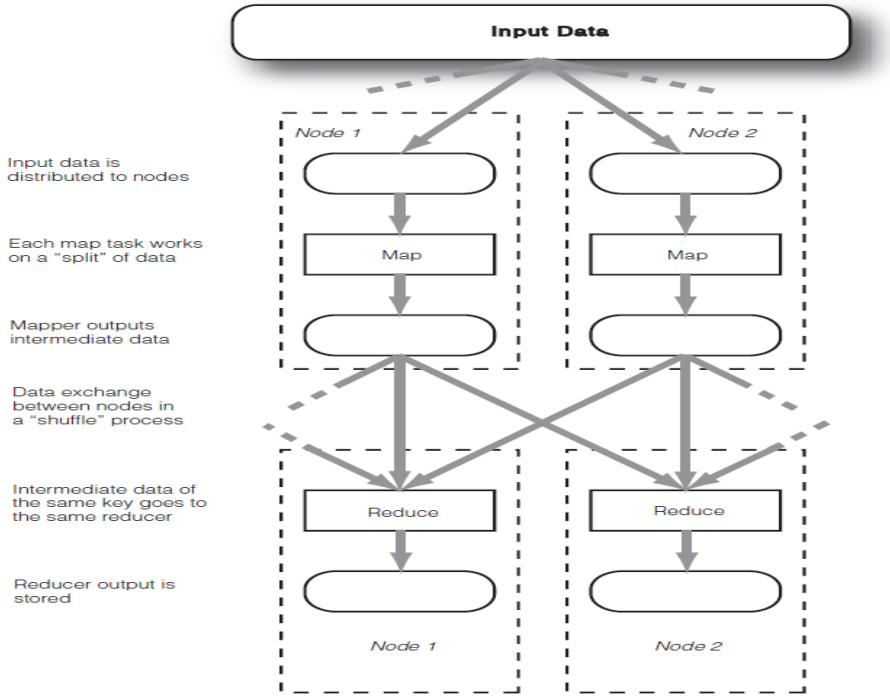

Figure 2. MapReduce Framework.

3.5. Apache Spark

Spark is an in-memory fast data analytic engine for large-scale data processing. It was developed in 2009 by AMPLab of University of California Berkeley as part of PHD thesis by Matei Zaharia. AMPLab donated this project to Apache Software Foundation in 2013 and it became the top-level project in Apache Software Foundation in the consecutive year. In contrast

to Hadoop, Spark extends the MapReduce programming model and supports both disk-based and in-memory computing of the data. It provides different Application Programming Interfaces (API) in Java, Scala, Python and R to develop applications for distributed computing. Spark mainly provides four main submodules which are Structured Query Language (SQL), MLlib, Graph and Spark streaming. Since Spark processes all the data in-memory it is much faster in compared to disk-based MapReduce model [14].

The Spark computational engine is different from that of the Hadoop-based MapReduce model. It does not have its own native distributed file system like HDFS but can read data from HDFS, local file system and from in-memory datasets stored as RDDs [15] [16].

3.5.1. Spark Components

Spark is a general-purpose data processing engine that supports a variety of processing components. There is a core component that acts as a base library to all components. Mainly it can be divided into following components [14] [18].

- Spark Core: It is a general executing engine for Spark platform which acts as a base for other components.
- Spark SQL: It is a Spark module for processing data which has structured format in form of row and columns. This module acts as a distributed SQL query engine.
- Spark Streaming: This component allows to build an interactive and analytical applications based on real-time streaming as well as historical data.
- Machine Learning Library (MLib): It provides different spark libraries used for making machine learning related applications.
- GraphX: It is a Spark API for performing parallel computation based on graph data.

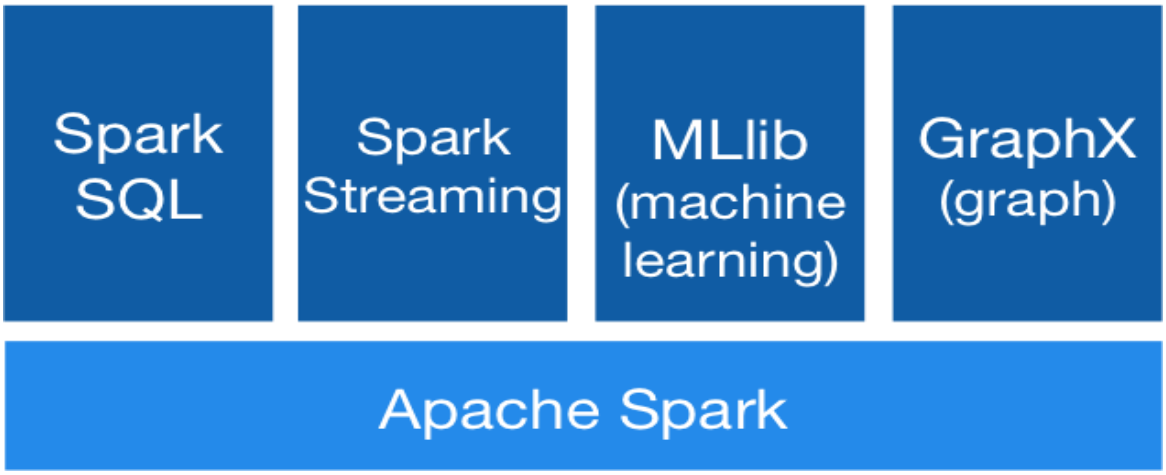

Figure 3. Spark Components.

3.5.2. Resilient Distributed Datasets (RDD)

Resilient Distributed Datasets is the parallel, abstract, fault-tolerant and immutable data structure used for data storage within the Spark architecture. When datasets are converted into RDD, Spark distributes the RDD data in parallel fashion throughout the cluster by dividing it into different logical partitions.

There are two types of operation namely transformation and action which Spark RDD supports. In case of transformation operation, new RDD data sets is generated from an existing one whereas in action operation a computation is applied to existing RDD and value is returned. Spark evaluates the transformations lazily so that high efficiency is achieved. It means that computation of new RDD only happens when action operation is executed. RDD can also be persisted or cached in main memory to allow faster computation on the data sets. Since the RDD can be cached, it makes the spark cluster fault tolerant as it can be recalculated if the partition of RDD is damaged [15] [17].

3.5.3. Spark Streaming

Spark Streaming module is an extension of the core Spark API for providing a scalable, high throughput, fault-tolerant stream processing of real-time streams data. Data into spark

streaming can be ingested through various data sources such as Apache Kafka, Flume, file source, Kinesis or Transmission Control Protocol (TCP) sockets-based stream processing. Once the data is ingested into spark cluster, it can be analyzed or processed using various functions such as map, reduce or other aggregate functions as provided by Spark API. Processed data can be either stored in the file system, databases or can be connected to live dashboard for dashboards in real time [15].

Figure 4. Spark Streaming Input/output Data Sources.

Figure 5. Spark Streaming Flow.

3.6. Apache Kafka

Apache Kafka is an open source real-time publish-subscribe messaging system that is fast, scalable and fault-tolerant. It is also a distributed, partitioned and replicated commit log

service. It was first developed at LinkedIn where they faced a similar problem where they had to manage logs coming from multiple data sources and destinations. With the help of Kafka, all the data pipelines were streamlined into a centralized server which prevented the need for developing one pipeline for each source as shown in Figure 6 figure. It also reduces the complexity of the pipelines and lowers the operational cost. Kafka needs an instance of Zookeeper cluster running which manages all the configurations related to Kafka.

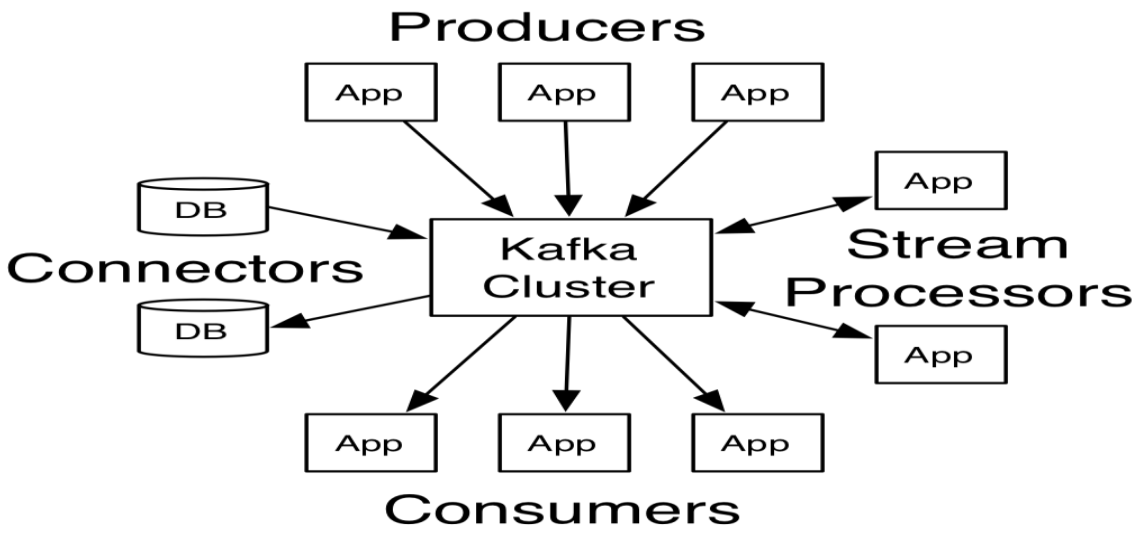

Figure 6. Kafka Architecture.

In traditional message processing, computations are applied on the messages mainly individually per cases. In stream processing platform, complex operation like aggregations and joins are applied on multiple records at once. Traditional messaging system cannot go back in time and execute complex operation on old messages as it is deleted once message is delivered to all subscribed consumers. In contrast to this, Kafka keeps the message for configurable amount of time as it uses a pull-based model for consuming messages. Because of this feature, consumer can consume data irrespective of the producer's life cycle or can re-process or re-consume the

messages multiple times if it failed for the first time because of some reason. This helps to write to loosely coupled micro-services with a stateless and scalable architecture [19].

3.6.1. Kafka Components

This section describes the most important components in Kafka [19].

- **Topic:** A topic is a category or feed name to which records or messages are published. In Kafka, topics can have either zero, one or many consumers which subscribe to the data written to it.
- **Partition:** Partition is an ordered and immutable sequence of records that is continually appended to a structured commit log. Each record in the partitions is assigned a sequential id number called as offset which uniquely identifies each record within the partition.
- **Producer:** Producer publishes the data to the desired topic of their choice and is responsible for choosing which record to assign which partition within the topic.
- **Consumer:** Consumer read the messages that are published in a topic by producer. Consumer can run as a separate process or on separate machines. Each consumer label themselves with a consumer group name such that each record published to a topic is delivered to one consumer instance within each subscribing group.

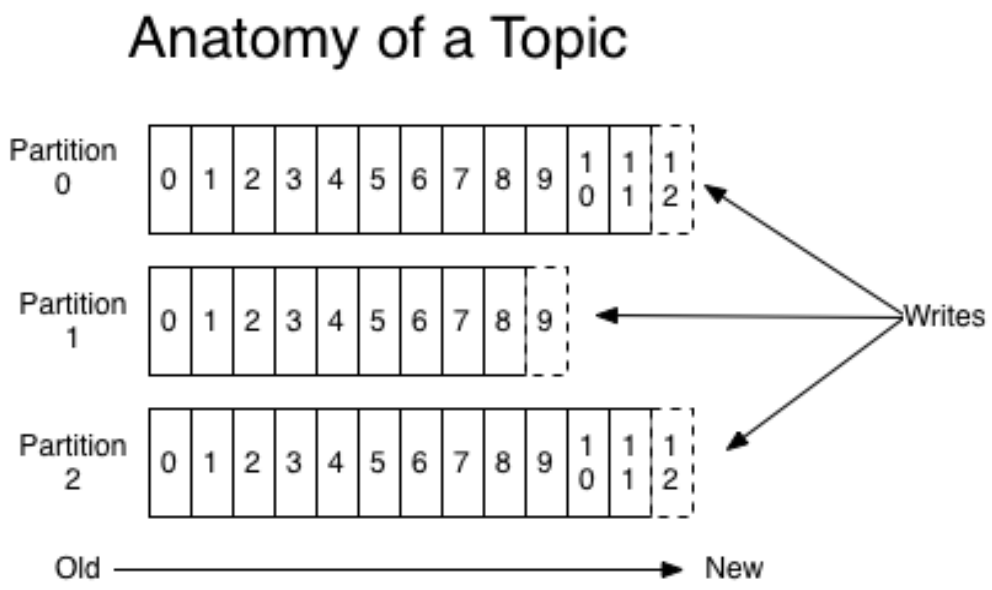

Figure 7. Kafka Log Anatomy.

3.7. Grafana Dashboard

Grafana is open source dashboard used for visualizing time series metrics and analytics data. It is commonly used for visualizing real and near-real-time data for data centers, production servers, industrial sensors, home automation, weather data and other Internet of Things(IOT) based applications. As it supports refreshing of a dashboard in periodic interval, it is perfect for visualizing data in real time. Grafana provides different data sources for connecting the dashboard. As Grafana natively supports time series data for refreshing the dashboard in periodic dashboard interval, it would be a good choice for displaying access log data [24].

3.8. InfluxDB

InfluxDB is a time series database that is used to handle high write and query loads. It is mainly used for large amount of timestamp data for monitoring server activities, application metrics, IOT sensor data and real-time analytics data. It supports expressive SQL like query language to get aggregated data easily [25].

4. DESIGN AND IMPLEMENTATION

I have conducted clickstream analysis to evaluate the performance of the streams and batch data processing architecture. This experiment is aimed to show the data analytics capability based on the platforms provided by Hadoop and Spark for batch and real-time data processing while performing clickstream analysis. These experiments do not intend to carry out a full benchmark comparison between these technologies, as this is beyond scope of the paper and computer hardware available.

In clickstream analysis, the access log is analyzed to extract the aggregate data such as unique IP address and number of times it visited the server. This experiment compares the time taken for Hadoop and Spark batch job while aggregating the log data from 1 million to 11 million records. Each of the log files is copied into local and Hadoop cluster before performing this analysis.

In an extension to the batch data processing architecture, real-time streaming architecture based on spark streaming is proposed which analyzes the access logs in real time. In stream data processing architecture using Spark, we calculate and aggregate the client IP address, HTTP Status Code, HTTP Request Bytes, HTTP Request field and timestamp from access logs. Then these aggregated logs are persisted into InfluxDB database and visualized using Grafana dashboard. Table 1 explains the different tools and their version used in this design.

Table 1. Software Version.

Software	**Java**	**Scala**	**Apache Kafka**	**Apache Spark**	**Apache Hadoop**	**Apache Zookeeper**	**Grafana**	**InfluxDB**
Version	1.8	2.11.8	0.9	2.01	2.8	3.4.9	4.3.2	2.7

This design is implemented in a virtual environment set up on HP omen laptop which consists of 2 core 12 GB RAM and 50 GB Disk capacity with an Ubuntu OS. In this

environment components such as Spark, Hadoop and Kafka are installed in a pseudo-distributed mode in a single node cluster. This environment used to compare the performance of Hadoop MapReduce, Spark Batch and Spark Streaming application.

4.1. Batch Based Architecture Using Spark

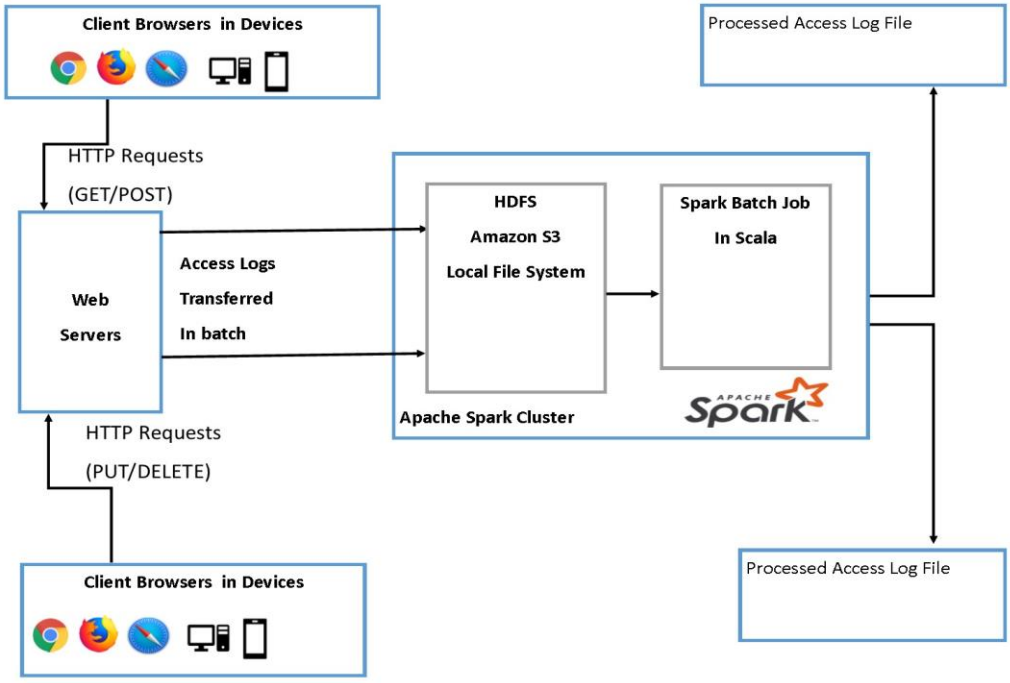

Figure 8. Batch Processing Architecture Using Spark.

In this architecture, access logs that are collected and stored in web servers are analyzed offline based on batch processing using Spark batch job. When different web browsers like Chrome, Safari and Firefox through desktop and handheld personal devices hit the web servers, access logs are collected in the servers. These access logs are collected and transferred to Spark storage cluster through SFTP (Secured File Transfer Protocol) where these logs are processed using Spark batch application. A Spark job can read the access logs through different storage types like disk, HDFS or an Amazon S3 storage. Below command is used to execute Spark job in

Spark cluster for 1 Million log record which can be modified for processing other log record files.

$SPARK_HOME/bin/spark-submit --class com.nitendragautam.sparkbatchapp.main.Boot
--master spark://192.168.133.128:7077 --deploy-mode cluster --supervise
--executor-memory 4G --driver-memory 2G --total-executor-cores 1
/home/hduser/sparkbatchapp.jar
/home/hduser/NDSBatchApp/input/KafkaStreaming_1Mil.log
/home/hduser/NDSBatchApp/output/KafkaStreaming_1Mil

Once data is read from the source, Spark caches it into memory and stores as RDD named *accessLogsRDD.* This RDD is processed in-memory to count the no of times client accessed the web server. After the Spark job is completed IP address and number of times Client IP address accessed the web servers is collected [30].

4.2. Batch Based Architecture Using Hadoop MapReduce Model

In this architecture, Hadoop based MapReduce model is used to process the access logs offline after the logs are collected. When different web client like chrome, Safari, and Firefox hit the web servers, access logs are collected in the servers. Once the access log is collected, it is sent to the Hadoop cluster through SFTP and then copied to HDFS using Hadoop file system command. Once the data is copied into HDFS, it is processed in batch mode using MapReduce job. Below command is used to execute MapReduce job in Hadoop cluster for 1 Million record which can be modified for processing other log record files.

hadoop jar /home/hduser/loganalyticshadoop-1-jar-with-dependencies.jar
com.nitendragautam.mapreduce.LogAnalytics /user/hduser/input/KafkaStreaming_1Mil.log
/user/hduser/output1/KafkaStreaming_1Mil

In the MapReduce application as shown in Appendix E and [31], each line of access logs is converted into key/value pairs of (Client IP Address, 1) in the *LogAnalyticsMapper* class. Once the mapper class is completed output from the mapper class acts as an input for reducer class.

In *LogAnalyticsReducer* class, input key/value from the mapper is iterated and count of (Client IP Address, total number occurrence) is calculated as key/value pairs. As a result, we get Client IP address and number of times it accessed the web server [31].

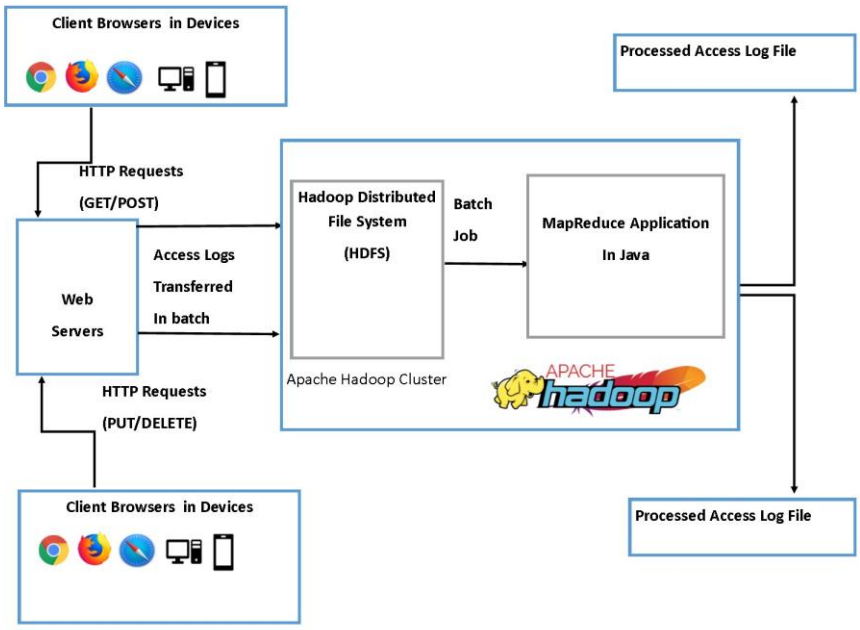

Figure 9. Batch Processing Architecture Using Hadoop.

4.3. Streaming Based Architecture Using Spark

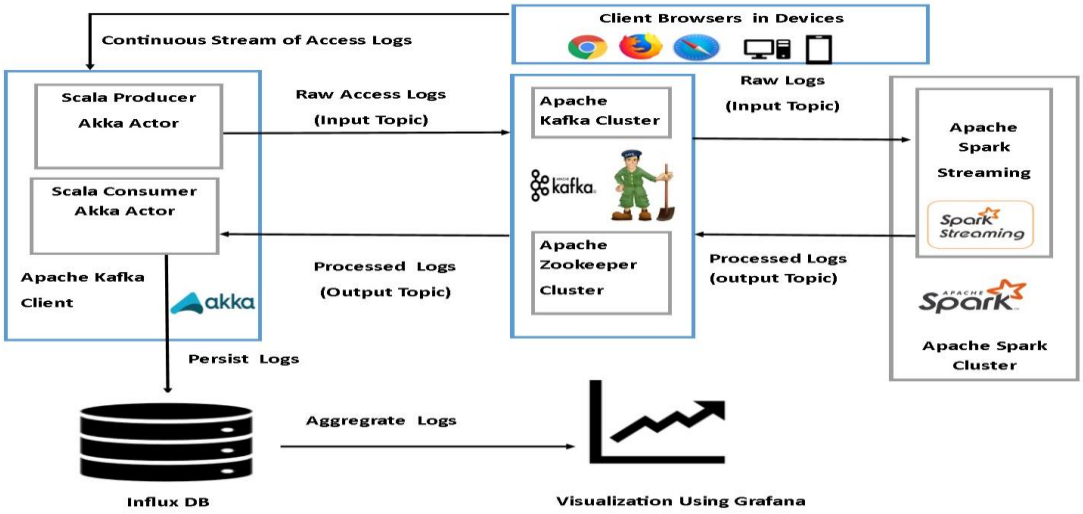

Figure 10. Proposed Streaming Based Architecture Diagram Using Spark Streaming.

In the proposed streaming based architecture, a continuous stream of access Logs is first generated by using open source python script- which produces logs continuously [26]. Given script is refactored to accommodate the static IP address and more HTTP Status codes along with a static log file name.

Following command produces a file named KafkaStreaming.log with 1 million records using this script.

python apache-fake-log-gen.py -n 10000000 -o LOG -p D:\App\KafkaStreamingProducerJob\Data\KafkaStreaming

A Kafka producer Client, natively written for this architecture, tails the logs file name *KafkaStreaming.log* and sends the raw access logs to a Kafka topic named *ndsloganalytics_raw_events*. The main responsibility of this producer client is to aggregate streaming raw logs from the local file system and to send the raw events to Kafka cluster. This topic in Kafka which represents the raw log events is created by the following command [27].

sudo $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper 192.168.133.128:2181 --replication-factor 1 --partitions 1 --topic ndsloganalytics_raw_events --config retention.ms=7200000

A Spark streaming application which is running in Spark cluster ,reads the raw logs from Kafka topic named *ndsloganalytics_raw_events* in real time. It creates a direct stream from this topic that continuously reads the data. For each record that is passed as stream every three seconds RDD is created as shown in code below.

```
val conf = new SparkConf().setAppName("SparkStreamingApp")
     //Spark Streaming Context
 val ssc = new StreamingContext(conf,Seconds(3))
         val kafkaTopic="ndsloganalytics_raw_events"
         val consumerTopic = List(kafkaTopic).toSet
         //Creating Direct Stream
val directKafkaStream = KafkaUtils.createDirectStream[String,
String,
             StringDecoder,StringDecoder] (<spark streaming
context>,<Kafka Broker Parameters>,consumerTopic)
         directKafkaStream.foreachRDD(rdd=> {
     //process records }
```

This converts the continuous stream of logs into micro matches within interval of three seconds which can be handled by Spark job.

As soon as the raw access logs are sent to this raw event Kafka topic, Spark streaming job parses the logs records and sends the processed result to another Kafka topic named *ndsloganalytics_processed_events* [29].

This topic in Kafka which represents the processed log events is created by following command.

sudo $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper 192.168.133.128:2181 --replication-factor 1 --partitions 1 --topic ndsloganalytics_processed_events --config retention.ms=7200000

A Kafka Streaming consumer client picks up the processed logs from the processed logs topic named *ndsloganalytics_processed_events* once spark streaming job finishes processing raw logs. Once the processed logs are read by the consumer job, it persists the data to InfluxDB in which data including IP address, httpRequest, httpMethod are treated as tags and httpRequest bytes is treated as field. Once data is persisted in InfluxDB it is continuously accessed by Grafana dashboard as shown in Figure 11. This dashboard can be configured to refresh itself in periodic interval so that dashboard is updated in real time [28].

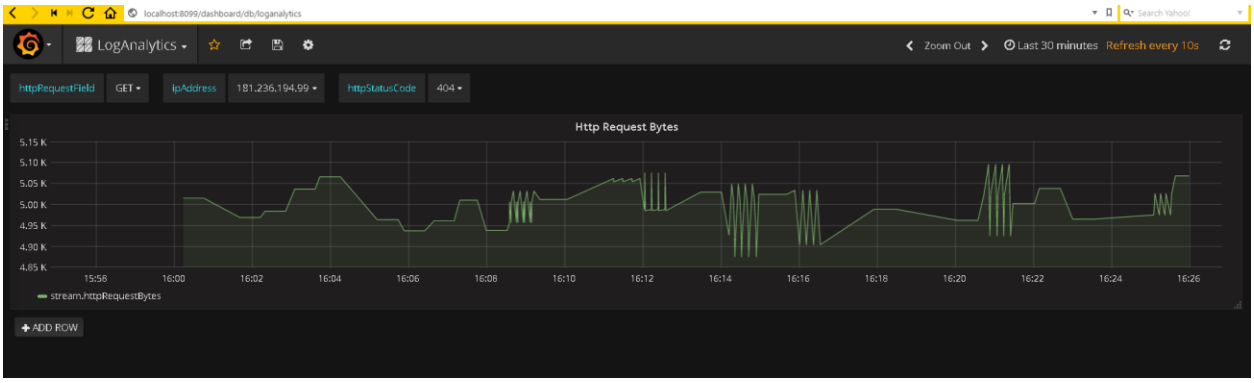

Figure 11. Grafana Dashboard Showing Access Logs.

5. COMPARISON AND TESTING

In this paper, time taken for processing the logs files using the Hadoop batch [31],Spark batch job [30] and Spark streaming [29] is measured. Before measuring the time, there is a data preparation phase in which data is prepared using the python script [26]. This python script is used to prepare the simulated log records with a count of 15 IP address and whose record counts range from 1 million to 11 million records. Data preparation time along with run time in seconds for both the Spark batch, Hadoop batch and spark streaming job is recorded and presented in table 2.

Table 2. Dataset Environment Properties.

Records	Data Preparation Time	Data SFTP time	Spark Total Run Time	Hadoop Batch Total Time	Spark Streaming Run Time
1	711	17	744	758.5	710
2	1326	27	1380	1390	1250
3	2066	36	2139.5	2147	2035
4	3084	48	3173.5	3187.5	3100
5	4379	60	4485.5	4505.5	4400
6	5010	72	5136	5158.5	5000
7	5460	85	5601	5632	5500
8	6200	99	6370	6391	6270
9	7520	112	7701.5	7737	7630
10	8820	128	9026.5	9056	8900
11	9915	142	10140	10168.5	10120

In this experiment access Log is processed twice using both the Spark and Hadoop batch job to calculate the average time taken for processing these records. Below shows the command to run the Hadoop MapReduce job to analyze 1 million records run the following command.

hadoop jar /home/hduser/loganalyticshadoop-1-jar-with-dependencies.jar com.nitendragautam.mapreduce.LogAnalytics /user/hduser/input/KafkaStreaming_1Mil.log /user/hduser/output/KafkaStreaming_1Mil

Below shows the command to run the Spark job to analyze the 1 million access logs record.

spark-submit --class com.nitendragautam.sparkbatchapp.main.Boot --master spark://192.168.133.128:7077 --deploy-mode cluster --supervise --executor-memory 4G --driver-memory 4G --total-executor-cores 2 /home/hduser/sparkbatchapp.jar /home/hduser/NDSBatchApp/input/KafkaStreaming_1Mil.log /home/hduser/NDSBatchApp/output/KafkaStreaming_1Mil

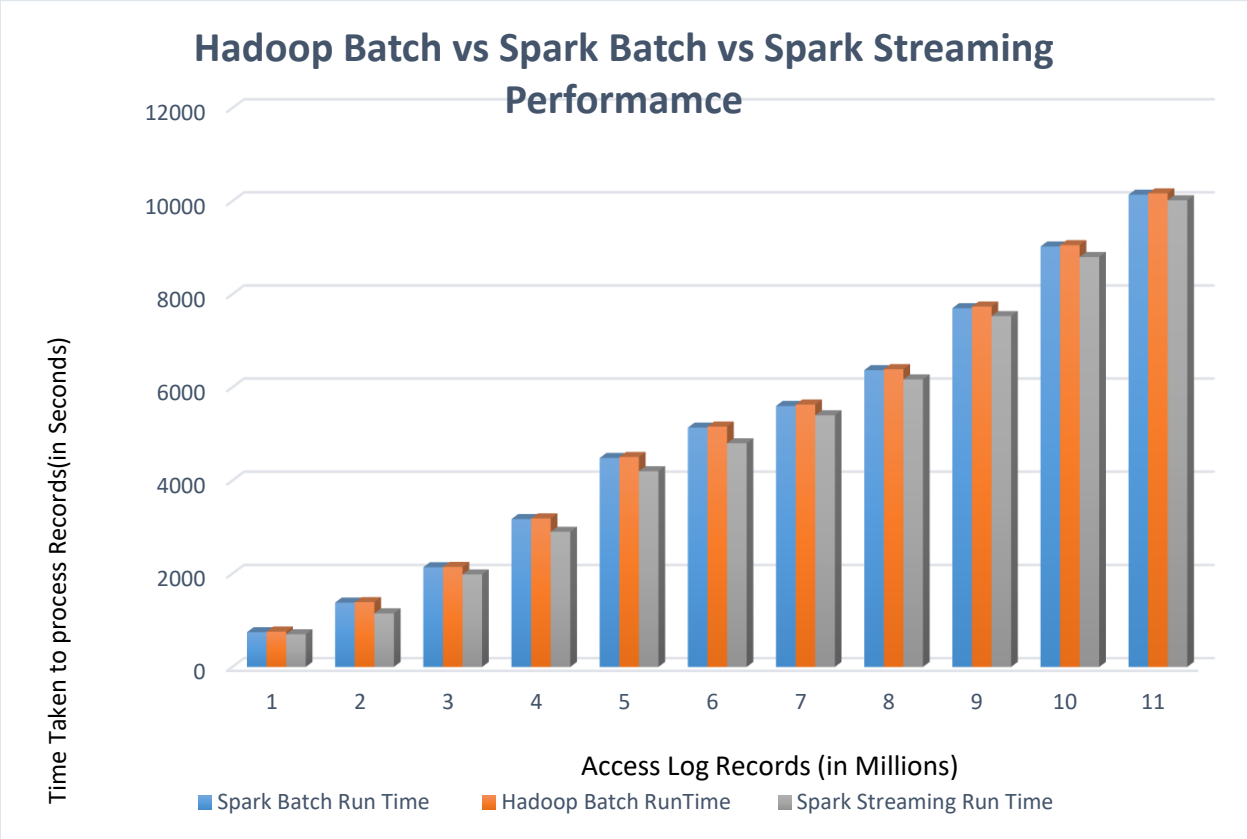

Figure 12. Hadoop Batch vs Spark Batch Response Time for Analyzing Access Logs.

Figure 12 gives the average time it takes to process access log record between Hadoop batch, Spark batch job and Spark streaming job for record count ranging from 1 million to 11 million. This time includes the data preparation time and SFTP time for Hadoop and Spark batch job during which data is prepared using the python script.

6. CONCLUSIONS AND FUTURE WORK

After carrying out the experiments for processing the log file from 1 million to 11 million records using all the architecture, results are observed in Figure 12. Based on this figure it can be concluded that, both Spark batch and Spark streaming has less run time than Hadoop MapReduce model while processing the access logs. This is because data needs to be shared across multiple MapReduce steps for a Hadoop job to execute it in a parallel and distributed fashion. Currently, the only way to share the data between the MapReduce jobs is by using a distributed file system such as HDFS. This adds an additional overhead because data is replicated across the Hadoop clusters, further increasing the disk I/O (Input/Output) activity.

MapReduce as part of the Hadoop platform was initially designed for batch processing of data that takes several hours and even days to process. This slowness of Hadoop pertains to the fact that the execution model of MapReduce jobs was not designed for fast execution. Even the scheduling, task assignment, transfer of code to Hadoop slaves and procedures for starting the Hadoop job are not designed to finish within few seconds. In MapReduce model, mapper process input data in a key-value pairs and are only able to process a single pair at a time. In this model, data is first written to local disk at the end of the map phase, and to HDFS at the end of reduce phase.

Hadoop based MapReduce model is divided among different phases such as input splits, mapper, shuffle & sort and reducers phase. Among these phases, shuffle and sort is a I/O intensive task in which intermediate results from the mapper function gets shuffled and sorted based on their key. This shuffling and sorting is a time-consuming task as it involves transferring the data throughput the network to different data nodes. Once this phase is completed reducer phase requires combining the data.

Spark uses a data abstraction layer called RDD, which can be stored in memory without additional I/O. As Spark does not have a map and reduce phase, data is typically read from disk when processing starts and written to disk when there is need to persist results. Spark RDD API can parallelly process multiple key-value pairs using partition.

Spark also provides fault tolerance on existing data using DAG (Direct Acyclic Graph) without the need of replicating it. Using DAG Spark can track and recompute the data from the disk or in-memory if data is lost during the job execution. Using an RDD-based paradigm, Spark provides both the streaming and batch-based platform with more read and write speed giving a challenge to challenging batch-based data processing platform using Hadoop.

This paper has taken a single node cluster to process a single source of data of same type for streaming data use case. Proposed streaming based architecture is scalable for multiple data source, which is the best fit for IOT related uses cases where you can have weather data, sensor data, and various other sources.

REFERENCES

[1] S. D. Patil, "Use of Web Log File for Web Usage Mining", International Journal of Engineering Research & Technology (IJERT), vol. 2, no. 4, Apr. 2013.

[2] V. Verma, A. Verma, and S. Bhatia, "Comprehensive Analysis of Web Log Files for Mining", International Journal of Computer Science Issues(IJCSI), vol. 8, Issue 6, pp. 199-202, 2011.

[3] N. K. Tyagi, A. K. Solanki, S. Tyagi, "An Algorithmic Approach to Data Preprocessing in Web Usage Mining", International Journal of Information Technology and Knowledge Management, vol. 2, no. 2, pp. 279-283, July-December 2010.

[4] I. Mavridis, E. Karatza, "Log File Analysis in Cloud with Apache Hadoop and Apache Spark", Proceedings of 2nd International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2015), Sept. 2015.

[5] J. Dave, "Three Approaches to Data Analysis with Hadoop", A Dell Technical White Paper, November 2013. [PDF]. Available: http://en.community.dell.com/cfs-file/__key/telligent-evolution-components-attachments/13-4491-00-00-20-43-79-41/Three-Approaches-to-Hadoop-Data-Analysis.pdf?forcedownload=true. [Accessed: Accessed: 16-April-2017]

[6] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters.", In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation (Berkeley, CA, USA, 2004), USENIX Association, pp. 10–10.

[7] S. Shahrivari, "Beyond Batch Processing: Towards Real-Time and Streaming Big Data", Computer, vol. 3, no. 4, pp. 117-129, 2014.

[8] W. Wolfram, F. Gessert, S. Friedrich, & N. Ritter, “Real-time stream processing for Big Data”, it -Information Technology, 58(4),2016, pp. 186-194.

[9] P. Parthiban and S. Selvakumar, “Big Data Architecture for Capturing, Storing, Analyzing and Visualizing of Web Server Logs”, Indian Journal of Science and Technology**,** Vol 9(4), DOI: 10.17485/ijst/2016/v9i4/84173, January 2016.

[10] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin, “A survey of open source tools for machine learning with big data in the Hadoop ecosystem”, Journal of Big Data, vol. 2, no. 1, May 2015.

[11] T. White, “Hadoop: the definitive guide”, O’Reilly Media, 4th edn.,2010

[12] “Welcome to Apache™ Hadoop®!”, Apache Hadoop. [Online]. Available: http://hadoop.apache.org/. [Accessed: -25-Feb-2017].

[13] C. Lam, *“*Hadoop in Action”, Manning Publications Co., 1st edition, 2010.

[14] “Apache Spark™ - Lightning-Fast Cluster Computing,” Apache Spark™ - Lightning-Fast Cluster Computing. [Online]. Available: http://spark.apache.org/. [Accessed: 16-Mar-2017].

[15] “Spark Streaming Programming Guide”, Spark Streaming - Spark 2.2.1 Documentation. [Online]. Available: http://spark.apache.org/docs/latest/streaming-programming-guide.html. [Accessed: 16-Mar-2017].

[16] S. M. Hernandez, “Near real time fraud detection with Apache Spark”, 2014-2015. [PDF]. Available: https://upcommons.upc.edu/bitstream/handle/2117/82311/Sergi_Martin_pfc.pdf?sequence=1&isAllowed=y. [Accessed: 25-Feb-2017].

[17] M. Zaharia "An architecture for fast and general data processing on large clusters", Feb. 2014. [PDF]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.pdf. [Accessed: 25-Feb-2017]

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. M. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. “Fast and interactive analytics over Hadoop data with Spark.”, USENIX;login:, 37(4):45–51, August 2012.

[19] “Apache Spark™ - Lightning-Fast Cluster Computing,” Apache Spark™ - Lightning-Fast Cluster Computing. [Online]. Available: http://spark.apache.org/. [Accessed: 16-Mar-2017].

[20] L.K. Joshila Grace, V.M.a.D.N., “Analysis of Web Logs and Web User in Web Mining”, International Journal of Network Security & Its Applications (IJNSA), Vol.3, No.1, 2011

[21] L, Ilkka, “Logging Web Behaviour for Association Rule Mining”, Helsinki Metropolia University of Applied Sciences, Master of Engineering, Master’s Thesis, 2 October 2015

[22] “Log Files”, Log Files - Apache HTTP Server Version 2.4. [Online]. Available: https://httpd.apache.org/docs/2.4/logs.html. [Accessed: 25-May-2017].

[23] “Extended Log File Format”, W3C: [Online]. Available: https://www.w3.org/TR/WD-logfile.html. [Accessed: 25-May-2017].

[24] “Grafana - The open platform for analytics and monitoring”, Grafana Labs. [Online]. Available: https://grafana.com/. [Accessed: 25-Jun-2017].

[25] “InfluxData Documentation”, InfluxData | Documentation. [Online]. Available: https://docs.influxdata.com/influxdb/v1.2/. [Accessed: 25-Jun-2017].

[26] N. Gautam, “Fake Apache Access Logs Generator”, GitHub. [Online]. Available: https://github.com/nitendragautam/Fake-Apache-Log-Generator. [Accessed: 19-May-2017].

[27] N. Gautam, “Kafka Streaming Producer”, GitHub. [Online]. Available: https://github.com/nitendragautam/KafkaStreamingProducerJob. [Accessed: 17-Jul-2017].

[28] N. Gautam, “Kafka Streaming Consumer”, GitHub. [Online]. Available: https://github.com/nitendragautam/KafkaStreamingConsumerJob. [Accessed: 09-Jul-2017].

[29] N. Gautam, “Spark Streaming Job”, GitHub. [Online]. Available: https://github.com/nitendragautam/SparkStreamingApp. [Accessed: 29-Jan-2018].

[30] N. Gautam, “Spark Batch App”, GitHub. [Online]. Available: https://github.com/nitendragautam/SparkBatchApp. [Accessed: 29-Jan-2018].

[31] N. Gautam, “Hadoop MapReduce Job”, GitHub. [Online]. Available: https://github.com/nitendragautam/LogAnalysisUsingHadoop. [Accessed: 8-Nov-2018].

[32] “Build powerful reactive, concurrent, and distributed applications more easily”, Akka | Akka. [Online]. Available: https://akka.io/. [Accessed: 25-April-2017].

[33] “Welcome to Apache ZooKeeper™”, Apache ZooKeeper - Home. [Online]. Available: https://zookeeper.apache.org/. [Accessed: 25-Apr-2017].

[34] “Chrome Logo”, Google. [Online]. Available: https://www.google.com/chrome/. [Accessed: 19-Jan-2018].

[35] “The new, fast browser for Mac, PC and Linux | Firefox Logo”, Mozilla. [Online]. Available: https://www.mozilla.org/en-US/firefox/. [Accessed: 19-Jan-2018].

[36] “macOS - Safari Logo”, Apple. [Online]. Available: https://www.apple.com/safari/. [Accessed: 28-Jan-2018].

[37] “Top 18 Open Source and Commercial Stream Analytics Platforms - Editor Review, User Reviews, Features, Pricing and Comparison in 2018”, Predictive Analytics Today, 06-Jun-2016. [Online]. Available: https://www.predictiveanalyticstoday.com/top-open-source-commercial-stream-analytics-platforms/. [Accessed: 28-Jan-2018].

APPENDIX A. KAFKA PRODUCER APPLICATION CODE

```
object Boot extends App{
       implicit val system = ActorSystem("KafkaActorSystem")
       implicit val materializer =ActorMaterializer()
       implicit val executionContext = system.dispatcher
       val filePath =
"D:\\App\\KafkaStreamingProducerJob\\Data\\KafkaStreaming.log"
       val logFilePath= Paths.get(filePath)
      val config = ConfigFactory.load()
       val producerConfig =config.getConfig("kafka.producer")
       val kafkaTopic =producerConfig.getString("topic")
       val producerSettings =ProducerSettings(system, new
StringSerializer, new
StringSerializer).withBootstrapServers(producerConfig.getString(
"bootstrap.servers"))
      val kafkaProducer = producerSettings.createKafkaProducer()
       //Creates a source of lines
       val lines :Source[String,NotUsed] =FileTailSource.lines(
         path=logFilePath,maxLineSize = 1000,pollingInterval =
5000.millis)
       lines.runForeach(line => {
         //Send Kafka Record for each Lines
         kafkaProducer.send(new
ProducerRecord[String,String](kafkaTopic,line))})}
```

APPENDIX B. KAFKA CONSUMER APPLICATION CODE

```
object Boot extends App {
 implicit val system = ActorSystem("ConsumerActorSystem")
 implicit val materializer =ActorMaterializer()
implicit val executionContext = system.dispatcher
     ConsumerActorService.apply(system)}


case class EventMessage(dateTime : String, clientIpAddress :
String, httpStatusCode : String,httpRequestField : String,
httpRequestBytes : String)
object ConsumerActorService {
      def apply(system :ActorSystem ): ActorRef ={
         val config =ConfigFactory.load()
val kafkaConsumerConf = KafkaConsumer.Conf( new
StringDeserializer, new StringDeserializer, bootstrapServers =
config.getString("akka.kafka.consumer.bootstrap.servers")
,groupId =
config.getString("akka.kafka.consumer.group.id"),enableAutoCommi
t = false,autoOffsetReset =
OffsetResetStrategy.LATEST).withConf(config)
val actorConf =KafkaConsumerActor.Conf(2.seconds,0.seconds)
         system.actorOf(Props(new
ConsumerActors(config,kafkaConsumerConf,actorConf)))}}
```

```
class ConsumerActors(config :Config
,kafkaConfig:KafkaConsumer.Conf[String,String],  actorConfig:
KafkaConsumerActor.Conf) extends Actor {
val influxdbRestService = new InfluxdbRestService
       val recordsExtractor = ConsumerRecords.extractor[String,
String]
       val kafkaTopic =
config.getString("akka.kafka.consumer.topic")
       val kafkaConsumerActor = context.actorOf(
         KafkaConsumerActor.props(kafkaConfig, actorConfig,
self))
       context.watch(kafkaConsumerActor)
       kafkaConsumerActor !
Subscribe.AutoPartition(List(kafkaTopic))
       def receive = {
         case recordsExtractor(records) => {
processRecords(records)
             sender() ! Confirm(records.offsets,commit = true)
} }


   def processRecords(records :ConsumerRecords[String ,String])
{records.values.foreach {
     case (value) => {        val gson = new
GsonBuilder().create()
```

```
     val kafkaMessage = gson.fromJson(value
,classOf[EventMessage])
       val dbName = config.getString("influxdb.dataBase")
     val point =
Point.measurement("stream").time(System.currentTimeMillis(),
TimeUnit.MILLISECONDS) .tag("httpStatusCode",
kafkaMessage.httpStatusCode) .tag("clientIpAddress",
kafkaMessage.clientIpAddress).tag("httpRequestField",
kafkaMessage.httpRequestField).addField("httpRequestBytes",
Integer.parseInt(kafkaMessage.httpRequestBytes)).build()
           influxdbRestService.writeDataInfluxDb(point, dbName)
    }}}}
     // Code for Manipulating InFluxDb database
     class InfluxdbRestService {
       val config = ConfigFactory.load()
       val influxDB
=InfluxDBFactory.connect(config.getString("influxdb.baseURL"),
           config.getString("influxdb.user"),
config.getString("influxdb.pass"))
       def createDatabase(dbName :String): Unit ={
         if(isDatabaseExists(dbName)==false){
influxDB.createDatabase(dbName)  }
       }
       // Query the Database and Gives the Results
```

```
       def queryDatabase(queryString :String ,dbName :String):
QueryResult={
         val query = new Query(queryString ,dbName)
         influxDB.query(query)}
       def writeDataInfluxDb(dataPoint :Point ,dbName :String):
Unit ={
         val batchPoints = BatchPoints.database(dbName)
           .tag("async","true").retentionPolicy("autogen")
.build()
         batchPoints.point(dataPoint)
         influxDB.write(batchPoints)}
       def isDatabaseExists(dbName :String): Boolean ={
         influxDB.databaseExists(dbName)   }}
```

APPENDIX C. SPARK STREAMING APPLICATION CODE

```
object Boot {
       def main(args: Array[String]) {
     val sr = new SparkServices
     sr.startSparkStreamingCluster()}}
class SparkServices extends Serializable{
val accessLogsParser = new AccessLogsParser  ; val dateFormat
="YYYY-MM-dd HH:MM:SS"
       def startSparkStreamingCluster(){
        val conf = new
SparkConf().setAppName("SparkStreamingApp")
val ssc = new StreamingContext(conf,Seconds(3))  ; val props =
getProducerProperties()
val kafkaSink = ssc.sparkContext.broadcast(ProducerSink(props))
//Broadcasting Kafka Sink
val kafkaTopic = "ndsloganalytics_raw_events"  ;
val producerTopic = "ndsloganalytics_processed_events"
         val consumerTopic = List(kafkaTopic).toSet
     val kafkaParams = Map[String,String]("metadata.broker.list"
-> "192.168.133.128:9093, 192.168.133.128:9094");
         //Create Direct Stream in spark to read data from Kafka
         val directKafkaStream
=KafkaUtils.createDirectStream[String, String,
```

```
            StringDecoder,StringDecoder]
(ssc,kafkaParams,consumerTopic)
directKafkaStream.foreachRDD(rdd=> rdd.foreachPartition(part
=>part.foreach(record => {
         val processedRecords
=accessLogsParser.parseAccessLogs(record._2)
              val clientIpAddress =
processedRecords.get.clientAddress
           val parsedDate = accessLogsParser.parseDateField
(processedRecords.get.dateTime)
              val httpStatusCode =
processedRecords.get.httpStatusCode
              val httpRequestField =accessLogsParser
.parseHttpRequestField(processedRecords.get.httpRequest).get._1
              val httpRequestBytes =
processedRecords.get.bytesSent
val kafkaMessage = new
EventMessage(convertDateFormat(parsedDate.get, dateFormat)
,clientIpAddress, httpStatusCode , httpRequestField ,
httpRequestBytes)
              val messageString = (new
Gson).toJson(kafkaMessage)
```

```
kafkaSink.value.sendMessageToKafka(producerTopic,messageString)}
)))
          ssc.start()
          ssc.awaitTermination()  }

def getProducerProperties(): HashMap[String, Object] ={
          val props = new HashMap[String, Object]()
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"192.168.133.128:9093")
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,

"org.apache.kafka.common.serialization.StringSerializer")
          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,

"org.apache.kafka.common.serialization.StringSerializer")
          props
        }
        // Converts date into Given Format
        def convertDateFormat(dateString :Date ,dateFormat
:String): String ={
          val format = new SimpleDateFormat(dateFormat)
          format.format(dateString)
        }}
```

```
     //Logic for Parsing Access Logs
     class AccessLogsParser extends Serializable{
       //Regex for Logs Data
       private val ddd = "\\d{1,3}"
       private val clientIpAddress =
s"($ddd\\.$ddd\\.$ddd\\.$ddd)?"   //Eg 192.168.138.1
       private val clientIdentity = "(\\S+)" ;  private val
remoteUser  = "(\\S+)" ;
       private val dateTime = "(\\[.+?\\])" ;  private val
httpRequest = "\"(.*?)\"" ;
       private val httpStatus = "(\\d{3})";  private val
requestBytes = "(\\S+)" ;
       private val siteReferer = "\"(.*?)\"";  private val
userAgent  = "\"(.*?)\""
       private val accessLogsRegex = s"$clientIpAddress
$clientIdentity $remoteUser $dateTime $httpRequest $httpStatus
$requestBytes $siteReferer $userAgent"
       private val pattern =Pattern.compile(accessLogsRegex)

        //Parses Access Logs by passing Single Line
       def parseAccessLogs(logRecord :String):
Option[AccessLogRecord] ={
         val regexMatcher=pattern.matcher(logRecord)
```

```
        if(regexMatcher.find){ //If Pattern in matched
          Some(buildAccessLogRecord(regexMatcher))
        }else{
          None //No Pattern is Matched  }}

      /*Parses the record but returns null object version of
Access LogRecord if the parsing of the log fails*/
      def parseNullRecordsonFailure(logRecord :String):
AccessLogRecord ={
        val patternMatcher = pattern.matcher(logRecord)
        if(patternMatcher.find){
          buildAccessLogRecord(patternMatcher)
        }else{ //If No matches Found
          AccessLogRecord("","","","","","","","","","") //Null
Access Record
        }}

      private def buildAccessLogRecord(matcher :Matcher)={
        AccessLogRecord(  matcher.group(1),   matcher.group(2),
matcher.group(3),  matcher.group(4), matcher.group(5),
matcher.group(6), matcher.group(7),   matcher.group(8),
matcher.group(9)  )
      }
```

```
       def parseHttpRequestField(httpRequest
:String):Option[Tuple3[String,String,String]] ={
         val splittedArray= httpRequest.split(" ") //Split the
Request based on Space
         if (splittedArray.size == 3)
Some((splittedArray(0),splittedArray(1),splittedArray(2)))
         else   None   }


  def parseDateField(dateField :String): Option[Date] ={
         val dateFormat ="\\[(.*?) .+]"
         val datePattern =Pattern.compile(dateFormat) //Using
Regex to compile the pattern
         val dateMatcher =datePattern.matcher(dateField)
         if(dateMatcher.find){
           val dateString = dateMatcher.group(1) //Match the
Date
val dateFormat = new
SimpleDateFormat("dd/MMM/yyyy:HH:mm:ss",Locale.ENGLISH
allCatch.opt(dateFormat.parse(dateString))  //Returns Option
[Date] //Catches All Exception
         }else{None  }}}
```

```
case class AccessLogRecord(clientAddress :String, clientIdentity
:String, remoteUser :String , dateTime :String,httpRequest
:String ,httpStatusCode :String , bytesSent :String,
siteReferer :String,  userAgent: String  )

case class EventMessage(dateTime : String,clientIpAddress :
String, httpStatusCode : String, httpRequestField : String,
httpRequestBytes : String)
```

APPENDIX D. SPARK BATCH APPLICATION CODE

```
 object Boot {
       def main(args: Array[String]) {
      val inputFile = args(0) //Input File ; val outputFile =
args(1) //Output File Location
     val sr = new SparkServices ;
sr.startSparkBatchCluster(inputFile ,outputFile) }}
class SparkServices extends Serializable{
val accessLogsParser = new AccessLogsParser;  val tapiStart =
System.currentTimeMillis()
       def startSparkBatchCluster(inputFile :String, outputFile
:String) {
         val conf = new SparkConf().setAppName("SparkBatchApp"
+getTodaysDate())
         val sc = new SparkContext(conf);       val accessLogs =
sc.textFile(inputFile)
         val accessLogsRDD =
           accessLogs.map(f =>
accessLogsParser.parseAccessLogs(f)).cache()
         //Calculate the Client IP address which came more than
10 times
         val ipAddress = accessLogsRDD.map(_.clientAddress ->
1L)
           .reduceByKey(_ + _) //add the number of Occurens
```

```
            .cache()
          ipAddress.coalesce(1).saveAsTextFile(outputFile)
          sc.stop() //Stopping Spark batch
          val tapiEnd =System.currentTimeMillis()
          val elaspedTimeApi = (tapiEnd -tapiStart)/1000
        }
        private def getTodaysDate(): String ={
          val dateFormat = new SimpleDateFormat("yyyy-MM-dd-HH-
mm-ss")
          val cal = Calendar.getInstance();
cal.add(Calendar.DATE,0)
          dateFormat.format(cal.getTime())    }}
      class AccessLogsParser extends Serializable{
        private val ddd = "\\d{1,3}"        //At least 1 but not
more than 3 times eg:192 or 92
        private val clientIpAddress =
s"($ddd\\.$ddd\\.$ddd\\.$ddd)?"   //Eg 192.168.138.1
        private val clientIdentity = "(\\S+)";     private val
remoteUser  = "(\\S+)"
        private val dateTime = "(\\[.+?\\])";    private val
httpRequest = "\"(.*?)\""
        private val httpStatus = "(\\d{3})";  private val
requestBytes = "(\\S+)"
```

```
       private val siteReferer = "\"(.*?)\"" ;  private val
userAgent  = "\"(.*?)\""
       private val accessLogsRegex = s"$clientIpAddress
$clientIdentity $remoteUser $dateTime $httpRequest $httpStatus
$requestBytes $siteReferer $userAgent"
       private val pattern =Pattern.compile(accessLogsRegex)


   def parseAccessLogs(logRecord :String): AccessLogRecord ={
         val regexMatcher=pattern.matcher(logRecord)
         if(regexMatcher.find){ //If Pattern in matched
          buildAccessLogRecord(regexMatcher)
         }else{
           AccessLogRecord("","","","","","","","","") //No
Pattern is Matched
         } }
       def parseNullRecordsonFailure(logRecord :String):
AccessLogRecord ={
         val patternMatcher = pattern.matcher(logRecord)
         if(patternMatcher.find){
           buildAccessLogRecord(patternMatcher)
         }else{ AccessLogRecord("","","","","","","","","")
//Null Access Record  }}
       private def buildAccessLogRecord(matcher :Matcher)={
```

```
        AccessLogRecord(matcher.group(1), matcher.group(2),
matcher.group(3), matcher.group(4), matcher.group(5),
matcher.group(6),  matcher.group(7),  matcher.group(8),
matcher.group(9))
       }
  def parseHttpRequestField(httpRequest
:String):Option[Tuple3[String,String,String]] ={
        val splittedArray= httpRequest.split(" ") //Split the
Request based on Space
        if (splittedArray.size == 3)
Some((splittedArray(0),splittedArray(1),splittedArray(2)))
        else {  None}}
       def parseDateField(dateField :String):
Option[java.util.Date] ={
        val dateFormat ="\\[(.*?) .+]"  ; val datePattern
=Pattern.compile(dateFormat)
        val dateMatcher =datePattern.matcher(dateField)
        if(dateMatcher.find){val dateString =
dateMatcher.group(1) //Match the Date
  val dateFormat = new
SimpleDateFormat("dd/MM/yyyy:HH:mm:ss",Locale.ENGLISH)
 allCatch.opt(dateFormat.parse(dateString))  //Returns Option
[Date] //Catches All Exception
        }else{None }}}
```

```
    //Domain Class for Access Logs which holds the Access Logs
    case class AccessLogRecord(clientAddress :String,
clientIdentity :String, remoteUser :String ,dateTime :String,
httpRequest :String , httpStatusCode :String , bytesSent
:String, siteReferer :String, userAgent: String )
```

APPENDIX E. HADOOP MAPREDUCE APPLICATION CODE

```
public class LogAnalyticsHadoop {
 public static void main(String args[]) throws Exception{
Path inputPath = new Path(args[0]);  Path outputPath = new
Path(args[1]);
     Configuration conf = new Configuration(); //Hadoop Config
     Job mapreduceJob = Job.getInstance(conf
,"LogAnalyticsHadoop");
     mapreduceJob.setJarByClass(LogAnalyticsHadoop.class);
     mapreduceJob.setMapperClass(LogAnalyticsMapper.class);
     mapreduceJob.setCombinerClass(LogAnalyticsReducer.class);

mapreduceJob.setReducerClass(LogAnalyticsReducer.class);
             mapreduceJob.setOutputKeyClass(Text.class); //Ouput
Key Type

mapreduceJob.setOutputValueClass(IntWritable.class);
             FileInputFormat.addInputPath(mapreduceJob
,inputPath);
             FileOutputFormat.setOutputPath(mapreduceJob
,outputPath);
             System.exit(mapreduceJob.waitForCompletion(true) ?
0 :1); }}
     // Mapper Class which takes log line as Input and Parses it
```

```
     public class LogAnalyticsMapper extends Mapper<Object,
Text, Text, IntWritable> {
         private IntWritable clientAddressCount = new
IntWritable(1);
         AccessLogsParser  accessLogsParser = new
AccessLogsParser();
public void map(Object key, Text value, Context context)  throws
IOException, InterruptedException  {
               String logRecord = value.toString();
 AccessLogsRecord accessLogRecord =
accessLogsParser.parseAccessLogs(logRecord);
              //Parsing the IP address and setting the count of
IP address as one
             context.write(new
Text(accessLogRecord.getClientAddress()), clientAddressCount);
         }}
  public class LogAnalyticsReducer extends
             Reducer<Text,IntWritable,Text ,IntWritable>{
         private IntWritable result = new IntWritable();
         public void reduce(Text key ,Iterable<IntWritable>
values
                 ,Context context) throws IOException ,
InterruptedException{
     int sumOfIPAddressOccurance = 0;
```

```
//Iterating every IP address and  Counts the add the occurence
of every Ip Address
            for (IntWritable val : values) {
sumOfIPAddressOccurance += val.get(); }
            result.set(sumOfIPAddressOccurance);
            context.write(key ,result);
        }}


     public class AccessLogsParser {
         public AccessLogsRecord parseAccessLogs(String
logRecord){
            AccessLogsRecord accessLogRecord =null;
            Pattern pattern = getLogsPattern(); //Gets the
Access Logs Pattern
            Matcher regexMatcher =pattern.matcher(logRecord);
     //If Pattern is Macthed
if(regexMatcher.find()==true){  accessLogRecord =
buildAccessLogRecord(regexMatcher);
            }else{ //Return Null Access Log Record
                accessLogRecord=new
AccessLogsRecord("","","","","","","","","");
            }
            return accessLogRecord;}
private AccessLogsRecord buildAccessLogRecord(Matcher matcher){
```

```
  return new AccessLogsRecord(matcher.group(1),
matcher.group(2),    matcher.group(3),
matcher.group(4),matcher.group(5), matcher.group(6),
matcher.group(7), matcher.group(8), matcher.group(9));}

  public Triple getHttpRequestField(String httpRequest){
Triple triple;    String splittedArray[] =httpRequest.split("
");//Split the Request
           if(splittedArray.length ==3){
         triple = new
ImmutableTriple(splittedArray[0],splittedArray[1],splittedArray[
2]);
             }else{  triple = new ImmutableTriple("","","");  }
             return triple;}

         public String getDateField(String dateField){
             String dateFormat = "([\\w:/]+\\s[+\\-]\\d{4})";
String parsedDate=null;
             Pattern datePattern = Pattern.compile(dateFormat);
             Matcher dateMatcher =
datePattern.matcher(dateField);
             if(dateMatcher.find()){  //If True
                 String dateString = dateMatcher.group(1);
//Match the Date
```

```
SimpleDateFormat localDateFormat = new
SimpleDateFormat("dd/MMM/yyyy:HH:mm:ss Z");
                try {  long epochTime
=localDateFormat.parse(dateString).getTime();
               SimpleDateFormat sdf = new
SimpleDateFormat("YYYY-MM-dd HH:MM:SS");
                    parsedDate=sdf.format(new Date(epochTime));
                }catch(ParseException e){  e.printStackTrace();
}
            }else{ parsedDate=null;  }
            return parsedDate; }

  private Pattern getLogsPattern(){
            //At least 1 but not more than 3 times eg:192 or 92
String clientIpAddress =  "^([\\d.]+) ";
  String clientIdentity = "(\\S+) ";
 String remoteUser  = "(\\S+) ";
String dateTime = "\\[([\\w:/]+\\s[+\\-]\\d{4})\\] ";
String httpRequest = "\"(.+?)\" ";
String httpStatus = "(\\d{3}) ";
 String requestBytes = "(\\d+) ";
String siteReferer = "\"([^\"]+)\"
String userAgent  =  "\"([^\"]+)\"";
//User Agents
```

```
String accesLogRegex = clientIpAddress +clientIdentity+
remoteUser + dateTime + httpRequest + httpStatus+requestBytes +
siteReferer+userAgent;
Pattern pattern = Pattern.compile(accesLogRegex);
            return pattern;  }}
public class AccessLogsRecord {
private String clientAddress;  private String clientIdentity;
private String remoteUser;
private String dateTime;  private String httpRequest;    private
String httpStatusCode;
 private String bytesSent;  private String siteReferer;
private String userAgent;   }
```