**RFID Analysis Using Apache Hadoop and Spark**

**Bishoy Magdy - Hesham Adel - Omar Sourour**

**Abstract**

This is an era of Big Data. Big Data is driving radical changes in traditional data analysis platforms. To perform any kind of analysis on such voluminous and complex data, scaling up the hardware platforms becomes imminent and choosing the right hardware/software platforms becomes a crucial decision if the user’s requirements are to be satisfied in a reasonable amount of time. Researchers have been working on building novel data analysis techniques for big data more than ever before which has led to the continuous development of many different algorithms and platforms.

Our project utilizes the power of advance big data analytics frameworks to provide real-time statistics over large datasets received from large groups of RFID sensors that records the movement of visitors in a huge conference.

**Chapter 1: Introduction**

Real-time Big Data Analytics – RFID Data Mining using Apache Spark is a project that presents a service for any big-scale party that is interested in organizing events that will be attended by numerous individuals and analyze their movements and interests. This service, although it may seem trivial, is very important for any corporate that looks forward to use the power of big data to learn the behavior and actions of its customers. Its application is very diverse, from the analysis of customers’ buying habits to the analysis of criminal behavior and action, the possibilities are endless.

The desired output of our project is to produce an environment complete with web interface and backend server connected to Spark Cluster that provides Real-time analysis, that is, the end-user will be able to see the analysis of his data instantly. Our main focus in this project was to discover the true potential of Big Data Technology and discover the most trending tools in this fields so we tried two approaches to get this project done(Apache Hadoop Technology, Apache Spark Technology) History of Big Data

**1.1: Used Tools**

**1.1.1: Apache Hadoop**

Apache Hadoop is an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures (of individual machines or racks of machines) are commonplace and thus should be automatically handled in software by the framework.

The core of Apache Hadoop consists of a storage part (Hadoop Distributed File System (HDFS)) and a processing part (MapReduce). Hadoop splits files into large blocks and distributes them amongst the nodes in the cluster. To process the data, Hadoop MapReduce transfers packaged code for nodes to process in parallel, based on the data each node needs to process. This approach takes advantage of data locality nodes manipulating the data that they have on hand—to allow the data to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are connected via high-speed networking.

The base Apache Hadoop framework is composed of the following modules:

1. Hadoop Common – contains libraries and utilities needed by other Hadoop modules;
2. Hadoop Distributed File System (HDFS) – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
3. Hadoop YARN – a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications; and
4. Hadoop MapReduce – a programming model for large scale data processing.

**1.1.2: Apache Spark**

Apache Spark is an open-source cluster computing framework originally developed in the AMPLab at UC Berkeley. In contrast to Hadoop's two-stage disk-based MapReduce paradigm, Spark's in-memory primitives provide performance up to 100 times faster for certain applications. By allowing user programs to load data into a cluster's memory and query it repeatedly, Spark is well suited to machine learning algorithms. Spark is a fast and general cluster computing system for Big Data. It provides high-level APIs in Scala, Java, and Python, and an optimized engine that supports general computation graphs for data analysis. It also supports a rich set of higher-level tools including Spark SQL for SQL and DataFrames, MLlib for machine learning, GraphX for graph processing, and Spark Streaming for stream processing.

**1.1.3: MapReduce**

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine

failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

**1.1.4: Other Technologies**

Other technologies used were to interface with Hadoop and Spark, such as Java EE, create the web interface, using HTML, JavaScript and CSS. Evenmore, we used Ruby as the server side language for our web interface.

**Chapter 2: RFID Analysis Using Hadoop**

**2.1: Introduction**

In the previous chapter, we gave a brief description on what Big Data is exactly. We outlined two Apache endorsed technologies used for big data analysis and the main idea of how they work. In this chapter we discuss our first assigned task. In this specific task we used Apache Hadoop to analyze the supplied dataset.

In the following sections we will dive into more details of our the task required and the way we tackled it. Section 2.2 describes our task in details, including the input and outputs required. Section 2.3 describes the main program flow and the key elements at play in the data pipeline. Next Section 2.4 delves into the details of each of the key elements marked in Section 2.3. Finally we end this chapter by displaying our results in Section 2.5.

**2.2: Task Description**

In the first task of our project, it was required to analyze a data set of RFID tags embedded in the doors of a client company, in order to retrieve some valuable statistics that can be used by our client. It was also required to use Apache Hadoop in order to handle the large amounts of data records, setup on a multi node cluster. What that means is to have multiple computers (nodes) running map-reduce tasks in parallel in order to decrease the processing time of the data. We note that since we are using the Hadoop framework, the data must be served in batches, that is, the data must already be saved (offline) and not coming in at real-time.

**2.2.1: Input Format**

The supplied data set is in the form of a Comma Separated Values (CSV) file. Each row is an independent record and is considered a data point. The row format was as follows:

Date, Time, Event Type, Number, Name, Access, Relay

Where the following keys map to:

* **Date/Time:** The date time the reading was made.
* **Event Type:** The module that triggered the read. We are only interested in data points where the module was of type Card.
* **Number:** The unique ID of the module that triggered the reading.
* **Name:** The name of the owner of the module.
* **Access:** Whether the person is granted access to enter or leave or not.
* **Relay:** Relay 2 corresponds to entrance while Relay 1 is vice versa.

**2.2.2: Statistics (Output) Required:**

The required statistics to calculate in this task were as follows:

* The number of departures and entrances per day.
* The number of departures and entrances per employee.
* The duration of stay per day of each employee in the company.

It was also required to visualize the results of the calculations and statistics as graphs in a user-friendly web interface.

**2.3: Program (Data) Flow**

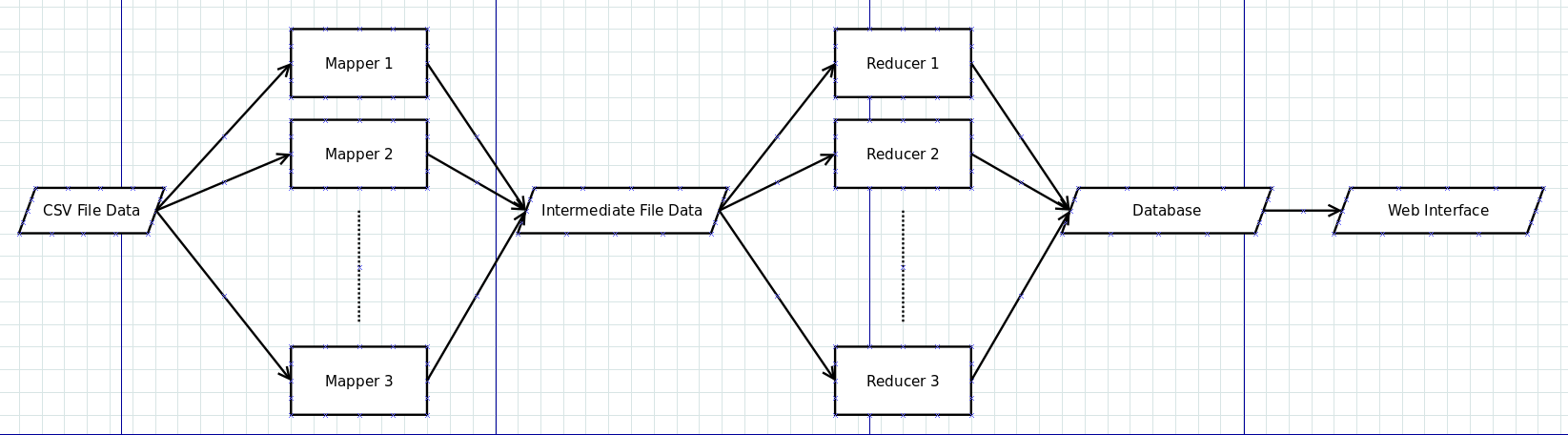
In this section, we will describe the main data flow, or pipeline through our program from the start of the reading in the date to the point of visualizing the statistics produced. In the heart of our program, lies the Hadoop Map-Reduce framework, which we use as our main tool to process the supplied data set.

To create a Hadoop Map-Reduce task, we need to do the following:

* Create a mapping function. This contains the logic that the mapping phase will execute.
* Create a reducing function. This contains the logic that the reducing phase will execute.
* Configure the Hadoop cluster to submit the MapReduce jobs over multiple nodes.

As the program starts to run the, data is read using the Hadoop framework. The configured number of mapper tasks start reading and processing the file in parallel and perform the mapping function as supplied. An intermediate file is generated with the results of the mapping phase. This file is used as input to the reducing phase, where the results of the mapping phase are aggregated to produce the final desired output.

As soon as each output data point was generated, it was written to its corresponding table in a MySQL database that was created specifically for this task. Each data point corresponds to a row in the database. These rows would later on be served using a server side language in our web application to populate the graphs. This is the final output that the user sees. The following diagram quickly visualizes the process.



**2.4: Program Structure**

In the previous section, we highlighted the main key points that our data passes through throughout the program. In this section we will talk about these key points in more detail.

**2.4.1: The Map Function**

The mappers, which execute the map function, are the first stage that the data goes through. The map function's job is to extract a key/value pair from each line of data. Similar keys throughout the data set are aggregated together, so that each key has a corresponding list of values, where these values were found in the data set.

The mapping is an important phase, as it prepares the foot set for the reducer to processes each list of values for each unique key produced by the mapper.

Since the task required 3 statistics, our program needed 3 different mapper functions that performed different logic for each of the required statistics. The following paragraph outlines the logic of our mapper functions:

* **Arrival/Departure Count per day**
  + The mapper function that we designed for this requirement, took the date as the unique key to be produced and the assigned a value of 1 for each occurrence of the key. For example, the following data row produces: <3/28/2015, 1>

3/28/2015,7:17 AM, Card, 5334, 219 Gysord, Admit, Relay 2

* **Number of Departures and Entrances Per Employee**
  + The mapper function for this statistic is similar to the one before it, but in addition to the date, the ID of the employee was also added. This ensures the uniqueness of the combination of the day/employee-id key. The value to be recorded for each occurrence is also 1. Following our previous example, the output would be: <3/28/2015-219 Gysord, 1>.
* **The Duration of Stay of Each Employee**
  + The mapper function for this statistic also contains the date and employee ID as the key, but the values assigned for each data point, is the time and the relay. To clarify, taking our previous example once again:

<3/28/2015-219 Gysord, 7:17 AM-Relay 2>

**2.4.2: The Reduce Function**

The second stage the data passes through after being mapped and saved in an intermediate file, is the reducing stage. A number of reducers process the mapped <key,List<values>> outputs that were produced by the mapping phase. The purpose of the reducing phase is to apply some aggregation the list data in order to produce one final representative value for each unique key. Similar to the mapping functions, we needed to create 3 reducer functions for each statistic. It so happens that two of these reducer functions share the same implementation. We will point this out in the following paragraph.

* **Arrival/Departure Count per day**
  + The reducer function to *reduce* the list of values for each key to a single value, was a simple summation. As was mentioned before, the mapper function produced a list of values (all of them being ones). By summing the list of ones, this effectively gives us the number of occurrences of this particular key in the original data point, hence the number of registered reads by the RFID, and hence the total number of entrances or departures in that day.
* **Number of Departures and Entrances Per Employee**
  + The reduce of this statistic is similar to the just mentioned implementation.
* **The Duration of Stay of Each Employee**
  + The reducer implementation of this statistic is a bit more complex. The list of times in the list are conveniently sorted by the mapping phase. Thus the reducing phase, iterates through the list reading off each two pairs of time values. For each iteration "i", the i'th value is subtracted from the i+1'th value to give the time the employee stayed within that interval pair. Since the employee can enter and leave the company many times, it is expected to find multiple pairs in the list. It is also expected to find an even number of entries in each key's list. If that is not the case, the dataset is erroneous.

**2.4.3: The Database and Web Interface**

Finally after the data has been successfully reduced, the output is saved in a database. The database contains 3 tables, a table for each statistic. As each data point is produced from the reduce stage, it is saved in the MySQL database. A web interface, implemented using HTML, CSS, JavaScript and Ruby as a server-side language, uses the database to pull the results and display them in a pleasing graphical interface.

**Chapter 3: RFID Analysis Using Apache Spark**

**3.1: Introduction**

There are some main concepts that we need to shed some light on before we delve into the details of the task itself. Like any framework, Spark depends on some main building blocks for its pipeline of execution. We will mention some of the most important points in the following paragraphs.

At a high level, every Spark application consists of a driver program that runs the user’s main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

A second abstraction in Spark is shared variables that can be used in parallel operations. By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared variables: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only “added” to, such as counters and sums.

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis or TCP sockets can be processed using complex algorithms expressed with high-level functions like MapReduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark’s machine learning and graph processing algorithms on data streams.

**3.2: Task Description**

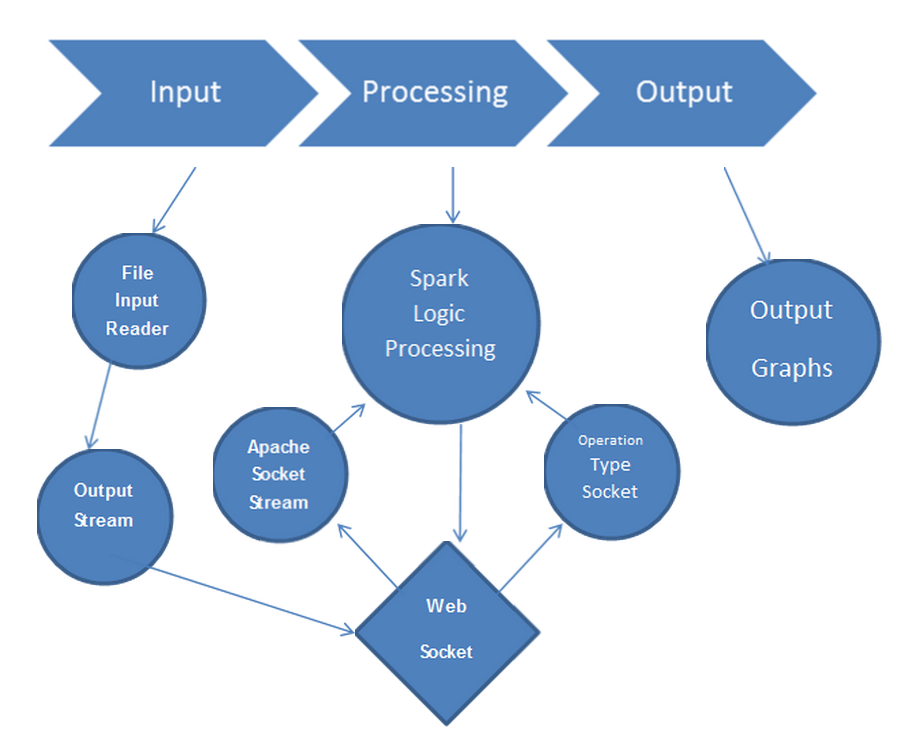
Apache Spark run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. Spark Streaming brings Spark's language-integrated API to stream processing, letting you write streaming jobs the same way you write batch jobs. It supports Java, Scala and Python. Even more, Spark Streaming recovers both lost work and operator state (e.g. sliding windows) out of the box, without any extra code on your part.

We decided to capitalize on these features and use them to provide real-time on-the-spot analysis by opening a websocket between the client and the Spark streaming socket which will also respond with the result through the full-duplex websocket.

**3.3: Program Flow**

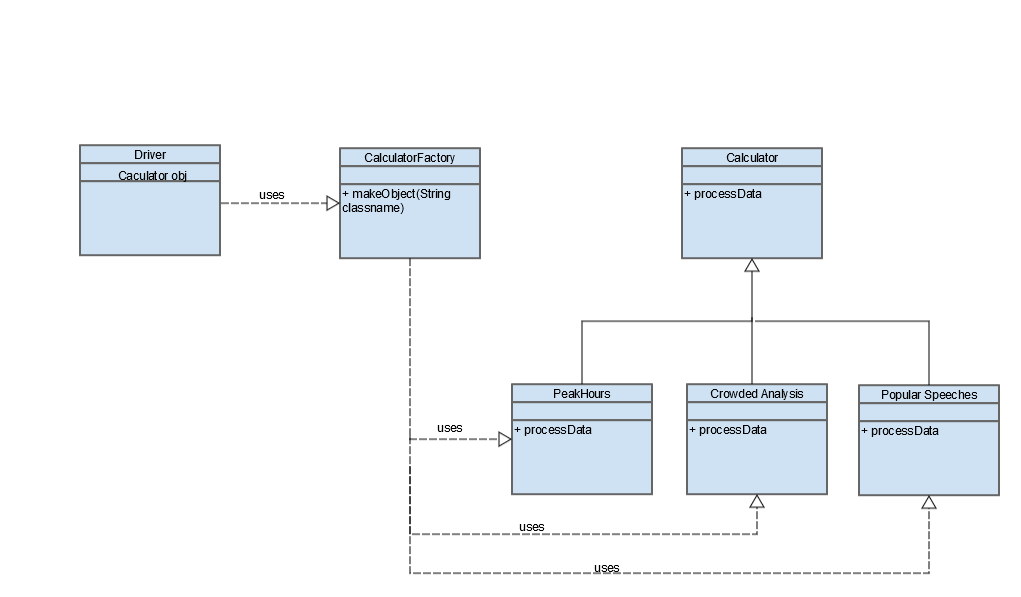
The user will upload the data by simply dragging and dropping the file into the specified area. Then the client side code will send each entity of the data through the full-duplex web socket to the server which will send it to Apache Spark Java program. Apache Spark opens two Sockets one for the analytical operation name and through it the user sends which operation he wants to perform on the data he will

provide and the other socket is for streaming the data to the RDD stream after each reduce task Spark program will send the result to Ruby-on-Rails which in turn will send it to the client.



**3.4: Program Structure**

Our Spark program consists of a main Driver class that makes a reference to the Calculator abstract class and initialize it to an object of PeakHours or CrowdedAnalysis or PopularSpeeches classes, then it call the process data method on this object. Each of the three later classes overrides the processData() method. In the processData() method the Spark RDD stream listens to the socket for input. When the input arrives it starts to preform mapReduce Operation and push it to the server.



**3.5: Demo**

A demo will be shown