



ECOLE  
**POLYTECHNIQUE**  
DE BRUXELLES



UNIVERSITAT POLITÈCNICA  
DE CATALUNYA  
BARCELONATECH

# Near real time fraud detection with Apache Spark

Universitat politècnica de Catalunya  
Université libre de bruxelles

**Sergi Martín Hernández**

Director  
Gianluca Bontempi

Supervisor  
Josep Vidal

Academic year  
2014 - 2015



# Abstract

Nowadays, there is a huge amount of information locked up in databases and these information can be exploited with machine learning techniques, which aim to find and describe structural patterns in data. After a clear dominance of Hadoop MapReduce as a Cloud Computing engine, Spark emerged as its successor because works in memory. This fact improved vastly the performance of machine learning algorithms and streaming applications.

Our work is focused on create an application capable of mix both concepts to make the most out of them. Hence, we assumed that a Fraud Detection scenario could be a great opportunity to show up Spark's capabilities.

In order to visualize and model the scalability of the application, we ran several and extensive performance tests at the university computing center, where we had all the necessary tools in the cluster to make it possible. The results mostly matched our expectations and theoretical analysis, when the application ran over the input transactions.



# Acknowledgments

Many people contributed to this experience. First of all, I am very grateful to my parents for supporting me during this journey since the beginning. They have given me all the education and love to make me success this first part of my life.

I want to thank Universitat Politècnica de Catalunya (UPC) for giving me the opportunity to spend my bachelor thesis abroad and Professor Gianluca Bontempi for hosting me in the Machine Learning Group (MLG) at Université Libre de Bruxelles (ULB). During the year they steadily followed my work and gave me important feedbacks. Among MLG members a special thank goes to Claudio and Fabrizio for advices and guidance.

This thesis is the last chapter of a great adventure in UPC, which would not have been the same without all my friends and coursemates I met during my studies.

I would like to finally mention my girlfriend Laura, since I met her five years ago she has been by my side in the good and the bad moments during my studies and I shared with her one of the most beautiful moments in my life.



# Contents

<b>Abstract</b>	<b>I</b>
<b>Acknowledgements</b>	<b>II</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Big Data . . . . .	1
1.1.1 Definition . . . . .	1
1.1.2 The four V's of Big Data . . . . .	1
1.1.3 Big Data platforms . . . . .	2
1.2 Machine Learning . . . . .	5
1.2.1 Definition . . . . .	5
1.2.2 Types of machine learning algorithms . . . . .	5
1.3 Thesis work description . . . . .	6
1.4 Thesis structure . . . . .	7
<b>2 State of the art</b>	<b>8</b>
2.1 Apache Spark . . . . .	8
2.1.1 History of Spark . . . . .	8
2.1.2 Introduction to Spark . . . . .	8
2.1.3 Basic concepts . . . . .	9
2.1.4 Scalability . . . . .	12
2.1.5 First example . . . . .	13
2.1.6 Machine Learning Library (MLLIB) . . . . .	14

---

2.1.7	Spark Streaming . . . . .	15
2.2	Data integration . . . . .	17
2.2.1	Apache Flume . . . . .	17
2.2.2	Apache Kafka . . . . .	19
2.3	HBase . . . . .	22
2.4	Scala . . . . .	23
2.4.1	Why Scala API? . . . . .	24
2.4.2	History of Scala . . . . .	25
2.4.3	What does Scala offer us? . . . . .	25
2.4.4	First look at Scala code . . . . .	25
<b>3</b>	<b>Random Forest</b>	<b>27</b>
3.1	Introduction . . . . .	27
3.2	Decision Trees . . . . .	27
3.3	Ensembles of classifiers . . . . .	29
3.3.1	Bagging . . . . .	29
3.3.2	Boosting . . . . .	30
3.3.3	Random Forest . . . . .	30
<b>4</b>	<b>Application description</b>	<b>34</b>
4.1	Introduction . . . . .	34
4.2	Summary . . . . .	35
4.3	A deeper look into the application . . . . .	36
4.3.1	Dashboard part . . . . .	36
4.3.2	Dictionary part . . . . .	37
<b>5</b>	<b>Deployment</b>	<b>40</b>
5.1	IRIDIA cluster . . . . .	40
5.1.1	Hardware setup . . . . .	40
5.1.2	Software setup . . . . .	41



5.2	Cluster manager . . . . .	42
<b>6</b>	<b>Results and scalability</b>	<b>44</b>
6.1	Scalability . . . . .	44
6.2	Random Forest accuracy . . . . .	53
<b>7</b>	<b>Conclusions</b>	<b>54</b>
	<b>Bibliography</b>	<b>55</b>
<b>A</b>	<b>HDFS architecture</b>	<b>57</b>
<b>B</b>	<b>Matrix correlation example</b>	<b>59</b>
<b>C</b>	<b>Deploying Spark</b>	<b>60</b>
C.0.1	Download Spark . . . . .	60
C.0.2	Spark Shell . . . . .	62
C.0.3	Submitting applications . . . . .	62
C.1	Spark and flume integration guide . . . . .	65
C.2	Spark and HBase integration guide . . . . .	67
C.2.1	Procedure . . . . .	67
<b>D</b>	<b>Application code</b>	<b>70</b>



# Chapter 1

## Introduction

### 1.1 Big Data

#### 1.1.1 Definition

If there are currently a couple of words that are trending today are: Big data. Many people have heard about them but what is it? Well, in general terms, we might refer to as the trend in the advancement of technology that has opened the door to a new approach to understanding and decision-making. Then, Big Data, is used to describe large amounts of data (structured, unstructured, and semi-structured) that would take too long and are too expensive to load into a relational database for analysis. Therefore, the concept of Big Data is applied to any information that can not be processed or analyzed using traditional tools or processes.

#### 1.1.2 The four V's of Big Data

Nowadays are emerging in many important applications that require Big Data tools such as Internet search, business informatics, social networks, social media, genomics, streaming services, meteorology and a big etcetera which are facing this enormous amount of data, and it is a problem to deal with.

The question now is: when do we consider something as Big Data? Many companies like to break Big Data in four different *V*'s [1]:

1. *Volume*: Every day there are more devices and more complex information systems. This plus the fact that more and more people have access to them creates huge amount of data that is increasing exponentially. This would be our first big data problem.
2. *Velocity*: As mentioned before, the devices are becoming more complex. Monitoring functions in real time are increasing and we need to treat them. Handling large amounts of streaming data is really challenging and Big Data takes care of it.

3. *Variety*: Social networks, weareables, health, streaming video . . . We have an endless of different types of data and they all have to be able to deal with.
4. *Veracity*: This V does not seem so obvious but it is of great importance. Most business leaders do not trust the information they use to make decisions. Poor data quality cost US economy around \$3.1 trillion a year.

### 1.1.3 Big Data platforms

Although Big Data is a very new technology, it has a large variety tools to work with. The most relevant are the Hadoop environment tools.

Understanding this platform means understand the history of Big Data and of course *Spark* (the engine that is going to used in this thesis). So it is interesting to make a brief summary of its story and then see which elements compose it. [2]

#### 2002-2004: Hadoop pre-history

In 2002 Appears Nutch, a robot and search engine based on Lucene. His goal was web-scale and crawler-based search. It was not very successful because it was still far from offering web-scale goal.

#### 2004-2006: Hadoop gestation

GFS & MapReduce papers are published and are directly addressed to Nutch's scaling issues. Then they added to it DFS & MapReduce implementation. Even with these improvements it was still away from web-scale objective.

#### 2006-2008: Hadoop childhood

Finally in 2006 Yahoo hires Doug Cutting and Hadoop spins out of Nutch<sup>1</sup>. Web-scale finally achieved.

Figure 1.1 shows the evolution of the platform in a clearly way:

---

<sup>1</sup><http://nutch.apache.org/>

<sup>2</sup>[oracle4ryou.blogspot.be/2014/09/hadoop-history.html](http://oracle4ryou.blogspot.be/2014/09/hadoop-history.html)

## Hadoop History

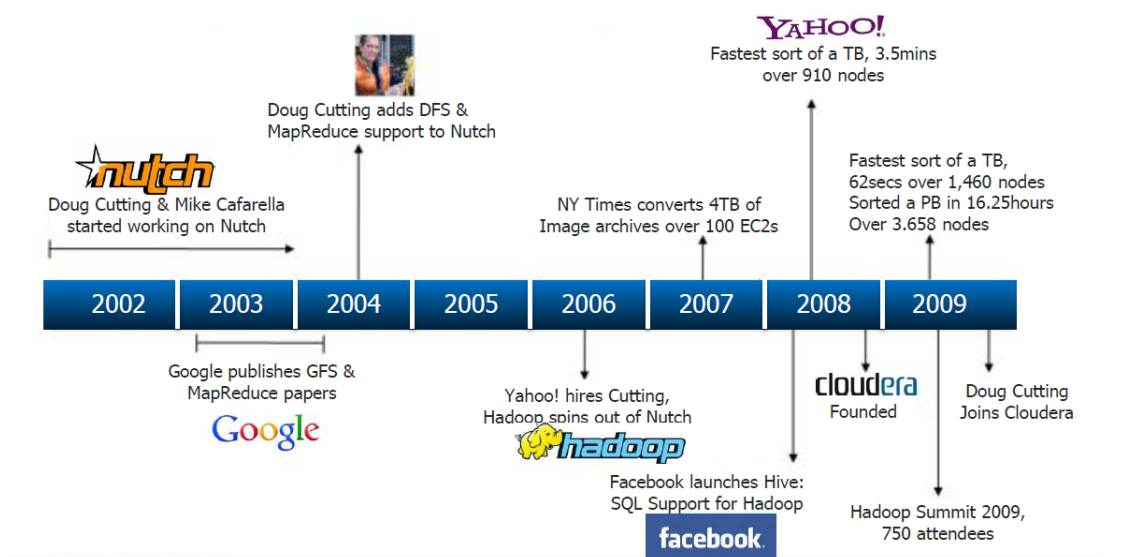


Figure 1.1: Hadoop timeline <sup>2</sup>

## Hadoop ecosystem

Apache Hadoop is a scalable fault-tolerant distributed system for data storage and processing (open source under the Apache license). It is composed of two main subsystems: Hadoop Distributed FileSystem (HDFS) and MapReduce. Basically Hadoop distributes our files through HDFS and then processes them through the MapReduce programming model.

Specifying. Hadoop by itself is a framework that allows other programs to interact over its system. So, in order to take advantage of Hadoop, we need the toolset that we see in Figure 1.2:

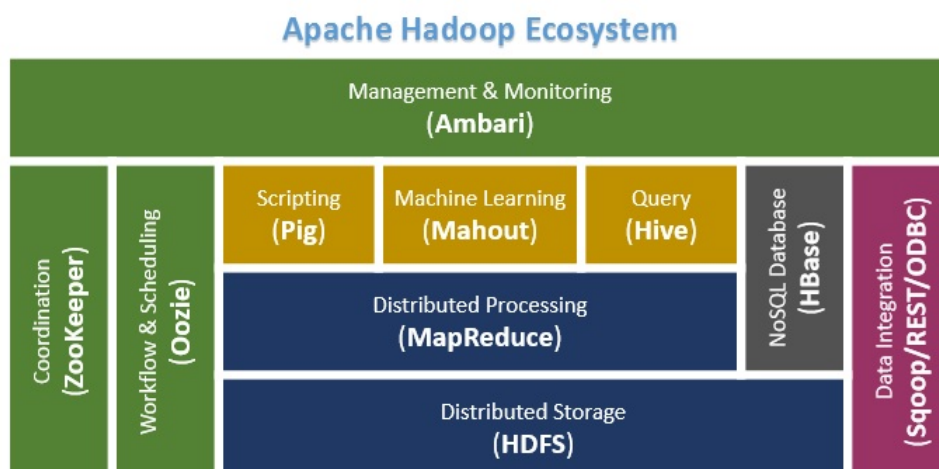


Figure 1.2: Hadoop ecosystem <sup>3</sup>

<sup>3</sup><http://www.mssqltips.com/sqlservertip/3262/big-data-basics-part-6-related-apache-projects-in->

## HDFS and MapReduce

The last step before starting to write about Apache Spark is understand with more detail the two main concepts of Hadoop: HDFS and MapReduce.

- *HDFS*: Is a distributed file system. It was created from the Google File System (GFS). HDFS is optimized for large files and flows of data. Its design reduces the Input/Output operations on the network. Scalability and availability are some of its key, because of data replication and fault tolerance. Important elements of the cluster are: [3]
  - *NameNode*: There is only one in the cluster. It regulates the access to the files from the clients. It keeps in memory the file system metadata and the block controls that each DataNode file has.
  - *DataNode*: They are responsible for reading and writing requests from clients. Files are composed of blocks, these are replicated in different nodes.

In the appendix A are included two images that show the writing process in HDFS (figure: A.1) and its replication system (figure: A.2)

- *MapReduce*: Is a batch process, created for distributed processing of data. Allows in a simple fashion, parallelize work on large volumes of data such as web logs.

MapReduce simplifies parallel processing, abstracting the complexity that distributed systems have. Map functions basically transform a dataset to a number of key/value pairs. Each of these elements will be ordered by key, and Reduce function is used to combine the values (with the same key) in the same result.

A program in MapReduce is generally known as Job. Running a Job begins when the client sends the configuration to JobTracker, this setting specifies the Map, Combine (shuffle) and Reduce functions. [4]

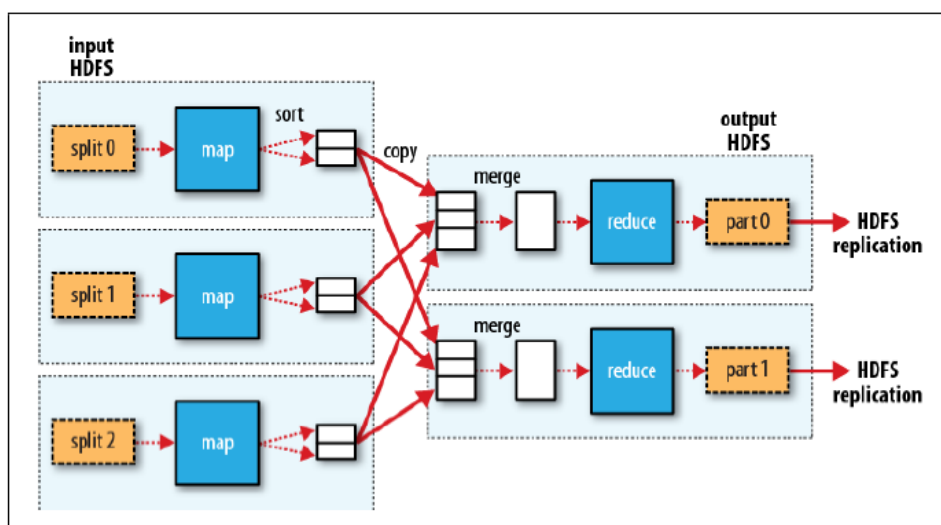


Figure 1.3: MapReduce schema <sup>4</sup>

hadoop-ecosystem/

<sup>4</sup><http://hao-deng.blogspot.be/2013/05/mapreduce-data-flow-with-reduce-tasks.html>

## 1.2 Machine Learning

### 1.2.1 Definition

Machine learning is a branch of Computer Science which, essentially, studies how systems learn tasks. Another way to think about machine learning is that it is *pattern recognition*: the act of teaching a program to recognize patterns and react to them.

Is not surprising that nowadays, with the amount of data and the computing capacity we have, the use of machine learning algorithms is increasing vastly. Here are some examples of its currently uses:

- Fraud detection: identify credit card transactions which may be fraudulent in nature.
- Weather prediction.
- Face detection: find faces in images.
- Spam filtering: identify email messages as spam or non-spam.
- Customer segmentation: predict, for instance, which customers will respond to a particular promotion.
- Robotics.
- Medical diagnosis: diagnose a patient as a sufferer or non-sufferer of some disease.

### 1.2.2 Types of machine learning algorithms

In order to teach our system, we have three different methods: *supervised learning*, *unsupervised learning* and *reinforcement learning*. [5]

- *Supervised learning*: this method has a database that we use to train our system.
- *Unsupervised learning*: discovers a good internal representation of the input. Unlike the supervised method, this one takes decisions without a training data base.
- *Reinforcement learning*: Input data is provided as stimulus to a model from an environment to which the model must respond and react. Feedback is provided not from of a teaching process as in supervised learning, but as punishments and rewards in the environment.

Finally, in order to show the most relevant algorithms of each learning method there is a table below:

<b>Supervised</b>	KNN
	Linear regression
	Logistic regression
	Naive Bayes
	Decision trees
	Random Forests
	Neural networks
	SVM
<b>Unsupervised</b>	K-means
	Hierarchical clustering
	Fuzzy clustering
	Gaussian mixture models
	Self-organizing maps
<b>Reinforcement</b>	Temporal Differences
	Sarsa
	Q-learning

Table 1.1: Table with most relevant algorithms of each kind of learning methods

### 1.3 Thesis work description

This thesis has two main objectives. In one hand, learn and understand a considerable amount of tools related with Big Data, a subject which I were not familiar with. On the other hand, as a student who loves machine learning, try to combine these two disciplines by making an application.

For the accomplishment of my objectives, I have used Apache Spark, a new born platform which pretends to replace the old one: Hadoop MapReduce. Therefore, a big part of my work has been to understand the capabilities and limits of this tool. Moreover acquire the criteria to develop an environment able to interconnect several programs.

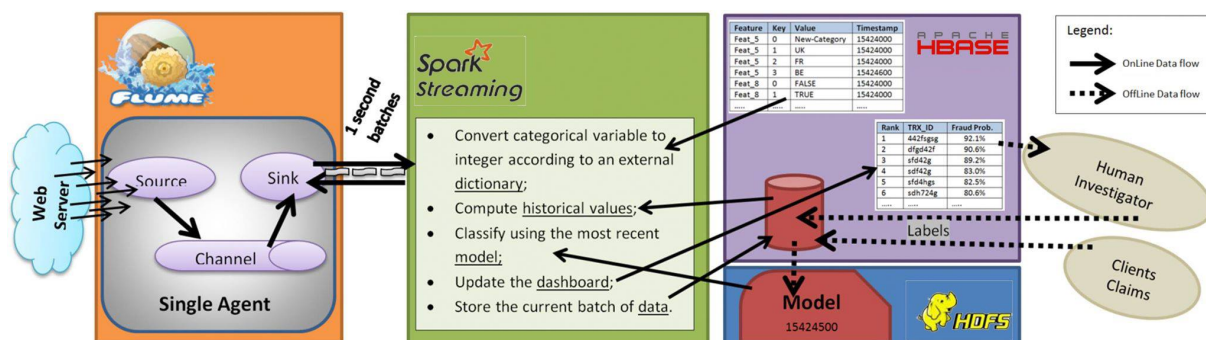


Figure 1.4: Thesis schema.

Thus, I have focused on the creation of a system able to analyse bank transactions on-line with the objective to discover fraudulence.



Finally, the fact that Spark is a new creation engine is a really positive point because it has a long road ahead, and although is still young it has raised a lot of expectation. Besides, this fact implies that I will be one of the few developers working on it (Spark can not be compared to other platforms like Hadoop in amount of developers). However, the fact of being a new born platform was a handicap too for two reasons:

- There are not many documentation compared to others platforms.
- I had to be continually alert with the new updates. As consequence of this I had to change some lines of code and even the orientation of my project in one occasion.

## 1.4 Thesis structure

To structure this thesis as best possible we decided that must proceed with the following fashion: In Chapter 2 we will introduce the state of the art of technologies: MapReduce paradigm, Apache Hadoop, Apache Spark, Spark libraries, programs that interact with our application like Apache Flume and HBase and Scala, the program language used in this thesis. Chapter 3 introduce a brief but necessary introduction to random forest, the machine learning algorithm used in this thesis. In Chapter 4 we describe our application in depth ; in Chapter 5 we explain where are we deploying our application and how the university cluster works. We will continue discussing the results of our system in Chapter 6. And finally, we provide our conclusions and future works in Chapter 6.

# Chapter 2

## State of the art

This chapter describes the technology used during this thesis. We will emphasize on Spark, the main core of this thesis, and will end with Random Forest algorithm explanation.

### 2.1 Apache Spark

“ Learning Spark is at the top of my list for anyone needing a gentle guide to the most popular framework for building big data applications.”  
—Ben Lorica Chief Data Scientist, O’Reilly Media

#### 2.1.1 History of Spark

Spark is an open source project made and maintained by a community of developers. Spark begins in 2009 in the UC Berkeley RAD Lab but finally will become part of AM-PLab. Spark born due the observed inefficiency of MapReduce in iterative and interactive computing jobs. That is why Spark, even from the beginning, was designed to be fast for the mentioned algorithms. Also brought ideas like support for in-memory storage and efficient fault recovery. As a result, once Spark was released was already 10–20 times faster than MapReduce for certain jobs.

At the very begining the first Spark’s users were from groups inside UC Berkeley. But in a very short time, over 50 organizations began using Spark, and today, half of them speak about their use cases at Spark community events such as Spark Meetups and the Spark Summit. Currently the most important contributors to Spark include UC Berkeley, Databricks, Yahoo!, and Intel. [6]

#### 2.1.2 Introduction to Spark

Apache Spark is a cluster computing framework built in Scala. It is a fast and general engine for large-scale parallel data processing.

Spark extends the popular MapReduce model. We can say that one of the most

important characteristic of Spark is that works *in memory*. This fact implies that is more efficient doing computations like iterative algorithms, interactive queries and stream processing due the avoidance of the disk read/write bottleneck.

The Spark project contains multiple integrated components. Spark SQL, Spark Streaming, MLlib and GraphX. These components have been designed to work together whenever you want. Thus, they can be combined like libraries in a software project. Where Spark is their core, the one that is responsible for scheduling, distributing, and monitoring applications over cluster. [6]

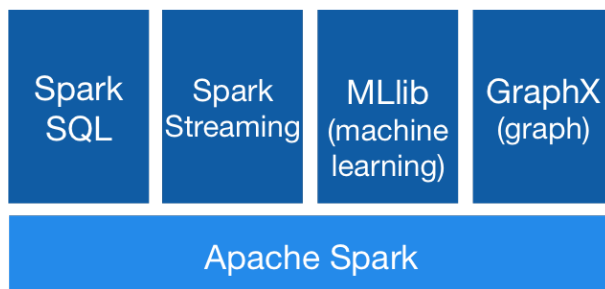


Figure 2.1: Spark stack graph. <sup>1</sup>

Finally before going deeply into Spark, it is worth to mention that Spark is also very accessible. It offers a few interesting APIs in Python, Java, Scala, SQL, and built-in libraries. It also can run in Hadoop clusters, which means that can interoperate with other Big Data tools like the ones showed on Figure 1.1.

### 2.1.3 Basic concepts

#### Key words

In order to understand the functioning of Spark is needed to feel comfortable with the following words: [7]

- *Job*: It is a parallel computation which reads some input from HDFS, HBase<sup>2</sup>, Cassandra<sup>3</sup>, local... and performs some computation on the data (e.g save, collect).
- *Tasks*: Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- *Executor*: The process responsible for executing a task.

The number of executors used in our programs are directly related with the amount of time that one job takes to be executed. The figure showed below shows the relevance of using a correct number of executors and how reduces exponentially the execution time of our jobs. Although we can also appreciate that there is a threshold where increasing the number of executors does not reduce the execution time any more.

<sup>1</sup>Extracted from the book: Learning Spark, Lightning-fast data analysis

<sup>2</sup><https://hbase.apache.org/>

<sup>3</sup><http://cassandra.apache.org/>

<sup>5</sup><http://spark.apache.org/docs/1.3.0/cluster-overview.html>

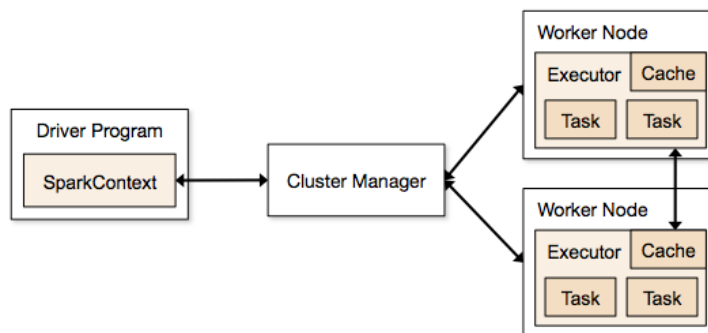
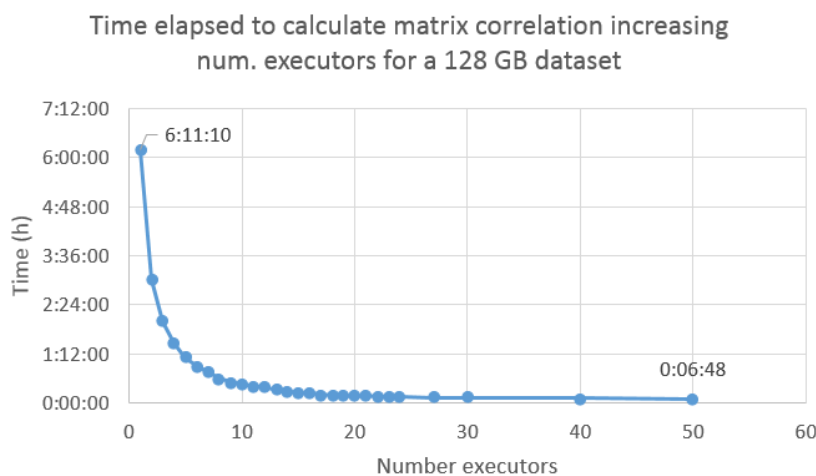


Figure 2.2: In this schema we show how a driver program send task for the executors to run <sup>5</sup>



- *Driver*: The program/process responsible for running the Job over the Spark Engine.
- *Master*: The machine on which the Driver program runs.
- *Slave*: The machine on which the Executor program runs.
- *Stages*: Jobs are divided into stages. Stages are classified as a Map or reduce stages. They are divided based on computational boundaries, all computations(operators) cannot be Updated in a single Stage. It happens over many stages.

### Spark DAG (directed acyclic graph)

DAG execution model is essentially a generalization of the MapReduce model. Whereas MapReduce model has two kind of computation steps (a map step and a reduce step), DAGs created by Spark can contain any number of stages. This allows some jobs to complete faster than they would in MapReduce. For example if we submit a Spark job like in the second stage of the Figure: 2.3 which contains a map operation followed by a filter operation. Spark DAG optimizer would rearrange the order of these operators and schedule them. This means that, in situations where MapReduce must write out intermediate results to the distributed filesystem, Spark can pass them directly to the next step in the pipeline.[?]

<sup>7</sup><http://blog.cloudera.com/blog/2014/03/apache-spark-a-delight-for-developers/>

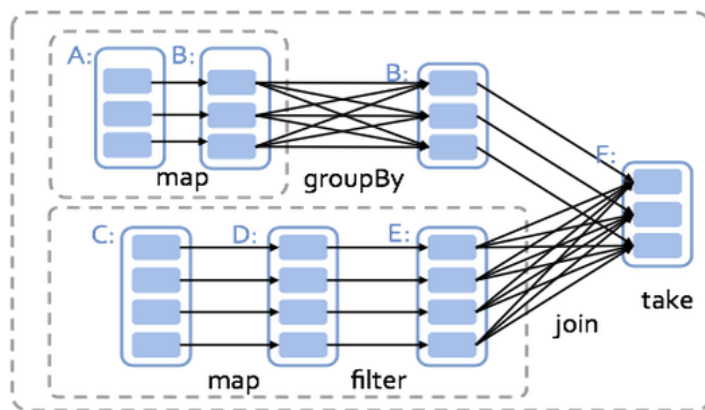


Figure 2.3: Example of how Spark computes job stages. Boxes with solid outlines are RDDs (Section 2.1.3). Partitions are shaded rectangles. The first stage is going to be executed first and will perform a map action. The second stage applies a map and filter function. Finally the results of both stages are going to be joined in the third stage. <sup>7</sup>

## RDD (resilient distributed dataset)

This is the core concept in Spark. Formally, an RDD is a read-only, partitioned collection of objects which can be stored either in memory or in disk. RDDs can only be created in two ways: by loading an external dataset, or by distributing a collection of objects using the command *parallelize*.

The main characteristics of using RDDs are [7]:

- It is *fault tolerant* due it uses a coarse-grained transformations. This kind of transformation means that all our transformation will be applied into all the dataset but not individual elements. With this functionality we lose flexibility, but in the other hand makes possible to save the operations done in our dataset in the DAG. As a result if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition. Therefore, lost data can be recovered without requiring costly replication.
- It uses *lazy evaluation*. *Lazy evaluation* means that when we call a transformation on an RDD, the operation is not immediately performed. Instead, Spark internally records metadata to indicate that this operation has been requested. This property is used to reduce the number of passes that Spark has to take over our data by grouping operations together.
- As we said before RDD is a dataset which is partitioned, that is, it is divided into *partitions*. Each partition can be present in the memory or disk of different machines. So if we process an RDD, then Spark will need to launch one task per partition of the RDD.
- The RDDs accept two types of operations: transformations and actions. The Figure 2.4 below shows and example of action and transformation.

<sup>9</sup><http://es.slideshare.net/tsliwowicz/reversim2014>

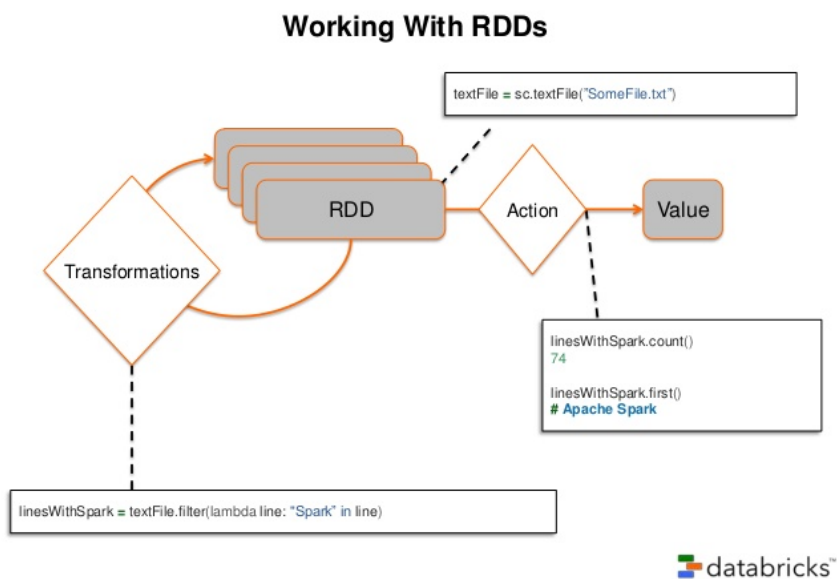


Figure 2.4: Here we generate our first RDD (`textFile`) loading a text file. Once we have generated this RDD we apply a *filter transformation* which will return us a new RDD (`linesWithSpark`) with the dataset filtered by the lines that contain *Spark*. Finally we perform two actions to `linesWithSpark` RDD that will return us the desired values. <sup>9</sup>

### 2.1.4 Scalability

This subsection is going to be quite short, that is because is meant to show how well Spark scales in a real example. In this case we are going to calculate the correlation matrix of large datasets.

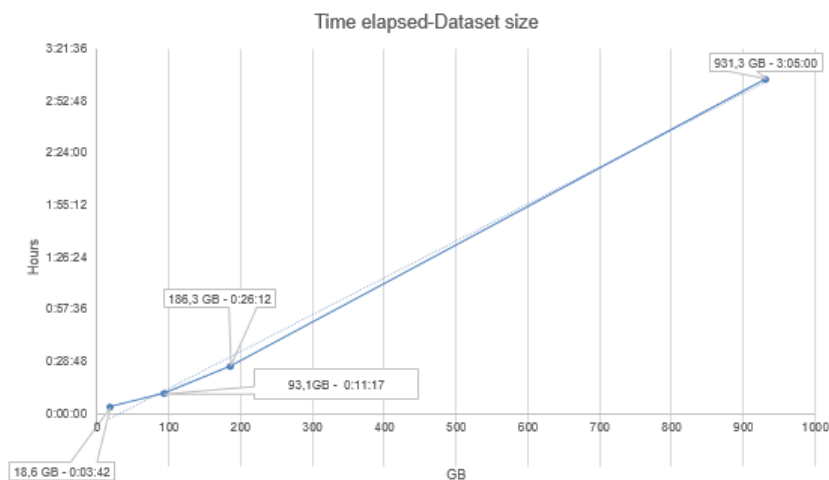


Figure 2.5: Chart with the time elapsed calculating a matrix correlation with datasets of different sizes.

### 2.1.5 First example

Once explained the key concepts of Spark, we are prepared to see and to explain our first example. In this case we are going to study probably the most representative example: Word Count.

```
1 val textFile = spark.textFile("hdfs://input.txt")
2 val counts = textFile.flatMap(line => line.split(" "))
3                   .map(word => (word, 1))
4                   .reduceByKey(_ + _)
5 counts.saveAsTextFile("hdfs://result.txt")
```

In the first line of code we create an RDD called *textFile*. This is the result of loading a distributed file in HDFS called: *input.txt*. Once we have created this RDD, in the second line, we can apply several operations. In this case we are using a *flatMap*, which receive the lines of *textFile*. Then *flatMap* applies a *lambda function* to each line of the RDD.

**NOTE:** Lambda functions are used inside some methods like *flatMap*. Basically writing:

```
1 line => line.split(" ")
```

Is equivalent to do the next function:

```
1 def function (String lines): RDD[String] = {
2   return lines.split(" ")
3 }
```

The function applied is *split*, which will create a new RDD with just the words. Then to this new RDD, in the third line, we will apply a *map* function. This *map* will receive the words from the new RDD created, and will create a new RDD with a pair of key values with the word and a number 1. Then in the fourth line we will create the last RDD, which will be the reduce of the last RDD by its key. Basically the "*\_ + \_*" takes the first value of the key/value pairs of the last RDD and will make a sum with itself. Finally we save this last RDD in the variable *counts* in order to finally save the result in a text file.

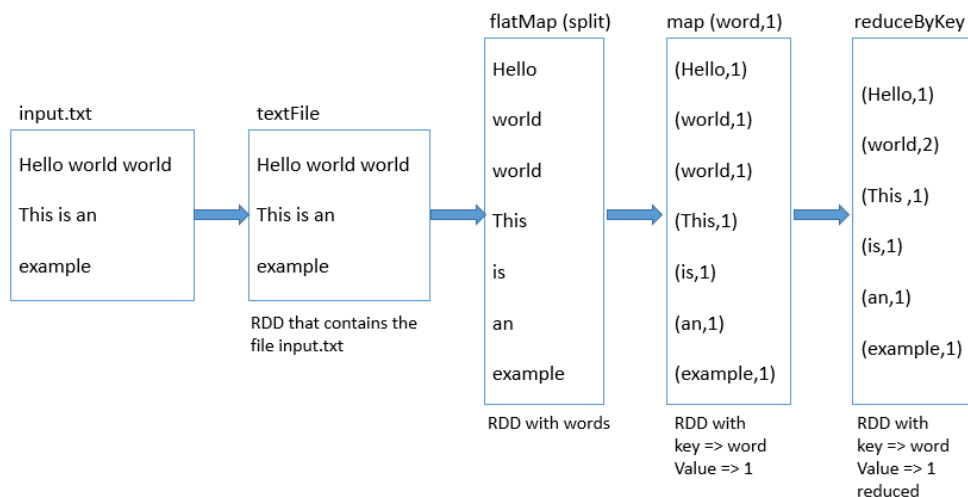


Figure 2.6: Flux diagram of the above explication.

### 2.1.6 Machine Learning Library (MLLIB)

One of the most important tools in Spark is its machine learning library. MLlib is the Spark's library that contains machine learning algorithms meant to work in a cluster. This library is part of the Spark core, therefore can be used from any other of the other libraries. MLlib's design and philosophy are simple: it lets you invoke various algorithms on distributed datasets, representing all data as RDDs. MLlib introduces a few data types (e.g., labeled points and vectors), but at the end of the day, it is simply a set of functions to call on RDDs. [8]

The algorithm that are currently available are:

Basic statistics	summary statistics
	correlations
	stratified sampling
	hypothesis testing
	random data generation
Classification and regression	linear models (SVMs, logistic regression, linear regression)
	naive Bayes
	decision trees
	ensembles of trees (Random Forests and Gradient-Boosted Trees)
	isotonic regression
Collaborative filtering	alternating least squares (ALS)
Clustering	k-means
	Gaussian mixture
	power iteration clustering (PIC)
	latent Dirichlet allocation (LDA)
	streaming k-means
Dimensionality reduction	singular value decomposition (SVD)
	principal component analysis (PCA)
Frequent pattern mining	FP-growth
Optimization (developer)	stochastic gradient descent
	limited-memory BFGS (L-BFGS)



### 2.1.7 Spark Streaming

The streaming processor of this thesis will be Spark Streaming. Basically, Spark Streaming is a library of Spark that deals with streaming data. It enables scalable, high-throughput, fault-tolerant stream processing of live data. [8] [7]



Figure 2.7: Spark streaming diagram. There are many ways to insert data to Spark streaming, like Kafka<sup>13</sup>, Flume<sup>14</sup>, HDFS, etc. Once this data is processed can be saved in HDFS, databases and dashboards. <sup>15</sup>

The main advantages of using Spark streaming are: [9]

- It is scalable.
- Achieve second-scale latencies.
- Is integrated with batch and interactive processing.
- It has a simple programming model.
- Its is efficient fault-tolerance. That is possible because it works with batches that are replicated over the cluster.

Internally, Spark Streaming receives data streams and divides this data into batches. After the division the data is processed by the core of Spark which will generate the final stream of results in batches.



Figure 2.8: Image that illustrates the above description. <sup>16</sup>

In order to deal with streaming processing there are a few different general techniques. Two of the most common ones the following:

<sup>13</sup><http://kafka.apache.org/>

<sup>14</sup><https://flume.apache.org/>

<sup>15</sup>Source: <https://spark.apache.org/docs/1.3.0/streaming-programming-guide.html#linking>

<sup>16</sup>Source: <https://spark.apache.org/docs/1.3.0/streaming-programming-guide.html#linking>

- Treat each record individually and process it on the fly.
- Combine multiple records into mini-batches. These mini-batches can be delineated either by time or by the number of records in a batch.

The technique used by Spark Streaming is the second one. The main concept that represent this approach is the DStream. A DStream is a sequence of mini-batches, where each mini-batch is represented as a Spark RDD.

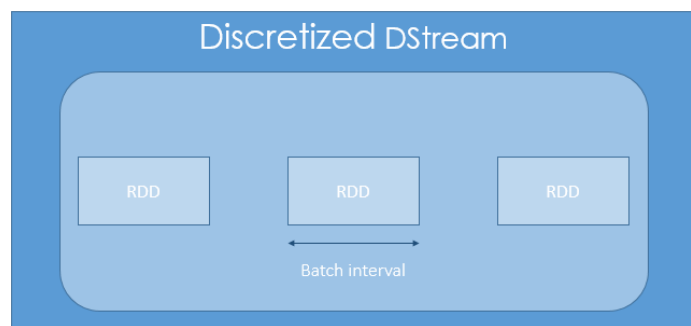


Figure 2.9: This image represent de concept of DStream.

To create a DStream we need to define the time of its mini-batches, called *batch interval*. During this interval the DStream input data will become into RDDs, like in the above figure. Each RDD in the stream will contain the records that are received by the Spark Streaming application during a given batch interval. If there is not data in a given interval, the RDD will be empty. [8]

## Transformations in Spark Streaming

The fact that DStreams contain RDDs, make possible that Spark Streaming have a set of transformations available on DStreams. These transformations are similar to those available on the typical RDDs of Spark. The figure 2.10 shows all the types of operations available with DStreams:

### Windows operations

As table 2.10 shows, a new kind of operations appears when we are working with DStreams and it is worth to understand, the windows operations.

- *Window duration*: The window duration is responsive of how much time Spark streaming is getting data.
- *Sliding window*: This one opens a window for an indicate amount of time, where the DStreams are going to accumulate until the window is closed. Once closed , Spark streaming is going to process all the Dstreams at the same time.




 Transformations	 Windows operations	 Output operations
<ul style="list-style-type: none"> <li>map(func), flatMap(func), filter(func), count()</li> <li>repartition(numPartitions)</li> <li>union(otherStream)</li> <li>reduce(func), countByValue (), reduceByKey(func, [numTasks])</li> <li>join(otherStream, [numTasks])</li> <li>transform(func)</li> <li>updateStateByKey(func)</li> </ul>	<ul style="list-style-type: none"> <li>window(windowLength, slideInterval)</li> <li>countByWindow(windowLength, slideInterval)</li> <li>reduceByWindow(func, windowLength, slideInterval)</li> <li>reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])</li> <li>countByValueAndWindow(windowLength, slideInterval, [numTasks])</li> </ul>	<ul style="list-style-type: none"> <li>print()</li> <li>foreachRDD(func)</li> <li>saveAsObjectFiles(prefix, [suffix])</li> <li>saveAsTextFiles(prefix, [suffix])</li> <li>saveAsHadoopFiles(prefix, [suffix])</li> </ul>

Figure 2.10: Table shows all the transformations and operations available in a DStream

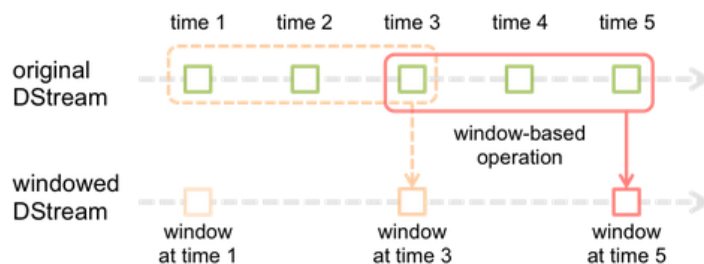


Figure 2.11:

## 2.2 Data integration

This section aims to explain how the data is received in our system. Two of the most common data integrators are Apache Flume and Apache Kafka. For the thesis we have used Flume due to its ease of use compared to Kafka. Although Kafka is better for systems with more than one stream. Let's explain then each one.

### 2.2.1 Apache Flume

Apache Flume is a reliable and distributed system for efficiently gathering, collecting and moving large amounts of log data from different sources to a centralized data store. [10]

Besides, Apache Flume can move massive quantities of event data too, so it is not only limited to log data aggregation. Then, Apache Flume can be useful in network traffic data, social-media-generated data, email messages and most likely, any data source possible. [11]

## Data flow model

Two important concepts have to be understood about the data flow model: Flume event and Flume Agent.

1. *Flume event*: Is the unit of data flow that have a byte payload and an optional set of string attributes.
2. *Flume agent*: Is a (JVM) process that hosts the components through which events flow from an external source to the next destination (Hop)

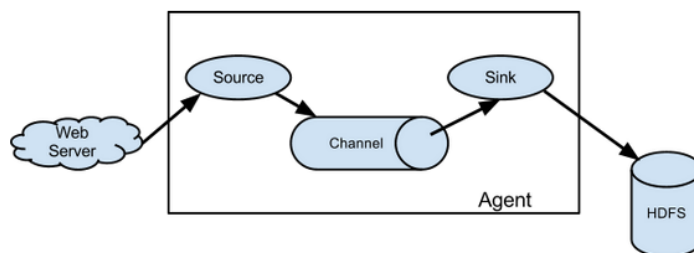


Figure 2.12: This image represents the basic idea of Flume Agent.<sup>17</sup>

Following the thread, if an external source like a web server is delivered to a Flume source it will be consumed. If we deep in the explanation, we will see that all of this happens because the external source sends events to Flume in a format that is recognized by the target Flume Source. At that point, when a Flume source receives an event, it stores it into one or more channels and this event will remain in the channels until a Flume sink consumes it.

How does Flume sink treat that information? It removes the event from the channel and puts it into an external storehouse like HDFS or forwards it to the Flume source of the next Flume agent (next hop) in the flow.

## Reliability

The fact events are staged in a channel on each agent and that they are only removed from the channel after they are stored in the channel of next agent or in the terminal repository provides Flume end-to-end reliability of the flow.

Furthermore, Flume, to guarantee the reliable delivery of the events, uses a transactional approach. In other words, Flume sources and Flume sinks encapsulate in a transaction the storage/retrieval, respectively, of the events placed in or provided by a transaction provided by the channel. This ensures that the set of events are reliably passed from point to point in the flow. Until now, we have focused in single-hop delivery but it is important to explain the reliability in multi-hop flow too. In multi-hop cases, the sink of from the previous hop and the source from the next hop *both have their transactions running* to guarantee that the data is safely stored in the channel of the next hop.

---

<sup>17</sup><https://flume.apache.org/>

## Recoverability

Recovery from failure is possible thanks to the storage of events in the channel. Flume supports a durable file channel, which is backed by the local file system. There is also a memory channel, which simply stores the events in an in-memory queue, which is faster, but any events still left in the memory channel when an agent process dies can not be recovered.

## Spark streaming and flume integration

Here we explain how to configure Flume and Spark Streaming to receive data from Flume. There are two approaches in order to achieve this.

- **Flume style push-based approach:** In this approach, Flume push data directly to Spark Streaming who sets up a receiver that acts an Avro agent for Flume.
- **Pull-based Approach using a Custom Sink:** This method runs a custom Flume sink that allows the following. In conclusion, Flume does no push directly Spark Streaming. It takes the following steps: [12]
  - Flume pushes data into the sink, and the data stays buffered.
  - Spark Streaming uses a reliable Flume receiver and transactions to pull data from the sink. Transactions succeed only after data is received and replicated by Spark Streaming.

This ensures stronger reliability and fault-tolerance guarantees than the previous approach. However, this requires configuring Flume to run a custom sink.

In this thesis we have chosen the second approach due it is the newest one. This new way to interconnect Flume with Spark has been thought to solve the deficiencies of the first one, that is why it offers best characteristics.

### 2.2.2 Apache Kafka

Despite that this thesis is going to integrate Flume and Spark, the integration between Kafka and Spark is probably the most used. In fact, Apache Kafka was our first option. But the ease of use of Flume and the fact that we are using just one data source, tip the scale in favor of Flume. That is why we think that Kafka is still worth to study deeply.

#### Introduction to Kafka

Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system. As mentioned, this system is excellent when we have different data sources. The question which we are asking ourselves is: what does Kafka offers us that makes it so popular?

To answer this question we need to understand the problems that other platforms have. In the figure below we see on a system with many sources have a problem of excess pipelines. This is really impractical and makes the system complex and inefficient.

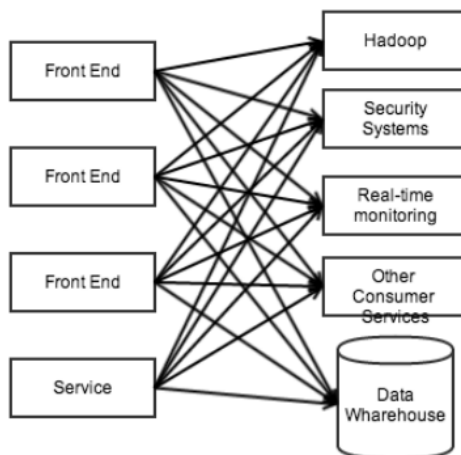


Figure 2.13: Image that shows the problem of multiple pipelines in Kafka. <sup>18</sup>

To overcome these complications, Kafka, will use a system of producer / consumer messaging. The Figure 2.14 shows how the complexity of our system has been reduced drastically.

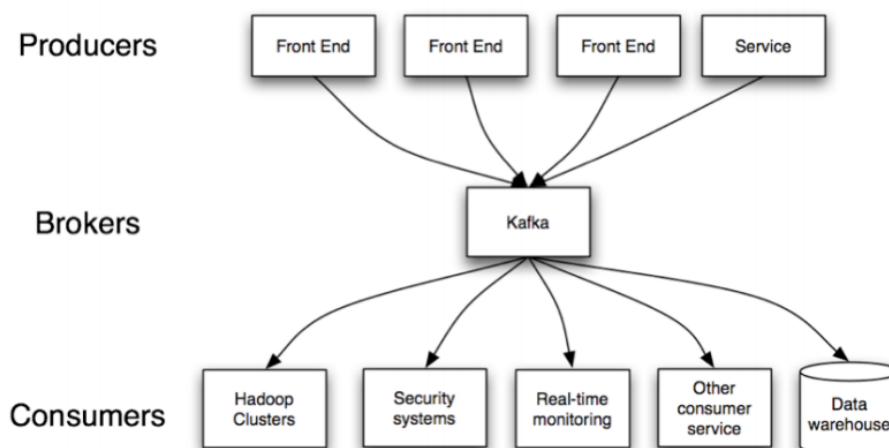


Figure 2.14: Diagram that shows how Kafka decouple the data-pipelines and simplify our system. <sup>20</sup>

After this brief introduction we need to understand the basics of this messaging system correctly:

- The first concept and the core of this system is the topic. Topics are categories where Kafka maintains feeds of messages. Later on we will explain with more detail this component.
- Of course with a producer / consumer system, the processes that publish messages to a Kafka topic are producers.

<sup>18</sup>[http://es.slideshare.net/charmalloc/developingwithapachekafka-29910685?from\\_action=save](http://es.slideshare.net/charmalloc/developingwithapachekafka-29910685?from_action=save)

<sup>20</sup>[http://es.slideshare.net/charmalloc/developingwithapachekafka-29910685?from\\_action=save](http://es.slideshare.net/charmalloc/developingwithapachekafka-29910685?from_action=save)

- Then, the processes that are subscribed to topics and take the published messages are the consumers.
- Finally, the Broker is one of the servers that comprise a Kafka cluster.

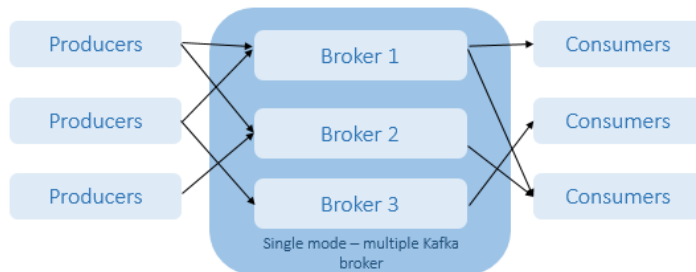


Figure 2.15: This diagram represents a simple system of producer / consumer in a Kafka cluster. <sup>22</sup>

Next, the most important parts in Kafka will be explained in detail.

## Topics

A topic is a category or feed name to which messages are published. We can think on it like a Twitter Hashtag. Imagine a topic called *test*, this topic is like a Hashtag *#test* where there are subscribers that are subscribed to this topic and will receive all the messages published by this topic. For each topic, Kafka maintains a partitioned log that looks like this:

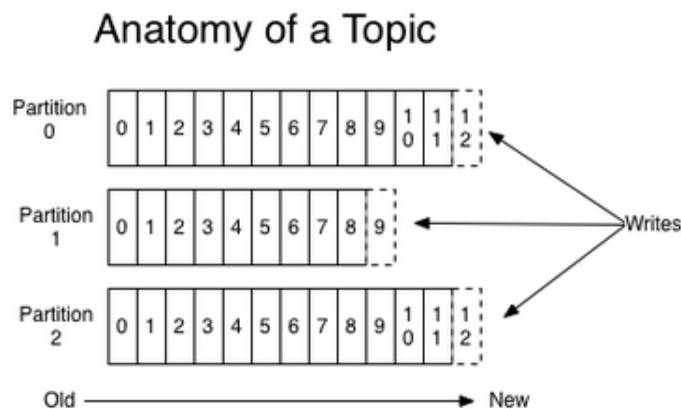


Figure 2.16: This figure shows how a topic is partitioned. <sup>23</sup>

As we see in this picture, each partition is like an ordered, immutable sequence of messages that is continually appended to a commit log. The messages in the partitions are each assigned a sequential id number called the offset that uniquely identifies each message within the partition. All these messages are retained inside a Kafka cluster for a configurable period of time even if they are not consumed.

<sup>22</sup><https://kafka.apache.org/>

<sup>23</sup><https://kafka.apache.org/>

Another concept appears when we work with this kind of partition model: the *offset*. The offset is the only metadata saved for the consumer and it allows it to find the position of the consumer in the log. The offset is controlled by the consumer. Normally, is that consumer who will advance its offset as it reads messages, but in Kafka the position is controlled by the consumer and it can consume messages in any order it likes.

We can summarize the partition purposes in two:

1. They allow the log to scale beyond a size that will fit on a single server.
2. They act as the unit of parallelism.

## Producers

The second key element which we are going to explain about Kafka are the Producers. The producer is responsible of publish data to the topics whom they are assigned. Besides, the producer is also responsible for choosing which message to assign to which partition within the topic.

## Consumers

Messaging traditionally has two models: queuing and publish-subscribe. In a queue, a pool of consumers may read from a server and each message goes to one of them; in publish-subscribe the message is broadcast to all consumers. Kafka offers a single consumer abstraction that generalizes both of these—the consumer group.

Consumers label themselves with a consumer group name, and each message published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines.

If all the consumer instances have the same consumer group, then this works just like a traditional queue balancing load over the consumers.

## 2.3 HBase

This thesis is going to deal with the fact of storing large amounts of streaming data in HDFS. To face this situation we have chosen a NoSQL database called HBase. In this section we are going to describe briefly how this database works and which are the most relevant concepts used in this thesis.

HBase is the open source implementation of Google's Big Table [16]. That implementation basically uses a sparse, distributed, persistent multidimensional sorted *map* or *dictionary*. This one is indexed by its row key, column key, and a timestamp. HBase's data model can be described easily in the form of tables, consisting of rows and columns (like in relational databases). But that is where the similarity between RDBMS data models and HBase ends. In fact, even the concepts of rows and columns are a little bit different [17]. Let's start then with the most relevant concepts:



- *Table*: HBase organizes data into tables. Table names are Strings and composed of characters that are safe for use in a file system path.
- *Row*: Within a table, data is stored according to its row. Rows are identified uniquely by their row key. Row keys do not have a data type and are always treated as a byte[ ] (byte array).
- *Column family*: Data within a row is grouped by column family. Column families also impact the physical arrangement of data stored in HBase. For this reason, they must be defined up front and are not easily modified. Every row in a table has the same column families, although a row need not store data in all its families. Column families are Strings and composed of characters that are safe for use in a file system path.
- *Column Qualifier*: Data within a column family is addressed via its column qualifier, or simply, column. Column qualifiers need not be specified in advance. Column qualifiers need not be consistent between rows. Like row keys, column qualifiers do not have a data type and are always treated as a byte[ ].
- *Cell*: Is the intersection between a row key, column family, and column qualifier that uniquely identifies a cell. The values within a cell do not have a data type and are always treated as a byte[ ].
- *Timestamp*: Values within a cell are versioned. This versioning is done with a timestamp, which is the identifier for a given version of a value. It is important to keep in mind that when a timestamp is not specified during a writing process, the timestamp that will be used is the current one.

The image shows a screenshot of HBase data with two rows. The first row has a row key '37222-1-100152443-T-C'. The second row has a row key '37222-1-100174678-C-T'. Both rows have column families: PROL, ST, ID, LQ, MTS, QU, MTP, and SB. The QU column family has a column qualifier 'QU' with a timestamp of 23/4/2015 20:14:09. The values in the cells are: Blood, 16488, 76030578.95, 0.998, 10924266.63, and POLYNORPHISM\_AUTOMA TIC.

Row Key	Column Family	Column Qualifier	Cell	Timestamp
37222-1-100152443-T-C	PROL	ST	Blood	
	ID		16488	
	LQ			
	MTS			
	QU	QU	76030578.95	23/4/2015 20:14:09
	MTP			
	SB			
37222-1-100174678-C-T	PROL	ST	Blood	
	ID		49669	
	LQ			
	MTS		0.998	
	QU		10924266.63	
	MTP		POLYNORPHISM_AUTOMA TIC	
	SB			

Figure 2.17: This image shows all the main parameters of a HTable. It has been taken from Hue <sup>25</sup>.

## 2.4 Scala

To finish this chapter we will talk about the main programming language used: *Scala*.

<sup>25</sup>[https://en.wikipedia.org/wiki/Hue\\_\(Hadoop\)](https://en.wikipedia.org/wiki/Hue_(Hadoop))

### 2.4.1 Why Scala API?

Spark has currently three main types of programming languages APIs: Scala, Python and Java. The points that tip the scale to choose Scala instead of the others programming APIs are the following:

- *Ease of use*: Here Scala and Python are pretty even. The loser of this part is Java.
- *Performance*: If we are working with simple systems is not a problem, but if we are working on complex ones Scala is x10 faster than Python because it works on JVM.
- *Documentation*: All the documentation of Spark is made in Scala, whereas Python and Java are not used all the examples.
- *Libraries*: Even though Python has better machine learning than Scala, they are not Big Data oriented. MLlib, which has machine learning algorithms Big Data orientes, has all its algorithms available for Spark and not in the others APIs.

In the figure 2.4.1 we have done a comparative table to see more visually the above points.

	Scala	Python	Java
Ease of use	✓	✓	✗
Performance	✓	✗	✓
Spark Documentation	✓	✗	✗
Greater amount of Spark libraries	✓	✗	✗
Own libraries	✗	✓	✓

Figure 2.18: This a comparison table between Spark APIs.

## 2.4.2 History of Scala

Now that we are decided which is going to be our programming language let's make a brief introduction. First we are going to start with its history. [18] [19]

### 2001

Scala was designed by Martin Odersky and his group at the Federal Polytechnic School of Lausanne (Switzerland). Odersky it aimed to combine functional programming and object-oriented programming.

### 2003

This year was launched the first public version.

### 2006

A second version was released: Scala v2.0.

### 2011

On 12 May of this year, Odersky and his collaborators launched Typesafe Inc., a company to provide commercial support, training, and services for Scala. This company received a \$3 million investment from Greylock Partners this year.

## 2.4.3 What does Scala offer us?

The main advantages of using Scala can be summarized like this:

- It is a programming language designed for general purpose to express common programming patterns in a concise and elegant fashion.
- The characteristics of object-oriented and functional languages are integrated.
- Scala is not an extension of Java, but it is completely interoperable with it.
- Scala is translated to Java bytecode and program efficiency is usually compiled as Java.

## 2.4.4 First look at Scala code

This subsection aims to show an easy example to see our first lines in Scala. The code used has been extracted from Programming Scala book. [20]

```
1 class Upper {
2   def upper(strings: String*): Seq[String] = {
3     strings.map((s:String) => s.toUpperCase())
4   }
5 }
6 val up = new Upper
7 Console.println(up.upper("A", "First", "Scala", "Program"))
```

The class of above converts Strings to uppercase. So, in the first line we are defining our class and giving to it the name of *Upper*. After that we define the method *upper* where we are going to introduce several strings (the operator *\** is the responsive of permit it). After the *:* we indicate to this method which kind of parameter we are going to receive, in this case a sequence of strings. Finally the last line of this method uses the method *map* into the *strings* variables. Inside the *map* we are doing the same as the code provided in the subsection of Spark "*First example*".

Once we have created the method *upper* we are going to create an object in line 6. Finally in the next line we will use the object *up* which will contain the method *upper* in order to finally print in console the uppercase of the strings introduced inside the method.

# Chapter 3

## Random Forest

### 3.1 Introduction

In a supervised classification problem a sample  $\mathcal{S}$  is defined as:

$$\mathcal{S} = \{(x_n, y_n) | n = 1, 2, \dots, N, x_n \in \mathbb{R}^d, y_n \in \{1, 2, \dots, C\}\} \quad (3.1)$$

being  $N$  the number of elements of the set of data,  $C$  is the number of different classes and  $d$  the number of variables that define the samples  $x_n$  of the set of data. Each of these examples is represented by a vector  $x_n$ , which is associated with a corresponding class label  $y_n$  and is defined by different variables, which can be numeric (values are real numbers) or categorical (take values in a finite set in which there is no order relation). Such vectors  $x_n$  are also known as feature vectors.

Based on these data, a learning algorithm tries to find a function  $f : \mathbb{R}^d \rightarrow 1, 2, \dots, C$  able to predict the class of new items and whose classification error is minimal. This objective function  $f$  is an element of a family of functions  $\mathcal{F}$ , which is called hypothesis space.

The application developed in this thesis is working with Random Forest algorithm. This algorithm is one of the most used tool for classification and regression. It relies on building several predictors each one trained with a different training subset data, and aggregate predictions into a final one. This chapter is meant to explain this algorithm in depth. So we will focus first on Decision Trees.

### 3.2 Decision Trees

To understand Random Forest we must first understand Decision Forest Trees. A decision tree  $T$  is an ordered sequence of questions where the question depends on the answer to the current question. Those questions are raised about the variables that define each element  $\mathbf{x}$  in order to end up assigning a particular class  $\mathbf{y}$ . This procedure, with their corresponding questions and forks, is naturally represented by a tree.

In a decision tree the first node is called the root node, which is in turn connected to other nodes until reaching the leaf nodes, those without descendants. Each internal node

is assigned one of the questions in sequence while each leaf node it is assigned a label of class. Thus, the question of the root node is made to the whole  $\mathcal{S}$ , which subdivide itself until reach the leaf node.

Graphic example of the explanation:

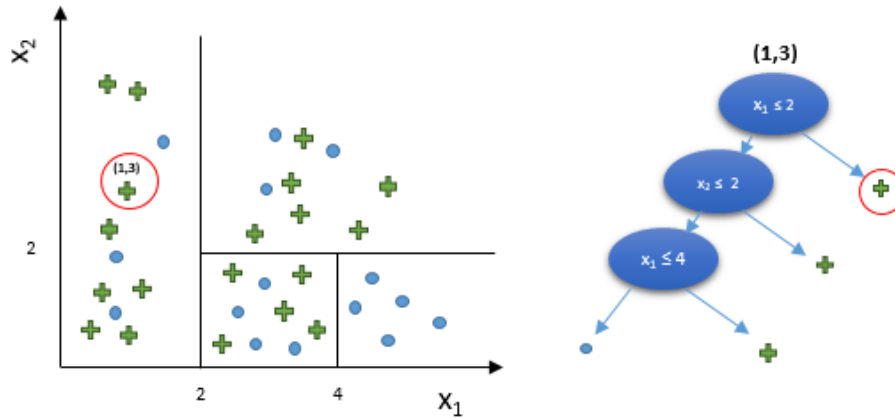


Figure 3.1: Decision tree example in  $\mathbb{R}^2$  space: right decision.

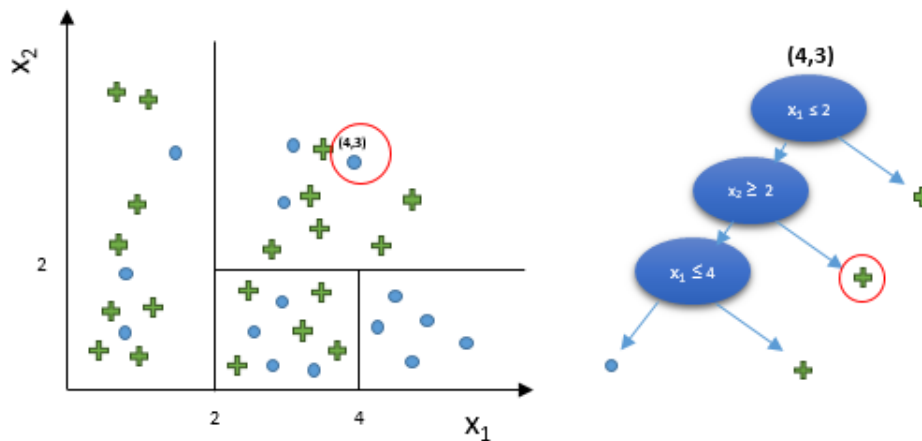


Figure 3.2: Decision tree example in  $\mathbb{R}^2$  space: wrong decision.

The figure 3.1 is an example of a right decision of our tree. There is an input, in this case a cross with the value (1,3). Once this input has been introduced in our tree the first question to classify is: Is the first value lesser than two? In this case there is a one as a value so the tree directly classify this sample as a cross, which is correct in this case. The figure 3.2 shows a fail in the decision. In this case the sample is (4,3) which means that the first question is not enough to classify it. Then the classificatory goes to the second question. Is the second value greater than 2? The condition is satisfied, finally the sample is classified as a cross again. Which is incorrect in this case.

After explaining the basics of decision trees it is interesting to note two major advantages over other classification algorithms:

- Allow to include categorical variables in the set of attributes of the elements  $\mathbf{x}$ .
- They can to be interpreted and understood as sets of rules such as if and then. Which makes decisions based on them they can be justified. Other learning algorithms,

such as neural networks, act as black boxes in which the decision is difficult to explain.

### 3.3 Ensembles of classifiers

Ensembles of classifiers are systems in which the results obtained by a series of individual classifiers are combined to predict labels of new examples, so that the accuracy of the assembly is greater than that obtained by each of the classifiers individually.

This section will explain the best known and currently used. Especially it will do more emphasis upon Random Forest, which is the method used in this thesis.

#### 3.3.1 Bagging

One of the most widespread methods and also one of the simpler to build sets of classifiers is bagging. The term bagging is an acronym of bootstrap aggregating since the algorithm constructs each set classifiers from what is known as a bootstrap sample, which are generated by taking of the training set as many elements with replacement as this contains. Thus,  $T$  sets of the same size as the original set are generated (one for each classifier). Once built the set, the outputs of each of the classifiers are combined by a non weighted polling.

Due to each individual classifier is built from a sample of the training set in which  $N$  elements, not all are used in each classifier, this fact has been thought in order to achieve independence between the classifiers. Therefore, the accuracy of each individual classifiers is lesser than would be obtained if a classifier would have been built with the whole but, when they are combined, the errors of each one are offset and the accuracy would get an improvement compared with one that only uses all data.

Bagging is an approach based on the manipulation of the training data, in which it is vital that the basic classifier used is unstable (high variance). In fact, bagging reduces very significantly the generalization error for such classifiers while it may not affect the performance in stable classifiers. This is so because bagging reduces the variance of the base algorithm, stabilizing the classification obtained by polling and getting predictions do not fluctuate much when using different sets of training. As a result, decision trees are good candidates on which apply bagging although this technique produce the loss of one of its great advantages, the easy interpretation of the trees in rules fashion.

Another advantage of this method is its high scalability: each of the classifiers can be built (and evaluated) in parallel since the construction of each is completely independent to the others, which means that the time required to construct a set of classifiers does not increase proportionally to the number of classifiers used.

### 3.3.2 Boosting

Another effective technique used to build sets and classifiers is boosting [24]. Boosting is an adaptive algorithm in which each classifier is built based on the results obtained in previous classifiers by assigning weights to each of the training samples: patterns that have been misclassified by previous classifiers will be more important when building a new classifier. Thus classifiers focus on those examples that are more difficult to label correctly. In this case, once the classifiers are built, the outputs are combined in a weighted way according to the importance of each classifier in order to obtain the result of the whole.

Originally, boosting was defined as a technique which significantly reduces the error of any algorithm whose error is a little lower than the random classifier (weak algorithm). In practice, is a method which is combined with decision trees, which are not considered weak. Whichever is the algorithm used, there are two variants of boosting: resampling and reweighting. In the case that the algorithm can be adapted to handle weighted data, all examples and their respective weights are used by each base classifier, which is responsible for taking them into account (boosting by reweighting). Otherwise if the algorithm can not handle weighted data, it is necessary to perform a sample with replacement depending on the weight of each sample. So that, the classifier receives a data set without weight (boosting by resampling).

As for the results, boosting is one of the best algorithms to build sets of classifiers. This is because boosting combines two effects: it reduces bias of the base classifier since it forces them to not make the same mistakes and also reduces the variance by combining by polling different hypotheses.

However, it presents problems of generalization in the presence of noisy data because the classifiers focus on the difficult examples regardless if the data is valid or not. Instead, bagging seems able to exploit the noise data to generate more different classifiers with very good results for these datasets. Finally, another disadvantage is that boosting is not parallelizable since each classifier is built based on the results obtained by the previous classifier.

### 3.3.3 Random Forest

Finally the algorithm used in this thesis is reached. This application uses Random Forest since as shown in the publication *Learned lessons in credit card fraud detection from a practitioner perspective* [27] this algorithm outperforms the others in this type of environment.

Thus, as mentioned above, there are various techniques for constructing sets of classifiers, which are aimed at increasing diversity among individual classifiers. In many cases, the way of how to carry out these techniques is from a set of random numbers:

- In bagging [25] each classifier is built from a bootstrap sample, which is generated using as many random numbers as elements have the training set.
- In random subspace each base classifier uses only a subset of randomly selected attributes from the total variables. Thus, each classifier is restricted to a random



subspace of attributes.

- Randomization introduces randomness in the learning algorithm, building sets of classifiers with decision trees where the cutoff value is randomly selected among the best  $F$  possible cuts.

If the basic classifier used is a decision tree, the concept random forest [26] covers all these techniques. Any set of classifiers where the base classifier is a decision tree constructed from a table of random numbers  $\Theta_t$ , where  $\Theta_t$  are independent and identically distributed and the result of the final classifier is obtained by unweighted polling is known as random forest.

The purpose of such methods is to inject to the algorithm randomness to maximize the independence of the trees maintaining reasonable accuracy. In the case of random forests, these qualities are measured in the whole and are denoted as strength and correlation. The strength of the set of classifiers is defined as

$$s = \mathbb{E}_{X,Y} \text{mr}(X, Y) \quad (3.2)$$

Where  $\text{mr}(X, Y)$  is the margin function of a random forest that in case of two classes is defined as

$$\text{mr}(X, Y) = \mathbb{E}_{\Theta}[c(X, \Theta = Y)] - \mathbb{E}_{\Theta}[c(X, \Theta \neq Y)] = 2 \cdot [c(X, \Theta = Y)] - 1 \quad (3.3)$$

where  $\mathbb{E}_{\Theta}[c(X, \Theta = Y)]$  is the limit of the proportion of trees  $c_t$  that, given a pattern  $x$ , classifies correctly when  $T$  is increased:

$$\frac{1}{T} \sum_{t=1}^T I(c_t(x, \Theta_t) = Y) \rightarrow \mathbb{E}_{\Theta}[c(X, \Theta = Y)] \quad (3.4)$$

The concept of margin in the set of classifiers is used to measure the certainty with which the set is right or wrong in its prediction as it is the difference between the proportion of trees that are right and which are wrong. Thus, the margin is a definite value in the range  $[-1, 1]$  being positive when the sample is classified correctly and negative otherwise. Therefore, the higher average margin received by a forest on a set of random data, the greater the strength.

Furthermore, also in the case of problems of two classes, the correlation between the trees within the set is measured as follows:

$$\bar{\rho} = \mathbb{E}_{\Theta, \Theta'}[\rho(c(\cdot, \Theta), c(\cdot, \Theta'))] \quad (3.5)$$

where  $\rho$  It corresponds to the coefficient of correlation between two random variables, where class labels should be fixed as  $+1$  and  $-1$  and where  $\Theta$  and  $\Theta'$  are independent with the same distribution. In this case, it is desirable that the correlation of the set is minimal in order to increase the independence between different trees of the set.

Although all the above techniques are aimed at reducing the correlation of the classifiers maintaining its accuracy and all are able to reduce the classifier error, none achieved in general, the performance obtained by methods which assign weights adaptively during

construction of the set of classifiers. However, random forests were able to equate and even improve the accuracy achieved.

Then, the pseudocode of random forest is:

```

1 Inputs set $ D = \{(x_{\{n\}}, y_{\{n\}}) \mid n = 1, 2, \dots, N, x_{\{n\}} \in \mathbb{R}^d, y_{\{n\}} \in \{1, 2\}\} $
2         Number of trees: T
3         Number of variables to select in each node F
4
5 for t := 1 to T do
6         $ L_{\{bt\}} := \text{Bootstrap sampling (D)} $
7         $ c_{\{t\}} := \text{Build random tree (L_{\{bt\}}, F)} $
8
9 Output: C(x) = $ \arg \max_{\{y\}} \sum_{\{t=1\}}^{\{T\}} I(c_{\{t\}}(x) = y) $
```

With all this is worthy to distinguish:

- One of the most surprising results of experiments carried out in [26] is that the generalization error is not very sensitive to the number of randomly selected variables on each node. The measures of strength and correlation of all classifiers justify this fact, because if the strength and the correlation of a set of classifiers built with different values of F is measured, it is observed that the force increases to a certain point (lower than the total number of variables) from which it is stabilized, while the correlation between classifiers always increases when the value of F is increased. Therefore, there is a certain optimum F value that for maximum strength, the correlation of classifiers is minimal though in any case the oscillations in the error by using different values of F are not very significant.
- By increasing the number of trees, the generalization error converges, in the case of two classes to:

$$\mathbb{E}_{X,Y}[I(\mathbb{E}_{\Theta}[c(X, \Theta) = Y] - \mathbb{E}_{\Theta}[c(X, \Theta) \neq Y] < 0)] \quad (3.6)$$

Therefore, in summary, the five major advantages of this algorithm are:

1. It is one of the most accurate learning algorithms available. For many data sets, it produces a highly accurate classifier.
2. It runs efficiently on large databases.
3. It is robust to noisy data. Being a method based on bagging inherits this stability against noise.
4. It is computationally faster than bagging, that evaluates all possible cuts for all variables, and boosting, which is sequential.
5. Easy to implement and parallelize.

In the other hand the most important disadvantages are:

1. It can be overfit for some datasets with noisy/regression tasks
2. The variable importance scores from random forest are not reliable for categorical variables with different number of levels.

# Chapter 4

## Application description

### 4.1 Introduction

After all the theoretical explanation we are now ready to start with the practical part of this thesis.

As explained in the introduction of this thesis, this project aims mixing machine learning with big data. To do this, the environment with which we will face is the banking transactions.

Nowadays our planet is around 7 billion inhabitants. Day by day people increasingly have more access to credit and debit cards. For example, Figure 4.1 shows as only US payment by debit cards has almost tripled in nine years. If we add the number of transactions made by credit and debit card in the US in 2012 we see that we reach the impressive amount of 75 billion annual transactions.

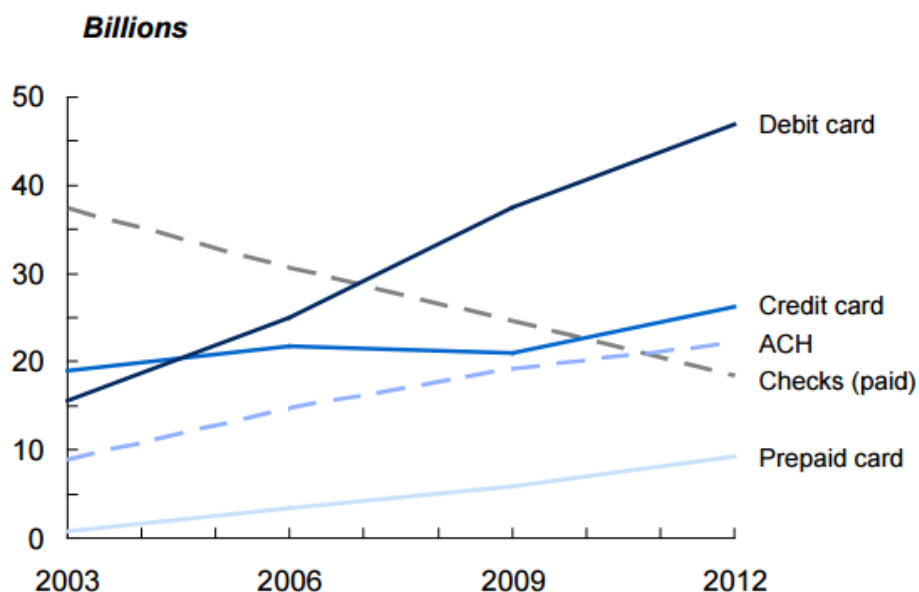


Figure 4.1: Chart with the trend of noncash payments by number and type of transaction <sup>2</sup>.

As it is expected with more transactions the likelihood of fraud increases. In fact the the estimated annual number of unauthorized transactions (third-party fraud) in EEUU in 2012 was 31.1 million, with a value of \$6.1 billion. [21]

## 4.2 Summary

Looking at these numbers it is normal for the detection of fraud in transactions to be a very important research field in which many companies are investing. In the case of this thesis we will work with real-time transactions provided by the company ATOS<sup>3</sup>.

To explain in more detail this application, we will return to the diagram used in Section 1.3.

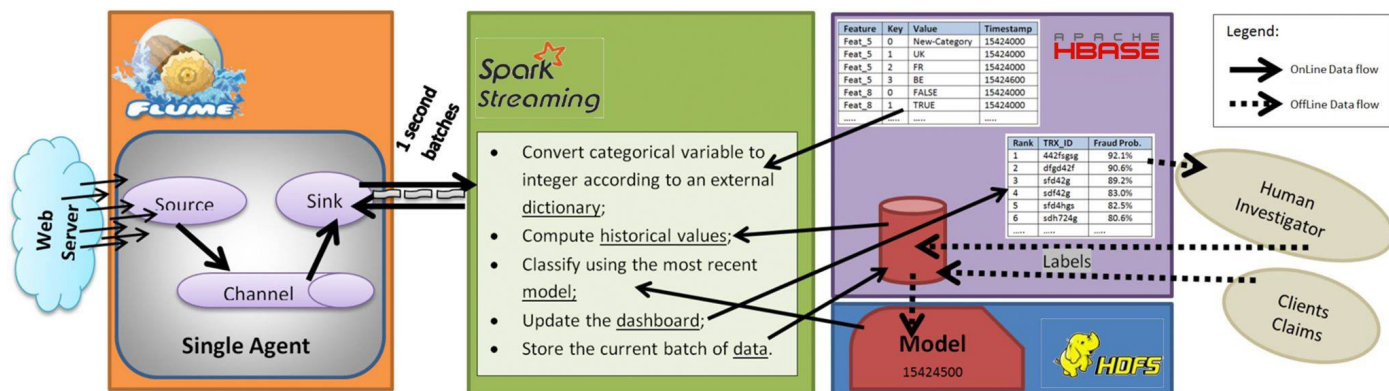


Figure 4.2: Thesis schema.

For starters we have a Flume agent to receive real-time data from the web server. This data is received by the source, which will send them to the channel in order to be gathered. These data will be extracted by Spark through a custom Sink (as discussed in the second spark closer integration with flume). After that Spark will create streaming data batches in each batch interval. This stream of batches will be processed by Streaming and in our case it will take care of five steps:

- Convert the categorical variables inside the transaction into an integer variable according to an external dictionary.
- Compute historical values.
- Classify using the most recent model (models will be saved with different timestamps).
- Update the dashboard.
- Finally store the current batch of data.

<sup>2</sup>[https://www.frb.services.org/files/communications/pdf/research/2013\\_payments\\_study\\_summary.pdf](https://www.frb.services.org/files/communications/pdf/research/2013_payments_study_summary.pdf)

<sup>3</sup><http://atos.net/en-us/home.html>

## 4.3 A deeper look into the application

This section is intended to give a more detailed explanation of the steps performed by Spark Streaming as well as the difficulties that have emerged in the development. We will place particular stress upon the operation of the dictionary as well as in the operation of the dashboard and its use for the update of our model.

This application was initially designed to save data in three HBase tables (one for the dashboard, one for the dictionary, another one for the data prepared to train our Random Forest and finally a last one to store input raw transactions) in order to then work with them through Spark. However, we had to change the approach due to open multiple HBase tables Spark is not trivial and so we had not enough time to implement it. This was definitely the biggest difficulty we encounter in developing this application. To overcome this problem we made a number of changes that are the following:

- On one hand the new dashboard would include more data than originally thought and would join with the original transactions to finally store all this information in HBase. Finally, selecting the columns needed, we could use this dashboard to train our Random Forest.
- On the other hand we have stored the dictionary in HDFS directly, instead of saving it in HBase.

### 4.3.1 Dashboard part

To explain the final structure of our Dashboard we believe that a picture is worth a thousand words:

Transaction 8213Ga		feature 0		...	feature 49	Timestamp
Label	P(fraud)	UK	1		1.87	1231231

Figure 4.3: Here we show the structure of our Dashboard. We can see how complete is because it includes all the possible data. Besides this example shows how if one feature is detected as categorical two column into our column family will be created. One with the categorical value and another one with the integer value associated.

We can see how this dashboard also includes the *Label* that tells us if the transaction with which we are treating is fraudulent or not. This field is initially empty (our program is responsible for providing the likelihood that a transaction is fraudulent, but do not determine it). The ones in charge of filling this field will be the detectives. Once labelled, after a  $\Delta T$ , transactions will become part of the training database training of our Random Forest. In this section timestamp field is vital because we have to train the algorithm to select the new transactions and not the older ones.

Conceptually this would be the followed flow:

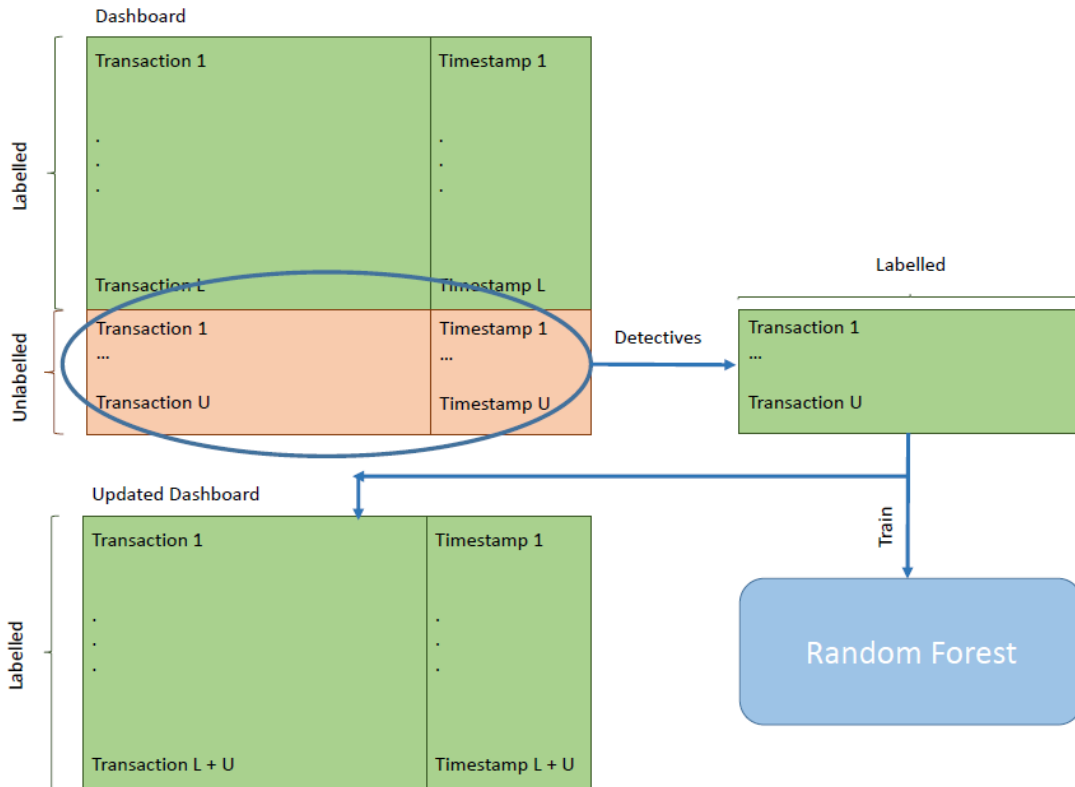


Figure 4.4: Random Forest and Dashboard update.

Once we have trained our Random Forest a new model will be created. This model is associated with a timestamp in order to identify it.

### 4.3.2 Dictionary part

The other key file in this application is the dictionary. Through it, at the time that we have a categorical variable, this variable will be translated and if it does not exist it will automatically create a new entry. Let's see an example of the structure of our dictionary:

Feature Name	Key	Value	Timestamp
Feature_5	0	New_categorical	1231231
Feature_5	1	UK	1231312
Feature_5	2	USA	1231453
Feature_5	3	SPA	1235001
Feature_5	4	BE	1238007
Feature_0	0	False	1231231
Feature_0	1	True	1231312

Figure 4.5: Dictionary example.

With this example we can see a basic dictionary structure where if we get a categorical variable we first need to check if it exists in it. In the case that the variable exists it will replace the key for its value. Otherwise it will create a new variable with the corresponding mapping.

As you have probably noticed the explanation above works perfectly for an offline system, but not for a system in real time. That is why the dictionary above includes the

*Timestamp* field and the *New\_categorical* value. Let's understand the use of each of these variables:

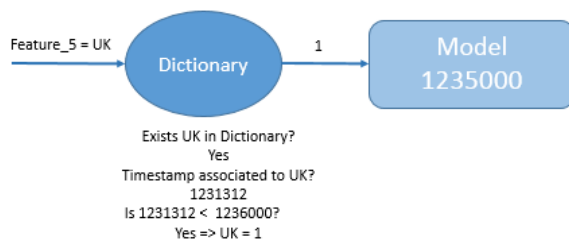
- *New\_categorical*: This variable is used to associate a value of 0 to new variables, in order to introduce our feature on the model.

Below is a picture showing different cases explained using the dictionary Figure 4.5.

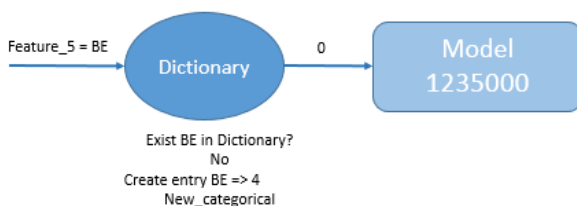
- *Timestamp*: The timestamp serves as a validator. Suppose we get a new variable that does not exist in our dictionary. To this one will be assigned the *New\_categorical* variable and thereupon will create a new entry in the dictionary. Now suppose this new feature comes back with the new value. Following the procedure it will look if there is a categorical variable with the same key, in this case will exist, so in theory it should use the new value created. What it happens? That our model is not trained yet with the new value, therefore if we introduce the new dictionary entry our model it would not understand it. *Timestamp* solves this problem. As discussed in the section of the dashboard, every time we train the Random Forest it gives to the model used a timestamp. This is done to make sure that the variables in the dictionary are in range of the Model time. In the case that the value of a timestamp of a categorical variable is greater than the model, the program automatically continues using the value 0 as the value to be entered into the model.

Below there is a picture that shows different cases explained using the dictionary Figure 4.5.

- **First case:** Existing feature entry with timestamp feature < timestamp Model.

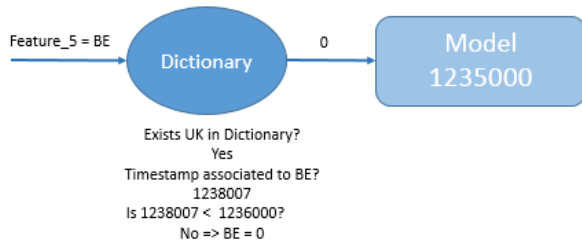


- **Second case:** Non existing feature.





- **Third case:** Existing feature entry with timestamp feature >timestamp Model.



# Chapter 5

## Deployment

Having explained the functioning of our application we will move to the chapter of the deployment. This chapter aims to detail:

1. The cluster where we work.
2. Give the steps to follow in order to reproduce the environment where we work.
3. Submit our programs.

### 5.1 IRIDIA cluster

In this thesis we have been fortunate to work in a high-performance cluster designed to primarily to work with Big Data. In this cluster we have found a very powerful hardware and software that is perfectly suited to our needs. Simplifying the deployment in our work.

#### 5.1.1 Hardware setup

The cluster infrastructure is located at IRIDIA. It has 1 machine managing the cluster, 2 master machines (32 Gb RAM) and 12 slave machines. All this resulting to a 48 TB of disk space usable for Hadoop and Spark. The architecture overview is shown in Figure 5.1, and details are reported in the Table 5.1.

Table 5.1: BridgeIRIS hardware infrastructure details.

Nodes	RAM	Disk Space	Description
Master	8 GB	4 TB	Node for communicating with ULB network
Storage	8 GB	2 TB	Node non Hadoop used for storing the network of citations (ElasticSearch Neo4J)
2 Nodes 001-002	32 GB	4 TB	Primary and secondary Namenodes (all services for CDH)
12 Nodes 003-010	16 GB	4 TB	Slave/Worker

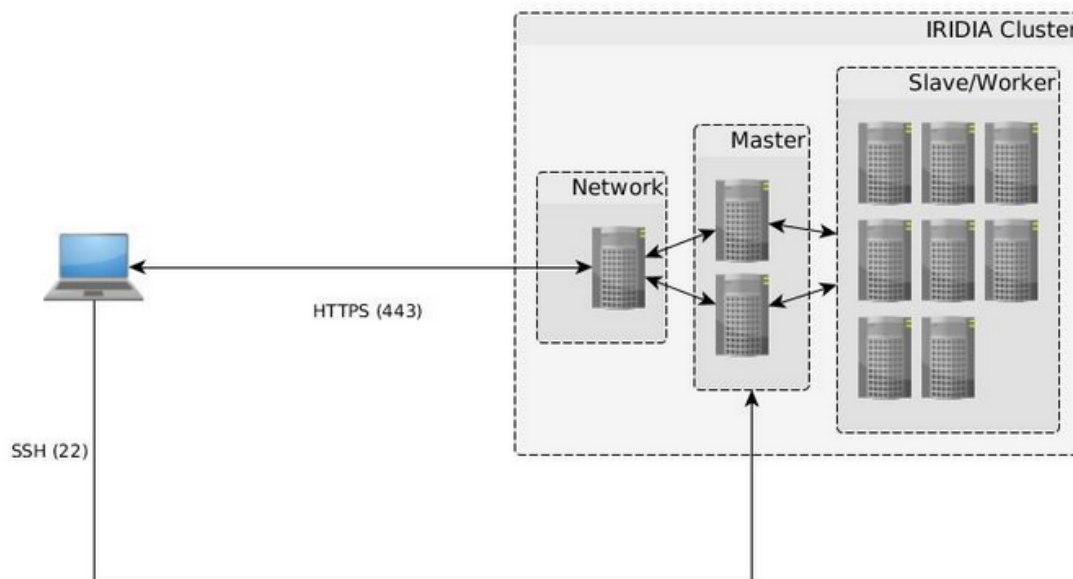


Figure 5.1: BridgeIRIS architecture overview.

## 5.1.2 Software setup

### Distributed Framework

All machines are installed under Ubuntu 12.04 LTS Server except for the node linking the cluster to the ULB network which is under Ubuntu 14.04 LTS Server.

Apache Hadoop has been deployed using Cloudera Express (CDH 5.1.3), which is an open source software distribution providing an unified querying options (including batch processing, interactive SQL, text search, and machine learning) and necessary enterprise security features (such as role-based access controls). The active frameworks are : HDFS et Yarn (MapReduce2) (core Hadoop), HBase, Hive, Hue, Impala, Oozie, Sqoop 2 and ZooKeeper.

### User interfaces

*Hue* is an open source Web interface that supports Apache Hadoop and its ecosystem. Username and password are required to access to the interface, through which it is possible to browse the data in HDFS or HBase and query the data using Hive, Pig or Impala. In

Figure 5.2 is reported one screenshot of the interface.

Registros	ID	Nombre	Estado	Usuario	Maps	Reduces	Cola	Prioridad	Duración	Submitted
	1432105438214_0393	org.apache.spark.examples.streaming.FlumePollingEventCount	FAILED	smheman	100%	100%	root.smheman	N/D	1m:8s	06/15/15 11:01:20
	1432105438214_0312	MyRF2	FAILED	smheman	100%	100%	root.smheman	N/D	4m:28s	06/04/15 02:39:14
	1432105438214_0311	MyRF2	FAILED	smheman	100%	100%	root.smheman	N/D	4m:44s	06/04/15 02:36:42
	1432105438214_0309	MyRF2	FAILED	smheman	100%	100%	root.smheman	N/D	5m:21s	06/04/15 02:31:49
	1432105438214_0307	MyRF2	FAILED	smheman	100%	100%	root.smheman	N/D	5m:22s	06/04/15 02:21:35
	1432105438214_0299	MyRF2	SUCCEEDED	smheman	100%	100%	root.smheman	N/D	4h:48m:37s	06/02/15 02:38:07
	1432105438214_0298	MyRF2	SUCCEEDED	smheman	100%	100%	root.smheman	N/D	5h:20m:56s	06/02/15 02:05:37

Figure 5.2: Hue interface (job browser tab).

## 5.2 Cluster manager

This section is very short, but we think that is worth to be included because it is important in order to have clear how the applications are deployed through the cluster. Besides, we have discussed between the use of several cluster managers.

To begin we need to understand what is a cluster manager. To do that we will proceed to define this concept: A cluster manager is a program that runs on one or all cluster nodes. The cluster manager works together with a cluster management agent. These agents run on each node of the cluster to manage and configure services, a set of services, or to manage and configure the complete cluster server itself.

Another common uses of a cluster manager are: dispatch work for the cluster and manage the availability, load and balancing of a cluster services. [22].

Then in a simplified fashion we can process in a distributed way with Spark as follows:

In the figure above we see how we can access to HDFS or database in two ways: directly with spark (if one machine) or through a cluster manager. Spark has 3 types: YARN, mesos and standalone.

In this thesis we will use YARN because mainly it was already installed on the cluster and on the other hand offers advantages over its rivals. Below is a list of advantages:[23]

- YARN allows us to dynamically share and centrally configure the same pool of cluster resources between all frameworks that run on YARN. Which means that we can start a job on a cluster, and the use some of this job on others programs like Hive without any changes on configuration.
- YARN offers a very interesting features like schedulers for categorizing, isolating, and prioritizing workloads.

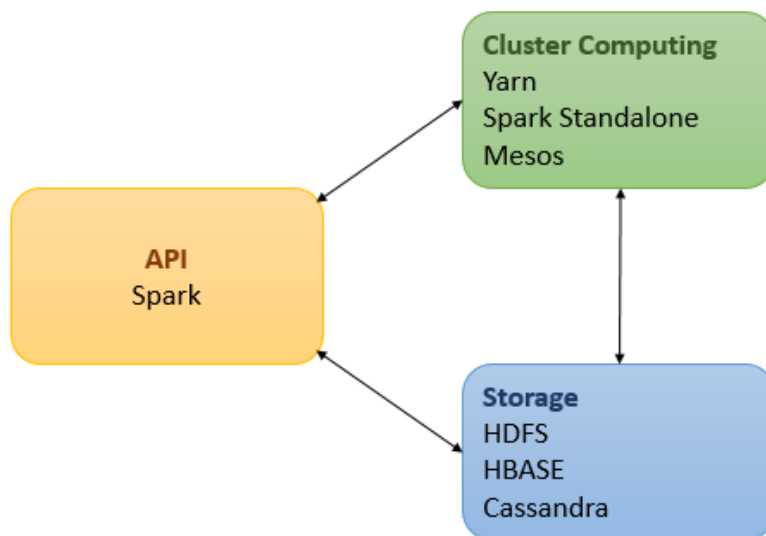


Figure 5.3: Distributed processing with the Spark Framework.

- In YARN we can choose the number of executors to use, whereas in Spark standalone is not possible (the applications will run on every node of the cluster).
- YARN is the only cluster manager for Spark that supports security (i.e Spark can run against Kerberized Hadoop<sup>1</sup> and uses secure authentication between processes).

Finally in Appendix C is detailed in an accurate way all the necessary downloads and integrations related to Spark in order to make work our application.

<sup>1</sup><http://henning.kropponline.de/2014/10/05/kerberized-hadoop-cluster-sandbox-example/>

# Chapter 6

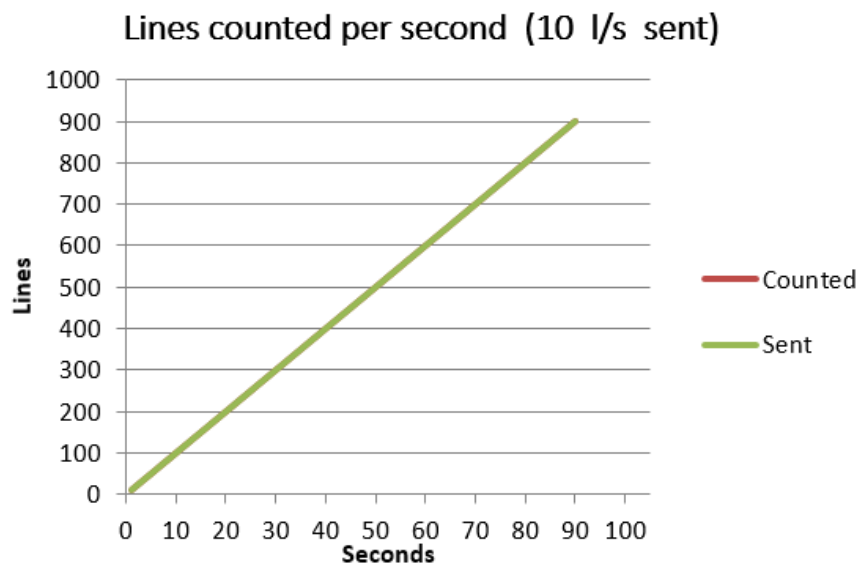
## Results and scalability

In order to test our application performance, we have created a script that was able to send to our application any amount of transactions. The idea is getting a reference from one machine: the maximum amount of lines sent per seconds without appreciate delay between the lines sent and the lines counted. After getting this value we will be able to measure the performance improvement increasing the amount of nodes. It is important to keep in mind that the channel capacity of flume has been set to keep more than 100 MB of transactions in queue, that means that in these tests we wont see any transaction loss.

### 6.1 Scalability

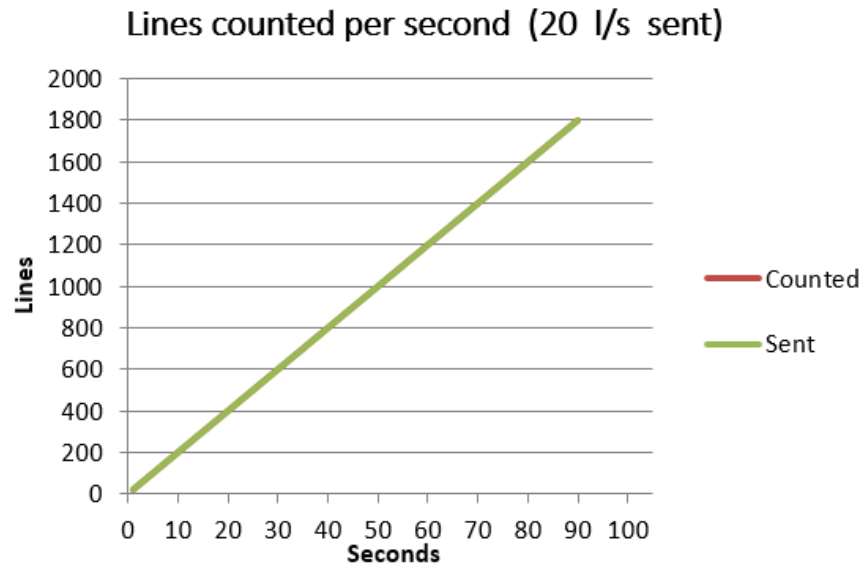
#### 1 Machine:

Sending 10 lines/second to our application.



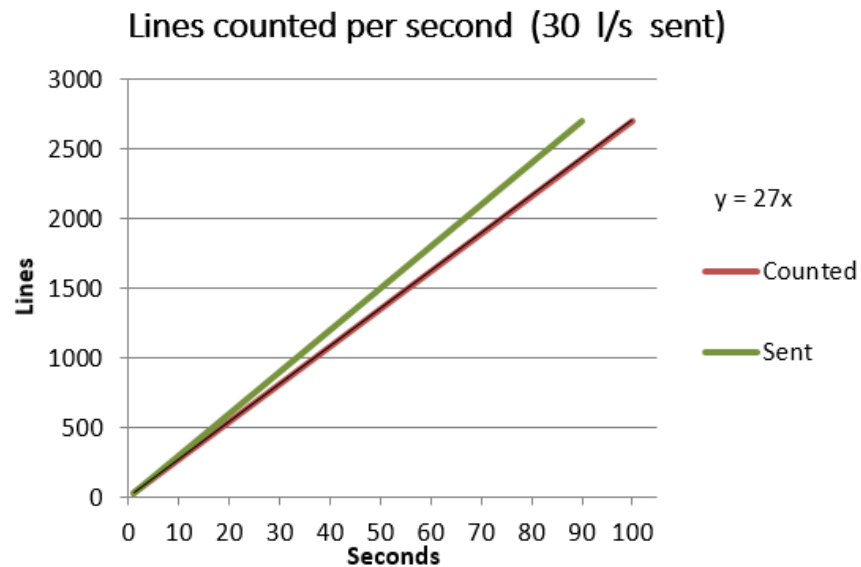
The reception is perfect there is no delay.

Sending 20 lines/second to our application.



Again no delay between both measures.

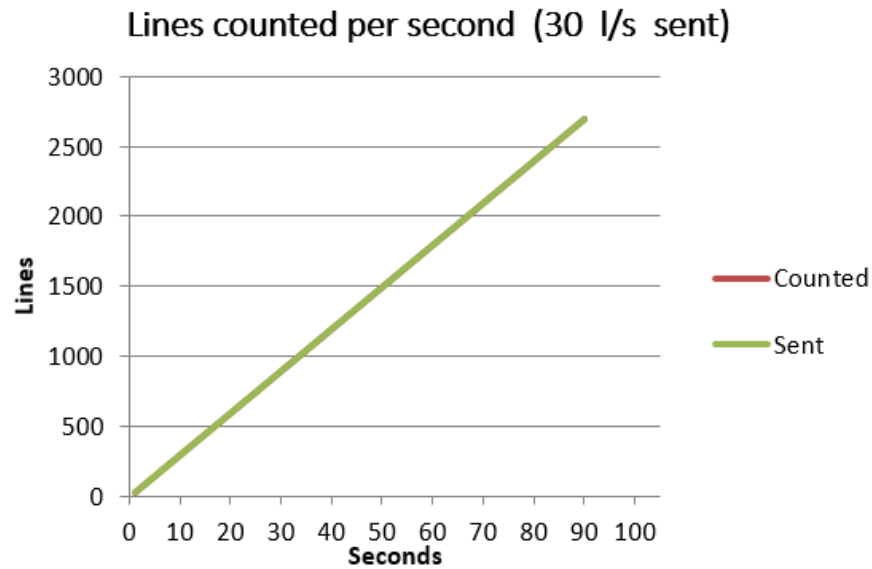
Sending 30 lines/second to our application.



There is a delay between both measures, we have calculated the pendent because the amount of lines sent are linear and the pendent give us the maximum amount of lines sent in this case. Hence, the max amount of lines that our application can accept without delay are 27.

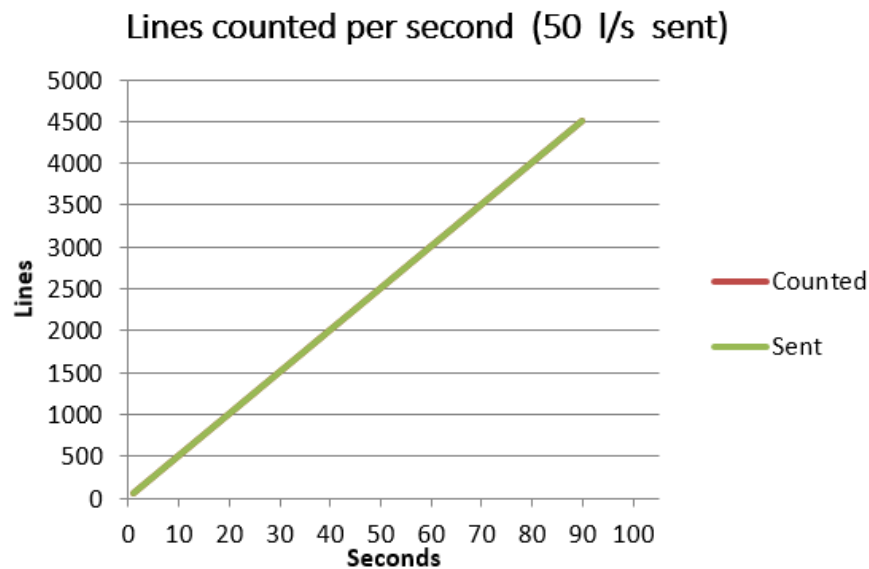
## 2 Machines:

Sending 30 lines/second to our application.



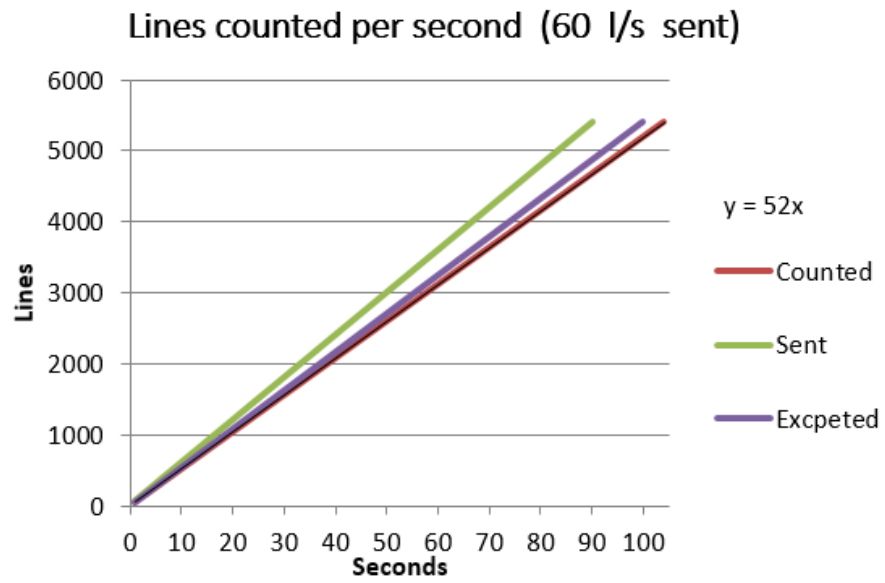
As expected, increasing the amount of nodes resolve the last delay issue.

Sending 50 lines/second to our application.



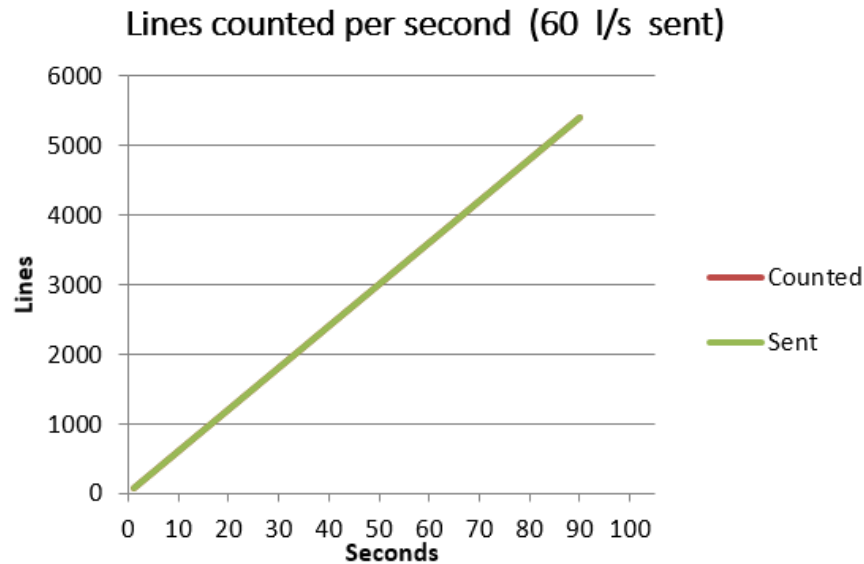


Sending 60 lines/second to our application.

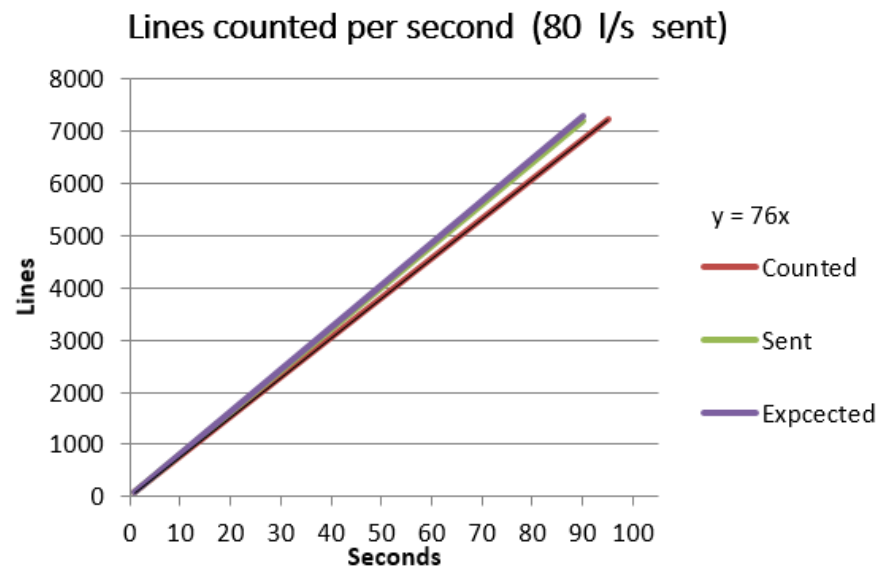


### 3 Machines:

Sending 60 lines/second to our application.

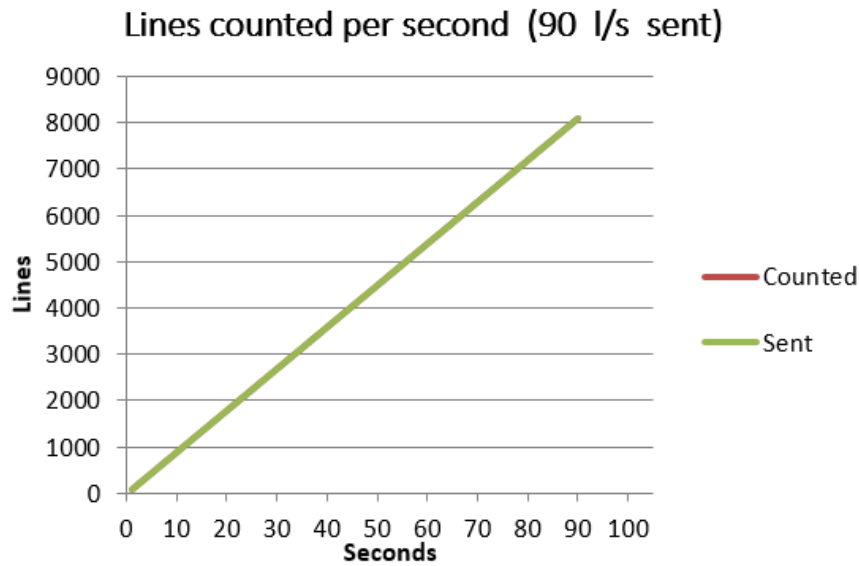


Sending 80 lines/second to our application.

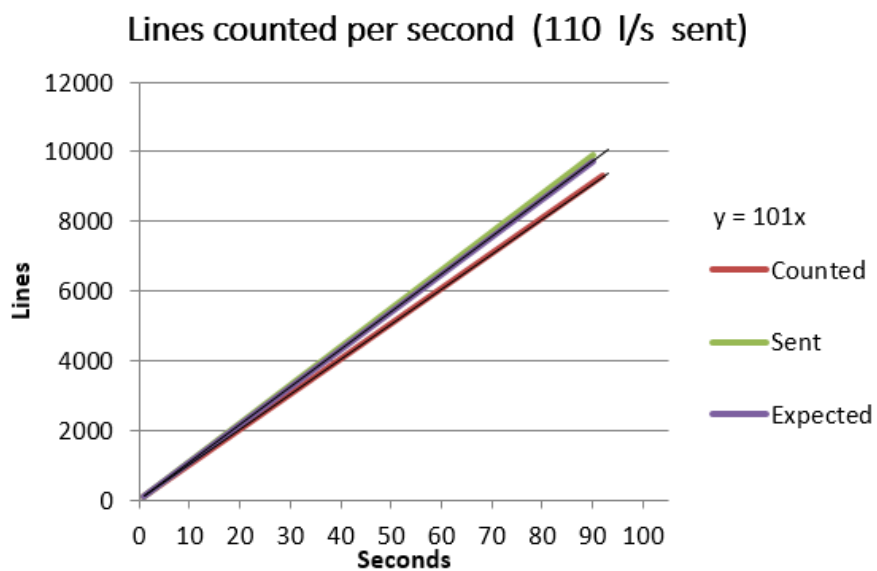


## 4 Machines:

Sending 90 lines/second to our application.

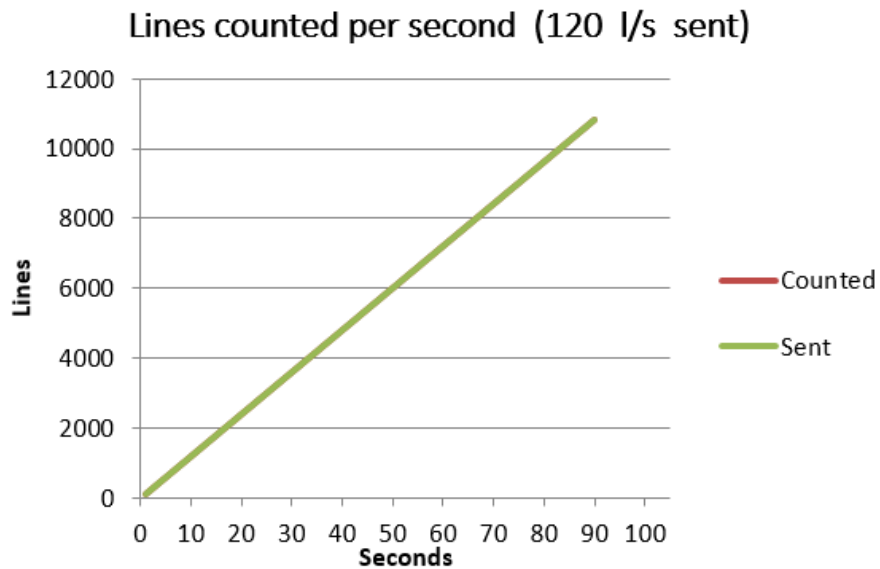


Sending 110 lines/second to our application.

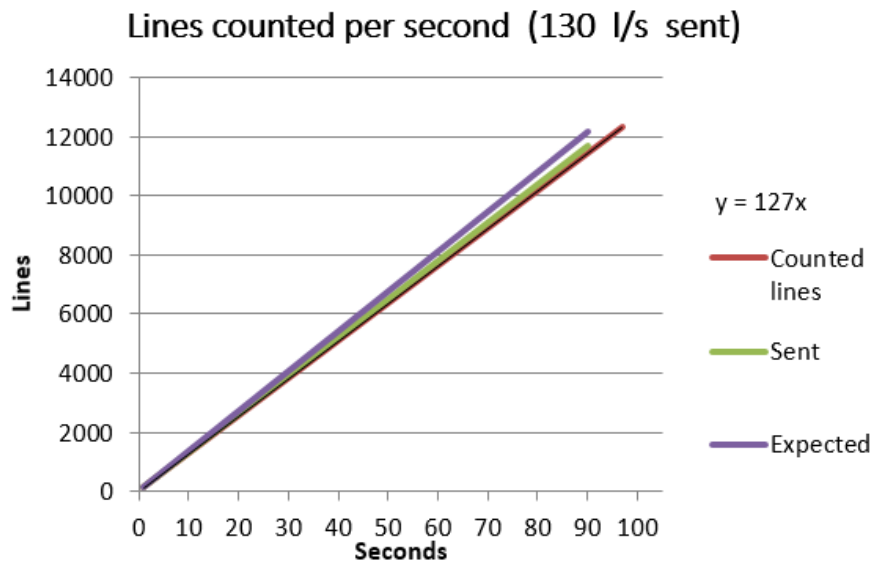


## 5 Machines:

Sending 120 lines/second to our application.

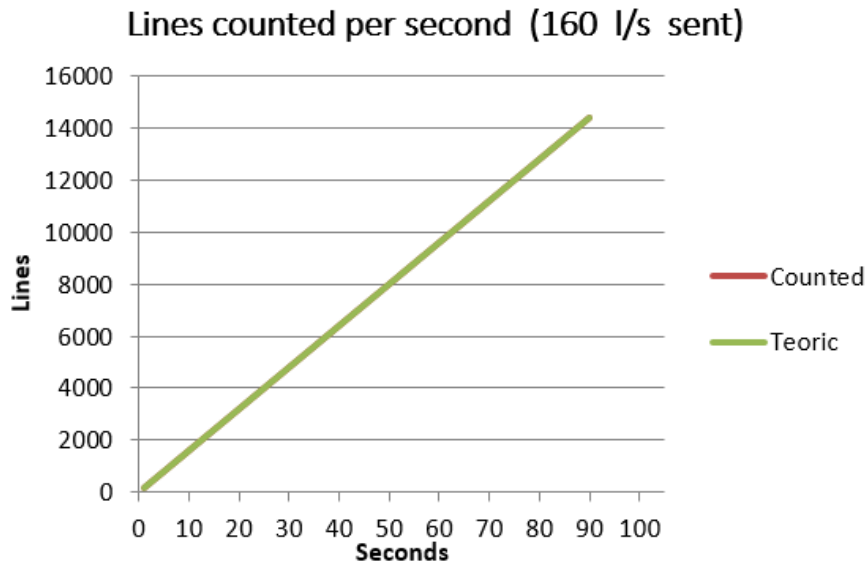


Sending 130 lines/second to our application.

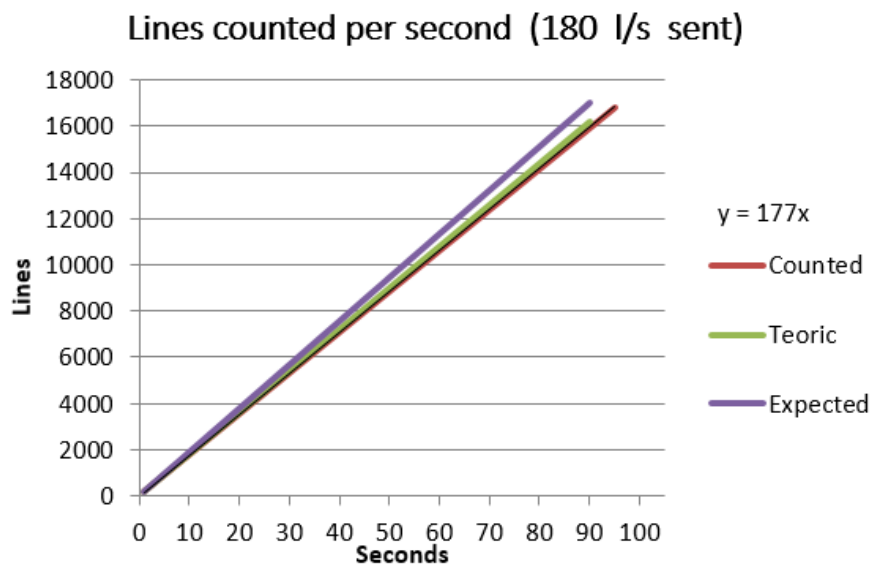


## 6 Machines:

Sending 160 lines/second to our application.

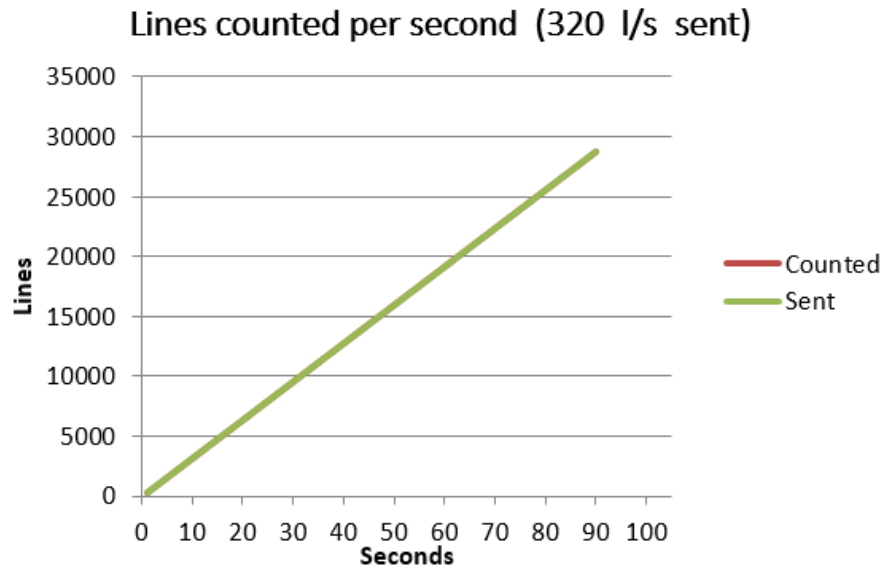


Sending 180 lines/second to our application.

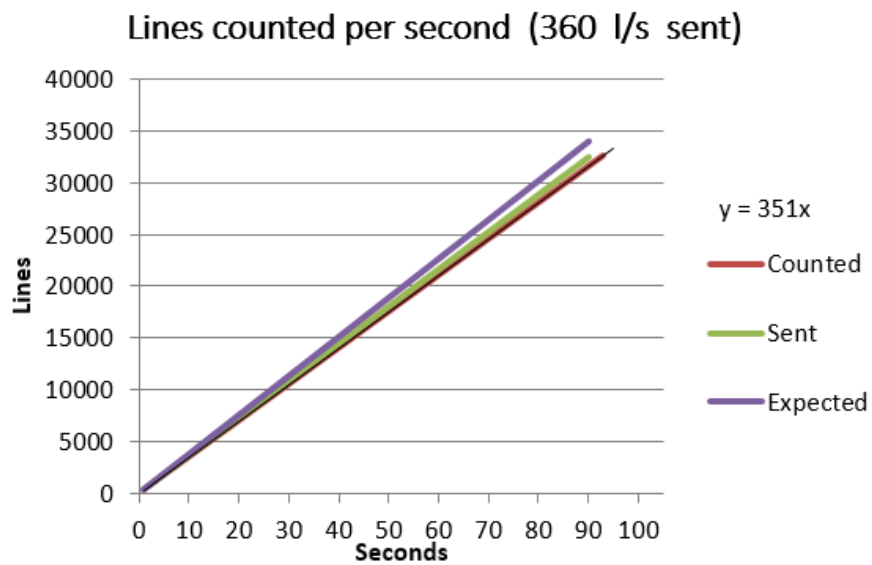


## 12 Machines:

Sending 320 lines/second to our application.

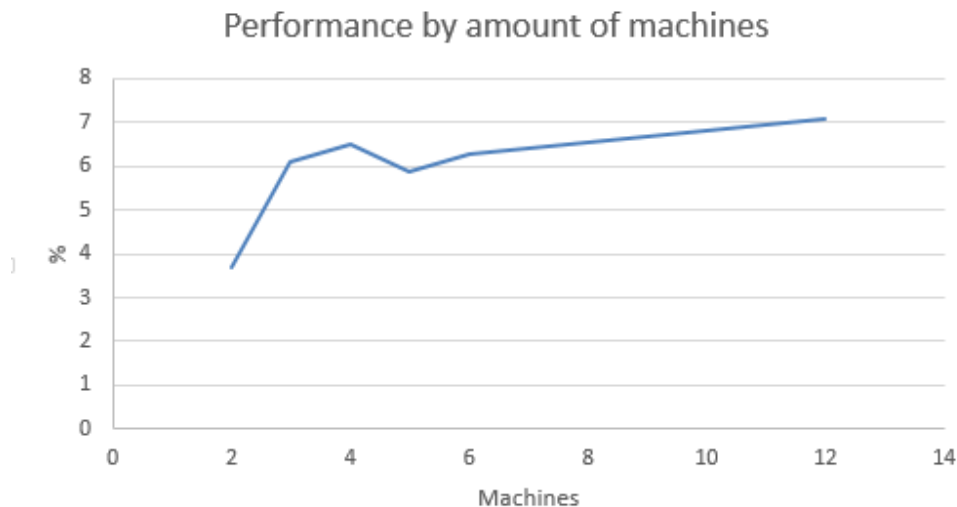


Sending 360 lines/second to our application.



## Relative performance

Taking into account these results we observe the following chart:



This chart represents the relative performance of our application by increasing the amount of machines. This means that if with one machine we were able to send 27 lines, with two machines we send approx 4% less than the expected; with three machines the value is increased to 6% and looks like this trend continues until twelve machines. This is the result of using different machines in the cluster, with this chart we can conclude that there are two machines with better specs than the others, and the rest are very similar.

## 6.2 Random Forest accuracy

As a result of the unavailability of the ATOs investigators, we were not able to train our model as we would like. This caused that this section has been relegated as we have found in a situation of lack of knowledge of certainty of our predictions. Thus, we conducted the test with our labeled dataset splitting 70% of it to training and 30% to test. With this scenario we got an error of 22.3% (approx 22300 errors over 100000 samples) data the certainly we believe we can improve.

# Chapter 7

## Conclusions

In this thesis we have created a system of near real-time streaming working in parallel. The results of these have been very satisfying to have been our first approach with Spark and other tools. Experience with this type of platform has been very satisfactory, all elements are easily scalable and combinable. Fortunately the cluster used has greatly facilitated the implementation of this application as it had all the tools necessary to do so, something that is very positive for implementation but also makes lose knowledge about the deployment.

Regarding the computing paradigm, MapReduce has been a challenge to adapt with it has been a total change in the way of think and programming. There is a big difference in programming an application for a single machine than for a cluster, details like how are the variables distributed in memory have to be taken into account in order to create an efficient application.

Even being very satisfied of this thesis it is noteworthy the disappointment of not being able to deploy our application to its full potential. It was originally intended to be implemented in a real scenario with Atos, but finally due confidentiality issues we could not work with them. So we had to work with one of their allowed dataset of 2010. Yet we believe our system is robust and scalable enough to be able to adapt to a real environment.

Undoubtedly Cloud Computing is already a notable feature of the new technologies and will certainly continue to grow to eventually become a must when it comes to compute complex algorithms, this thesis is a proof of that. We have seen how increasing the number of nodes increased computing capacity greatly. The results are easily extrapolated to all kinds of scenarios that require this kind of computation power.



# Bibliography

- [1] Ibmbigdatahub.com, (2015). The Big Data Hub — Understanding big data for the enterprise. [online] Available at: <http://www.ibmbigdatahub.com/>
- [2] Hadoop: A brief history. Doug Cutting, Yahoo! [online] Available at: [research.yahoo.com/files/cutting.pdf](http://research.yahoo.com/files/cutting.pdf)
- [3] Hadoop.apache.org, (2015). HDFS Architecture Guide. [online] Available at: [https://hadoop.apache.org/docs/r1.2.1/hdfs\\_design.html](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html)
- [4] MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean and Sanjay Ghemawat. <http://static.googleusercontent.com/media/research.google.com/es/us/archive/mapreduce-osdi04.pdf>
- [5] Wikipedia, (2015). Machine learning. [online] Available at: [https://en.wikipedia.org/?title=Machine\\_learning](https://en.wikipedia.org/?title=Machine_learning)
- [6] Learning Spark Lightning-fast data analysis. Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia.
- [7] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., J. Franklin, M., Shenker, S. and Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. [online] Available at: <https://www.cs.berkeley.edu/~matei/papers/2012/nsdi-spark.pdf>
- [8] Pentreath, N. and Paunikar, A. (n.d.). Machine learning with Spark.
- [9] Spark.apache.org, (2015). Spark Streaming - Spark 1.3.0 Documentation. [online] Available at: <https://spark.apache.org/docs/1.3.0/streaming-programming-guide.html>
- [10] Flume.apache.org, (2015). Welcome to Apache Flume — Apache Flume. [online] Available at: <https://flume.apache.org/>
- [11] Shreedharan, H. (2015). Using Flume. Sebastopol, CA: O'Reilly Media.
- [12] Spark.apache.org, (2015). Spark Streaming + Flume Integration Guide - Spark 1.3.0 Documentation. [online] Available at: <https://spark.apache.org/docs/1.3.0/streaming-flume-integration.html>
- [13] Kafka.apache.org, (2015). [online] Available at: <http://kafka.apache.org/documentation.html>
- [14] Garg, N. (2015). Learning Apache Kafka. Birmingham: Packt Publishing.

- [15] Team, A. (2015). Apache HBase™ Reference Guide. [online] Hbase.apache.org. Available at: <http://hbase.apache.org/book.html#datamodel>
- [16] Anon, (2015). [online] Available at: <http://static.googleusercontent.com/media/research.google.com/es//archive/bigtable-osdi06.pdf>
- [17] Es.slideshare.net, (2015). HBaseCon 2012 — HBase Schema Design - Ian Varley, Salesforce. [online] Available at: [http://es.slideshare.net/cloudera/5-h-base-schemahbasecon2012?qid=6884da10-f052-45a6-862f-914c298e983c&v=default&b=&from\\_search=3](http://es.slideshare.net/cloudera/5-h-base-schemahbasecon2012?qid=6884da10-f052-45a6-862f-914c298e983c&v=default&b=&from_search=3)
- [18] Wikipedia, (2015). Scala (programming language). [online] Available at: [https://en.wikipedia.org/wiki/Scala\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Scala_(programming_language))
- [19] Odersky, M. (2015). A Brief History of Scala. [online] Artima.com. Available at: <http://www.artima.com/weblogs/viewpost.jsp?thread=163733>
- [20] Wampler, D. and Payne, A. (2009). Programming Scala. Sebastopol, CA: O’Reilly.
- [21] Anon, (2015). [online] Available at: [https://www.frbervices.org/files/communications/pdf/research/2013\\_payments\\_study\\_summary.p](https://www.frbervices.org/files/communications/pdf/research/2013_payments_study_summary.p)
- [22] Wikipedia, (2015). Cluster manager. [online] Available at: [https://en.wikipedia.org/wiki/Cluster\\_manager](https://en.wikipedia.org/wiki/Cluster_manager)
- [23] Ryza, S. (2015). Apache Spark Resource Management and YARN App Models. [online] Cloudera Developer Blog. Available at: <http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/>
- [24] Yoav Freund, Robert E. Schapire (1999). [online] Available at: <http://cseweb.ucsd.edu/~yfreund/papers/IntroToBoosting.pdf>
- [25] L. Breiman., (1996). [online] Available at: <https://www.stat.berkeley.edu/~breiman/OOBestimation.pdf> [Accessed 18 Oct. 2015].
- [26] L. Breiman., (2001). Random forests. Machine learning
- [27] Andrea Dal Pozzolo, Olivier Caelen, Yann-Aël Le Borgne, Serge Waterschoot, Gianluca Bontempi (2014). [online] Available at: [http://www.ulb.ac.be//di/map/adalpozz/pdf/FraudDetectionPaper\\_8.pdf](http://www.ulb.ac.be//di/map/adalpozz/pdf/FraudDetectionPaper_8.pdf)

# Appendix A

## HDFS architecture

### Writing files to HDFS

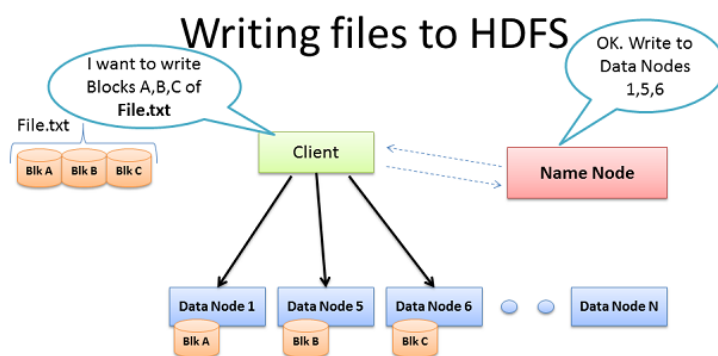


Figure A.1: Writing files to HDFS <sup>1</sup>

<sup>1</sup><http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/>

## Replication system of HDFS

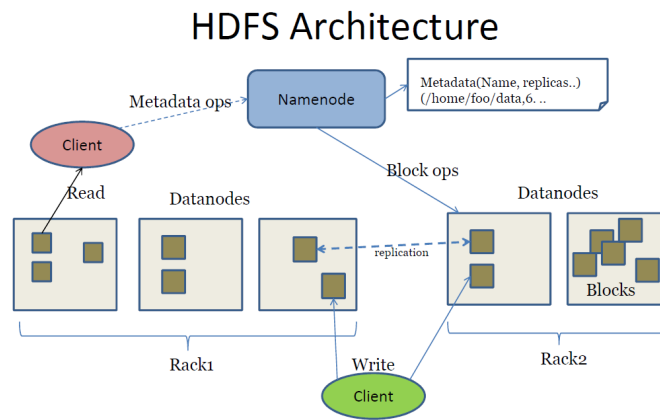


Figure A.2: Replication system of HDFS <sup>2</sup>

<sup>2</sup><https://searchenginedeveloper.wordpress.com/tag/hadoop/>

# Appendix B

## Matrix correlation example

```
1 import org.apache.spark.SparkContext
2 import org.apache.spark.mllib.linalg._
3 import org.apache.spark.mllib.stat.Statistics
4 import org.apache.spark.{SparkConf, SparkContext}
5
6 object MatrixCorrelation {
7     def main(args: Array[String]) {
8
9         val conf = new SparkConf().setAppName("MatrixCorrelation")
10        val sc = new SparkContext(conf)
11
12        val data = sc.textFile(args(0))
13        val parsedData = data.map(x => Vectors.dense(x.split(',').slice(0,100).
14            map(_.toDouble))).cache()
15        val correlMatrix: Matrix = Statistics.corr(parsedData, "pearson")
16        println(correlMatrix)
17    }
18 }
```

# Appendix C

## Deploying Spark

During this thesis we have talk about Spark quite a lot, but still we have not explained how to deploy it. In this section we aim to show how to install Spark in a cluster step by step.

### C.0.1 Download Spark

In order to install Apache Spark we have to go to the official webpage <https://spark.apache.org/download>. Then we click on Downloads, where we will see several downloads available.

#### Download Spark

The latest release of Spark is Spark 1.3.0, released on March 13, 2015 ([release notes](#)) ([git tag](#))

1. Chose a Spark release:
2. Chose a package type:
3. Chose a download type:
4. Download Spark: [spark-1.3.0-bin-hadoop2.3.tgz](#)
5. Verify this release using the [1.3.0 signatures and checksums](#).

In this this thesis we choose the 1.3.0 version and a Pre-built version already configured for hadoop 2.3. This is the version of Hadoop that we use on the cluster IRIDIA, also is the the minimum required one to use the cluster manager YARN. Since we are working remotely on IRIDIA, we used *wget + mirror URL of the download file* to download Spark in every cluster node.

Once we have downloaded Spark we have to download Scala with the version 2.10.4 or above. The download link is: <http://www.scala-lang.org/download/>.

## DOWNLOAD

Choose one of three ways to get started with Scala!



Download Scala 2.11.7 binaries for your system ([All downloads](#)).



[Need help installing?](#)

Again like in the Spark part, we will use the linux command *wget* to download the mirror in every node.

Since Scala is a programming language that runs on top of JVM of Java we also will need to download it. So we will go to the official Java webpage and will select the following link:

Linux x86	119.43 MB		jdk-7u75-linux-i586.rpm
Linux x86	136.77 MB		jdk-7u75-linux-i586.tar.gz
Linux x64	120.83 MB		jdk-7u75-linux-x64.rpm
Linux x64	135.66 MB		jdk-7u75-linux-x64.tar.gz
Mac OS X x64	185.86 MB		jdk-7u75-macosx-x64.dmg
Solaris x86 (SVR4 package)	139.55 MB		jdk-7u75-solaris-i586.tar.Z
Solaris x86	95.87 MB		jdk-7u75-solaris-i586.tar.gz
Solaris x64 (SVR4 package)	24.66 MB		jdk-7u75-solaris-x64.tar.Z
Solaris x64	16.38 MB		jdk-7u75-solaris-x64.tar.gz
Solaris SPARC (SVR4 package)	138.66 MB		jdk-7u75-solaris-sparc.tar.Z
Solaris SPARC	98.56 MB		jdk-7u75-solaris-sparc.tar.gz
Solaris SPARC 64-bit (SVR4 package)	23.94 MB		jdk-7u75-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	18.37 MB		jdk-7u75-solaris-sparcv9.tar.gz
Windows x86	127.8 MB		jdk-7u75-windows-i586.exe
Windows x64	129.52 MB		jdk-7u75-windows-x64.exe

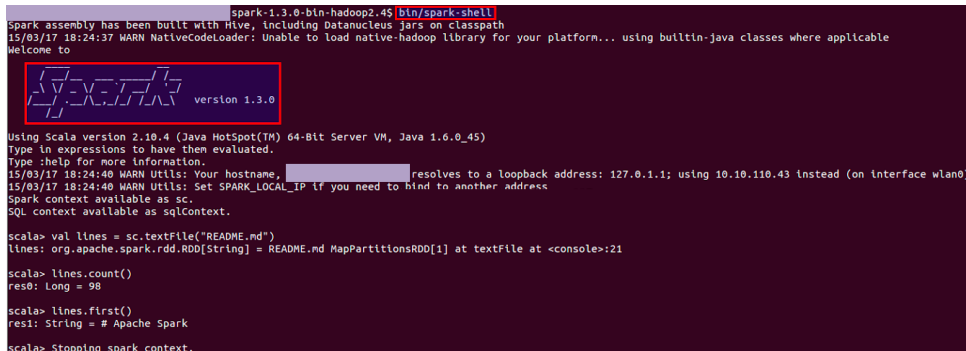
Once we installed all the necessary dependences we finally need to export some environment variables in *.bashrc* file.

```
export JAVA_HOME=/your_path_to_java
export SCALA_HOME=/your_path_to_scala
export SPARK_HOME=/your_path_to_spark
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$SPARK_HOME/bin
```

Finally everything is set up. Now we are able to submit our applications through the cluster. But before starting with this part we believe that is worth to mention the Spark Shell before.

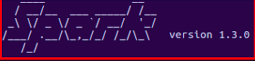
## C.0.2 Spark Shell

Spark shell is a very useful tool which Spark provide us. This allows us to try our code in a shell before compile and submit it. This is very grateful, because Hadoop users who wanted to test their code had to compile it all over again every time they wanted to test something, making the phase of testing quite tedious.



```

spark-1.3.0-bin-hadoop2.4$ bin/spark-shell
Spark assembly has been built with Hive, including Datanucleus jars on classpath
15/03/17 18:24:37 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Welcome to

 version 1.3.0

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_45)
Type in expressions to have them evaluated.
Type :help for more information.
15/03/17 18:24:40 WARN Utils: Your hostname, [redacted] resolves to a loopback address: 127.0.1.1; using 10.10.110.43 instead (on interface wlan0)
15/03/17 18:24:40 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Spark context available as sc.
SQL context available as sqlContext.

scala> val lines = sc.textFile("README.md")
lines: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[1] at textFile at <console>:21

scala> lines.count()
res0: Long = 98

scala> lines.first()
res1: String = # Apache Spark

scala> Stopping spark context.

```

As we can see in the image the access to Spark Shell is simple, we just need to type `bin/spark-shell` within Spark folder.

## C.0.3 Submitting applications

To use `spark-submit` we must create an application folder with its classes. In this example we have created the `test` folder, which contains `Test.scala`. The structure of the application should be set up with the following fashion.

- Main Folder
  - utils
  - app\_creator.sh
  - example
    - Build.sbt
    - Compile.sh
    - Src
      - Main
        - Scala
          - Example.scala

Finally we have to create a `sbt` file in order to compile our application. The content of this file must be as follows:

```

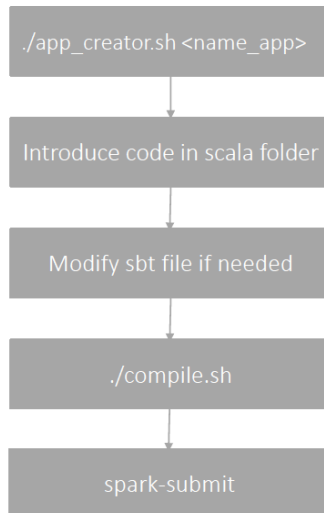
name:= "nameapp" //App name
version:= "1.0" //Version of our application
scalaVersion:= "2.10.4" Scala version
libraryDependencies+= Seq(
"org.apache.spark" %% "spark-core" % "1.3.0",
"org.apache.spark" %% "spark-mllib" % "1.3.0"
) //Dependencies

```



In order to simplify and automatize this process we created two bash scripts: *app\_creator.sh* and *compile.sh*.

Resuming the process to make an application run would be:



## app\_creator.sh

```
#!/bin/bash

if [ $# -eq 0 ]
then
    echo "One argument has to be introduced."
    exit
elif [ $# -gt 1 ]
then
    echo "Too many arguments."
    exit
fi

NAME_APP=$1
NAME_CLASS=${1^} #upper case the first letter

mkdir $NAME_APP
mkdir $NAME_APP/src
mkdir $NAME_APP/src/main

cp utils/build.sbt $NAME_APP/.
cp utils/compile.sh $NAME_APP/.

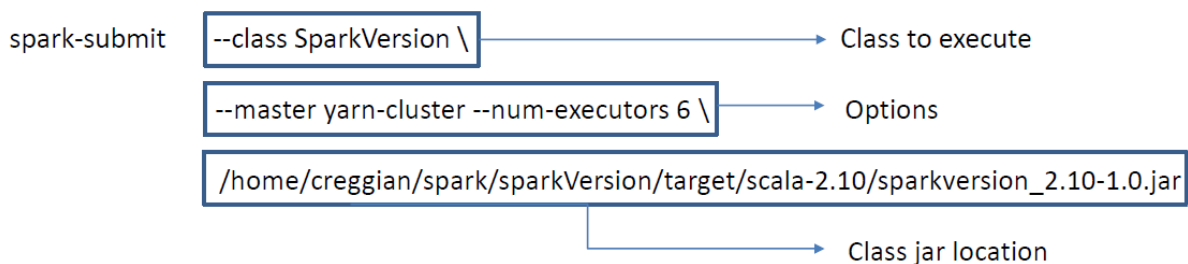
sed -i "s/nameapp/$NAME_CLASS/g" $NAME_APP/build.sbt

mkdir $NAME_APP/src/main/scala
touch $NAME_APP/src/main/scala/$NAME_CLASS.scala
```

---

```
echo -e "import org.apache.spark.{SparkConf, SparkContext}
\n\n object $NAME_CLASS {\n    def main(args: Array[String]) {\n    }\n}" > $NAME_APP/s
```

Once we have set up our environment we just need to understand how spark-submit works. Let's use an example for a better understanding:



## C.1 Spark and flume integration guide

Fortunately in IRIDIA cluster flume was already installed. This means that in this case we just needed to configure the corresponding Flume Agent and integrate it with Spark.

In order to simplify this part we are going to explain directly the configuration file of flume and relate it to the theoretical part in the section 2.2.1.

```

1 #Name the components on this agent
2 a1.channels = c1
3 a1.sources = r1
4
5
6 #Configure sink spark
7 a1.sinks = spark
8 a1.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink
9 a1.sinks.spark.hostname = node001
10 a1.sinks.spark.port = 11111
11 a1.sinks.spark.channel = memoryChannel
12
13
14 # Describe/configure the source
15 a1.sources.r1.type = netcat
16 a1.sources.r1.bind = node001
17 a1.sources.r1.port = 44444
18 a1.sources.r1.max-line-length=5096
19
20
21 # Use a channel which buffers events in memory
22 a1.channels.c1.type = memory
23 a1.channels.c1.capacity = 1000
24 a1.channels.c1.transactionCapacity = 100
25
26
27 # Bind the source and sink to the channel
28 a1.sources.r1.channels = c1
29 a1.sinks.spark.channel = c1
  
```

Let's analyze the first two lines of the configuration file:

```
a1.channels = c1
a1.sources = r1
```

Here we are creating an agent *a1*. In this agent we are defining the name of it channel (*c1*) and source (*r1*). So, every time that we want to add properties or define new channels, sources or sinks to the agent *a1* we will use a dot after it. If we would want to create another agent we just would need to add one line following the same structure than before (i.e *a2.channels = c2* and so on).

Well this is the standard part of the configuration. And we still need to add a sink to complete our agent. The typical sink to process streaming data is the *Avro Sink*, but since we are using the second approach of Flume and Spark integration we will use a special one. In order to do that we need to give a name to the sink. In this case we gave the name of *spark* to the sink.

Once we have named the sink we need to associate which type of sink are we using. This correspond to line 8:

```
a1.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink
```

In the following lines (9, 10 and 11) we are giving more functionalities to the Spark sink like setting the name of the host, the port used and the type of channel.

Lines from 14 to 18 and 21 to 24 treats with the configuration of the source and the channel. And finally lines from 27 and 29 are the responsive of the binding of the source and the sink with the channel.

The last step in order to work with Flume is how to launch it. The figure below shows the line used in order to launch the configuration file used before as example.

```

└─ Launch agent
/home/smhernan/apache-flume-1.5.2-bin/bin/flume-ng agent \
--classpath /home/ubuntu/spark-1.3.0-bin-hadoop2.3/lib/spark-assembly-1.3.0-hadoop2.3.0.jar:/home/smhernan/apache-flume-1.5.2-
bin/plugins.d/spark/lib/spark-streaming-flume-sink_2.10-1.3.0.jar:/home/smhernan/apache-flume-1.5.2-bin/plugins.d/spark/lib/scala-library-
2.10.4.jar:/opt/cloudera/parcels/CDH-5.1.3-1.cdh5.1.3.p0.12/lib/hbase/hbase-hadoop2-compat.jar:/opt/cloudera/parcels/CDH-5.1.3-
1.cdh5.1.3.p0.12/lib/hbase/hbase-client.jar:/opt/cloudera/parcels/CDH-5.1.3-1.cdh5.1.3.p0.12/lib/hbase/hbase-server.jar:/opt/cloudera/parcels/CDH-
5.1.3-1.cdh5.1.3.p0.12/lib/hbase/hbase-common.jar:/opt/cloudera/parcels/CDH-5.1.3-1.cdh5.1.3.p0.12/lib/hbase/lib/htrace-
core.jar:/opt/cloudera/parcels/CDH-5.1.3-1.cdh5.1.3.p0.12/lib/hbase/lib/guava-12.0.1.jar \
--conf /home/smhernan/apache-flume-1.5.2-bin/conf \
--conf-file /home/smhernan/apache-flume-1.5.2-bin/conf/test2.conf \
--name a1 └─ Name agent (Here we can add how many agents we want, and they will be received by the conf file.
-Dflume.root.logger=INFO,console
└─ Configuration file used
└─ Here uses some files inside conf folder
└─ Classpath with jars necessary in order to use Spark Sink

```

## C.2 Spark and HBase integration guide

As with Flume, HBase was already installed on the cluster IRIDIA, so we had to do the deployment on it. So we only had to focus on the integration of HBase with Spark.

At first we were surprised that there was not an implementation of the API Scala in HBase whereas in Java it does. But then we realized about the property of interoperation between Scala and Java so that the implementation finally was possible. So now we proceed to detail the steps to follow in order to work with HBase and Spark together.

### C.2.1 Procedure

The procedure of this integration is not very difficult, but it is still a main part of this thesis. That is why we include it.

#### Include

First we need to include the libraries needed in order to work with HBase. The libraries needed are:

```
1 import org.apache.hadoop.hbase.{HBaseConfiguration, HTableDescriptor}
2 import org.apache.hadoop.hbase.client.HBaseAdmin
3 import org.apache.hadoop.hbase.mapreduce.TableInputFormat
4 import org.apache.hadoop.fs.Path;
5 import org.apache.hadoop.hbase.HColumnDescriptor
6 import org.apache.hadoop.hbase.util.Bytes
7 import org.apache.hadoop.hbase.client.Put;
8 import org.apache.hadoop.hbase.client.HTable;
9 import unirest.spark.hbase._
```

After that we are able to use HBase tables inside our code. The steps to follow in order to use one HBase table are:

```
1 // Add local HBase conf
2 val conf = HBaseConfiguration.create()
3
4 // Set configuration
5 conf.set(TableInputFormat.INPUT_TABLE, <table_name>)
6
7 //Opening Table
8 val table = new HTable(conf, <table_name>)
```

With the above code we are creating a new HBase table called *table*. In this case the API search inside the tables available in HBase and takes the one with the name *<table\_name>*.

## Relevant operations in HBase tables

Now that we have created a HBase table is time to play with it. The most used operations in this thesis are: *put* and *get*. Let's see how to use them and what are they doing.

**PUT:**

```
1 //Create Put Object
2 val p = new Put(new String("Row Key").getBytes())
3
4 //Filling Put object
5 p.add("Column Family".getBytes(), "Column".getBytes(), new
6 String("Hello").getBytes())
7
8 //Putting data into the table: table
9 table.put(p)
```

**GET:**

```
1 // Instantiating Get class
2 val g = new Get(Bytes.toBytes("Row Key"))
3
4 // Reading the data
5 val result = table.get(g)
6
7 // Reading values from Result class object
8 val value = result.getValue(Bytes.toBytes("Column Family"), Bytes.toBytes("
9 Column"));
```

In both operations (Put and Get) we see that there are three steps to follow.

1. In one hand create an instance of their respective classes.
2. Once we instantiated the variable we can Put or Get a variable.
3. Finally we will be able to read or extract data from the HBase tables.

# Appendix D

## Application code

```
1 import scala.collection.mutable.ArrayBuffer
2 import org.apache.spark.storage.StorageLevel
3 import org.apache.spark.streaming._
4 import org.apache.spark.streaming.flume._
5 import org.apache.spark.util.IntParam
6 import java.net.InetSocketAddress
7 import org.apache.spark.mllib.tree.RandomForest
8 import org.apache.spark.mllib.util.MLUtils
9 import org.apache.spark.{ SparkConf, SparkContext }
10 import org.apache.spark.mllib.tree.model.RandomForestModel
11 import org.apache.spark.mllib.regression.LabeledPoint
12 import org.apache.spark.mllib.linalg.{ SparseVector, DenseVector, Vector,
    Vectors }
13 import org.apache.spark.rdd.NewHadoopRDD
14 import org.apache.hadoop.hbase.{ HBaseConfiguration, HTableDescriptor }
15 import org.apache.hadoop.hbase.client.HBaseAdmin
16 import org.apache.hadoop.hbase.mapreduce.TableInputFormat
17 import org.apache.hadoop.fs.Path;
18 import org.apache.hadoop.hbase.HCColumnDescriptor
19 import org.apache.hadoop.hbase.util.Bytes
20 import org.apache.hadoop.hbase.client.Put;
21 import org.apache.hadoop.hbase.client.HTable;
22 import org.apache.spark.AccumulatorParam
23 import org.apache.spark.rdd.RDD
24 import org.apache.log4j.Logger
25 import org.apache.log4j.Level
26
27 //
28 //
29 // Application that determines in real time if a transaction is fraud or not.
30 //
31 // This should be used in conjunction with the Spark Sink running in a Flume
   agent. See
32 // the Spark Streaming programming guide for more details.
33 //
```



```
34 // Usage: SparkTxFraudDetector <host> <port>
35 //   'host' is the host on which the Spark Sink is running.
36 //   'port' is the port at which the Spark Sink is listening.
37 //
38 // To run this App:
39 // '$ ./bin/spark-submit --class SparkTxFraudDetector --driver-class-path
   spark-streaming-flume_2.10-1.3.0.jar:/home/smhernan/apache-flume-1.5.2-bin/
   spark-streaming-flume-sink_2.10-1.3.0.jar:/home/smhernan/apache-flume
   -1.5.2-bin/lib/*:/opt/cloudera/parcels/CDH-5.1.3-1.cdh5.1.3.p0.12/lib/hbase
   /hbase-server.jar:/opt/cloudera/parcels/CDH-5.1.3-1.cdh5.1.3.p0.12/lib/
   hbase/hbase-protocol.jar:/opt/cloudera/parcels/CDH-5.1.3-1.cdh5.1.3.p0.12/
   lib/hbase/hbase-hadoop2-compat.jar:/opt/cloudera/parcels/CDH-5.1.3-1.cdh5
   .1.3.p0.12/lib/hbase/hbase-client.jar:/opt/cloudera/parcels/CDH-5.1.3-1.
   cdh5.1.3.p0.12/lib/hbase/hbase-common.jar:/opt/cloudera/parcels/CDH
   -5.1.3-1.cdh5.1.3.p0.12/lib/hbase/lib/htrace-core.jar:/opt/cloudera/parcels
   /CDH-5.1.3-1.cdh5.1.3.p0.12/lib/hbase/lib/guava-12.0.1.jar --master local
   [*] /home/smhernan/spark-1.3.0-bin-hadoop2.3/Project/flumePollingEventCount
   /target/scala-2.10/flumepollingeventcount_2.10-1.0.jar [host] [port] '
40 //
41 //
42 //
   //////////////////////////////////////
43
44 object SparkTxFraudDetector extends Serializable {
45
46   def main(args: Array[String]) {
47
48     //Setting variables that will be shown in terminal
49     Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
50     Logger.getLogger("org.eclipse.jetty.server").setLevel(Level.OFF)
51
52     //Can not use more than 4 input variables
53     if (args.length < 4) {
54       System.err.println("Usage: SparkFraudDetector <host> <port> <batch
         interval> <Random Forest model path>")
55       System.exit(1)
56     }
57
58     //Dictionary used to know if a feature is used and if it is categorical or
       not
59     val featuresDic = Map(
60
61       0 -> ("dateTime", "Not Used"),
62       1 -> ("MIN_AMT_HIS", "Integer"),
63       2 -> ("SUM_AMT_HIS", "Integer"),
64       3 -> ("NB_REFUSED_HIS", "Integer"),
65       4 -> ("LAST_COUNTRY_HIS", "Categorical"),
66       5 -> ("NB_TRX_SAME_SHOP_HIS", "Integer"),
67       6 -> ("NB_TRX_HIS", "Integer"),
68       7 -> ("TERM_MCC", "Integer"),
69       8 -> ("TERM_COUNTRY", "Categorical"),
70       9 -> ("amount", "Integer"),
```

```
71     10 -> ("TX_ACCEPTED", "Categorical"),
72     11 -> ("TX_3D_SECURE", "Categorical"),
73     12 -> ("AGE", "Integer"),
74     13 -> ("TX_HOUR", "Integer"),
75     14 -> ("IS_NIGHT", "Categorical"),
76     15 -> ("HAD_TEST", "Categorical"),
77     16 -> ("HAD_REFUSED", "Categorical"),
78     17 -> ("HAD_TRX_SAME_SHOP", "Categorical"),
79     18 -> ("D_AGE", "Categorical"),
80     19 -> ("D_AMT", "Categorical"),
81     20 -> ("D_SUM_AMT", "Categorical"),
82     21 -> ("HAD_LOT_NB_TX", "Categorical"),
83     22 -> ("TX_INTL", "Categorical"),
84     23 -> ("TX_ATM", "Categorical"),
85     24 -> ("TERM_REGION", "Categorical"),
86     25 -> ("TERM_CONTINENT", "Categorical"),
87     26 -> ("TERM_MCCG", "Categorical"),
88     27 -> ("TERM_MCC_GROUP", "Not Used"),
89     28 -> ("subject", "Not Used"),
90     29 -> ("LAST_MIDUID_HIS", "Not Used"),
91     30 -> ("TERM_MIDUID", "Not Used"),
92     31 -> ("LANGUAGE", "Categorical"),
93     32 -> ("GENDER", "Categorical"),
94     33 -> ("BROKER", "Categorical"),
95     34 -> ("CARD_BRAND", "Categorical"),
96     35 -> ("RISK_LAST_COUNTRY_HIS", "Integer"),
97     36 -> ("RISK_TERM_MCC", "Integer"),
98     37 -> ("RISK_TERM_COUNTRY", "Integer"),
99     38 -> ("RISK_D_AGE", "Integer"),
100    39 -> ("RISK_D_AMT", "Integer"),
101    40 -> ("RISK_D_SUM_AMT", "Integer"),
102    41 -> ("RISK_TERM_REGION", "Integer"),
103    42 -> ("RISK_TERM_CONTINENT", "Integer"),
104    43 -> ("RISK_TERM_MCCG", "Integer"),
105    44 -> ("RISK_TERM_MCC_GROUP", "Integer"),
106    45 -> ("RISK_LAST_MIDUID_HIS", "Integer"),
107    46 -> ("RISK_TERM_MIDUID", "Integer"),
108    47 -> ("RISK_LANGUAGE", "Integer"),
109    48 -> ("RISK_GENDER", "Integer"),
110    49 -> ("RISK_BROKER", "Integer"),
111    50 -> ("RISK_CARD_BRAND", "Integer"))
112
113    //Set the node and the port into array
114    val host = args(0)
115    val port = args(1).toInt
116
117    //Set batch interval (RDD streaming duration)
118    val batchSize = args(2).toInt
119
120    //Get timestamp model
121    val modelPath = args(3)
122    val modelPathSplited = modelPath.split("-")
123    val timestampModel = modelPathSplited(2).toLong
```

```
124     val batchSize = Milliseconds(batchTime)
125
126     //Dictionary Path
127     val dictionaryPath = "hdfs://node001/user/smhernan/DStreams/
        categories_table/categories_table.csv"
128
129     //HBase table name
130     val tableName = "smhernan_features_full"
131
132     // Create the context and set the batch size
133     val sparkConf = new SparkConf().setAppName("SparkTxFraudDetector")
134     val sc = new SparkContext(sparkConf)
135     val ssc = new StreamingContext(sc, batchSize)
136
137     // Create a flume stream that polls the Spark Sink running in a Flume agent
138     val addresses = Seq(new InetSocketAddress(host, port))
139     val storageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
140     val batchSize = 5
141     val parallelism = 5
142
143     // Create DStream
144     val stream = FlumeUtils.createPollingStream(ssc, addresses, storageLevel,
        batchSize, parallelism)
145     val lines = stream.map(record => new String(record.event.getBody().array(),
        "UTF-8"))
146
147     //Split transaction by ";"
148     val transactions = lines.map(line => line.split(";")).map(entry => Array(
        entry(0), entry(1), entry(2), entry(3), entry(4), entry(5), entry(6),
        entry(7), entry(8), entry(9), entry(10), entry(11), entry(12), entry(13)
        , entry(14), entry(15), entry(16), entry(17), entry(18), entry(19),
        entry(20), entry(21), entry(22), entry(23), entry(24), entry(25), entry
        (26), entry(27), entry(28), entry(29), entry(30), entry(31), entry(32),
        entry(33), entry(34), entry(35), entry(36), entry(37), entry(38), entry
        (39), entry(40), entry(41), entry(42), entry(43), entry(44), entry(45),
        entry(46), entry(47), entry(48), entry(49), entry(50)))
149
150     //Create the dictionary through categories table. We will have 4 columns:
        Feature name, key, value and timestamp
151     val dictionary = sc.textFile(dictionaryPath).map(line => line.split(";")).
        map(entry => Array(entry(0), entry(1), entry(2), entry(3))).map(elem =>
        (elem(0), elem(1), elem(2), elem(3)))
152     //Dictionary as a array in order to use it inside Map function (otherwise
        we have an error of nested RDD)
153     val dictionaryArray = dictionary.collect
154     val dictionaryQuotes = sc.textFile(dictionaryPath)
155
156     //Load Random Forest model
157     val model = RandomForestModel.load(sc,modelPath)
158
159     //Variables used to plot the results of our thesis
160     //Lines sent by our script
161     val linesSend = new ArrayBuffer[Int]()
```

```

162 //Count the lines in HBase
163 val countHbase = new ArrayBuffer[Int]()
164 //Timestamp used to calculate the time axis
165 val linesTimestamp = new ArrayBuffer[Long]()
166
167 val out1 = new ArrayBuffer[Array[String]]()
168 val new_dictionary = Array("0", ("0", "0", "0", "0"))
169
170 //Transactions is a DSTREAM which contains a RDD with transactions per line
171 transactions.foreachRDD { rdd =>
172 //Using foreachRDD implies that what are we going to process will be done
173 //in each RDD
174 //We count the amount of lines sent in one batch
175 linesSend += rdd.count().toInt
176 //foreach2
177 //Here we are accessing inside each RDD (we are reading each transaction
178 //inside the rdd)
179 val new_dictionary = rdd.map { line =>
180 //Lines is something like: [(feature1, feature2, ..., feature N),(
181 //feature1, feature2, ..., feature N),..., (feature1, feature2, ...,
182 //feature N)]
183 //In this map function we are going to apply all this operations on
184 //each line
185
186 //HBASE configuration
187 implicit val conf = HBaseConfiguration.create()
188 conf.addResource(new Path("/opt/cloudera/parcels/CDH-5.1.3-1.cd5.1.3.
189 //p0.12/etc/hbase/conf.dist/hbase-site.xml"))
190 conf.addResource(new Path("/etc/hbase/conf.cloudera.hbase/core-site.xml
191 //"))
192
193 // Add local HBase conf
194 conf.set(TableInputFormat.INPUT_TABLE, tableName)
195
196 // Create table instance
197 val myTable = new HTable(conf, tableName)
198
199 //Getting transaction and associate it to the dictionary
200 val lineWithIndex = line.zipWithIndex //Add index to each feature [((
201 //feature0,0) , (feature1,1), ..., (feature N,N)),((feature0,0) , (
202 //feature1,1), ..., (feature N,N)),...]
203 val lineID = lineWithIndex.map(value => (value._1 -> featuresDic(value.
204 //_2))) // [(feature0,("dateTIme","Not used")), (feature1,("AMS_MIN_INT
205 //","Integer")),...]
206 val timestamp: Long = System.currentTimeMillis / 1000
207 val row = timestamp.toString + "|" + line(28) //The row key is the
208 //timestamp + subject
209
210 //Save the timestamp to hbase for each transaction
211 val p = new Put(new String("stream " + row).getBytes())
212 p.add("timestamp".getBytes, "Not used".getBytes, timestamp.toString.
213 //getBytes()) // (column family, column, value)
214 myTable.put(p)

```

```

202
203 //Finally we will perform the map function to every feature
204 val new_dictionary = lineID.map { feature =>
205     //Decomposing features
206     val value = feature._1.trim //Getting first field of (feature0,("
    dateTime","Not used")) => feature0 and delete whitespaces
207     val columnFamily = feature._2._1 //Getting first field of second field
        of (feature0,("dateTime","Not used")) => "dateTime"
208     val column = feature._2._2 //Getting second field of second field of (
        feature0,("dateTime","Not used")) => "Not used"
209
210     p.add(columnFamily.getBytes, column.getBytes, value.getBytes()) // (
        column family, column, value)
211     myTable.put(p)
212
213     //We will return value to insert to the model,
214     var out = value
215     //We will use this variable in order to check if we have new entries
216     var new_dictionary_entry = ("0", "0", "0", "0")
217
218     //Check if categorical and is not a digit
219     if (column == "Categorical" && !value.matches("( )?\\d+(\\.\\d+)?(\\s$
        )?")) {
220         //Array that contains the result of filter by the FeatureName and
            its value. Keep in mind that is an array
221         val filtered_dic = dictionaryArray.filter(elem => (elem._1 ==
            columnFamily && elem._3 == value))
222
223         //Case not empty head to get first element of the array
224         if (!filtered_dic.isEmpty) {
225             if (filtered_dic.map(i => i._4).head.toDouble > timestampModel.
                toDouble) {
226                 out = "0"
227             } else {
228                 out = filtered_dic.map(i => i._2).head
229             }
230
231             //If does not exist
232         } else {
233             //First check if the label exist
234             out = "0" //If does not exist is not even necessary to check the
                timestamp, we always will use 0
235             val filtered_dic_label = dictionaryArray.filter(elem => (elem._1 =
                = columnFamily))
236             //If the label already exist, create a new entry to the dictionary
                adding + 1
237             if (!filtered_dic_label.isEmpty) {
238                 val new_value = filtered_dic_label.map(i => i._2.toInt).max + 1
239                 //Actualize dictionary in memory
240                 new_dictionary_entry = ((columnFamily.toString, new_value.
                    toString, value.toString, timestamp.toString))
241                 val new_dictionary_entry_array = Array((columnFamily.toString,
                    new_value.toString, value.toString, timestamp.toString))

```

```
242         dictionaryArray.union(new_dictionary_entry_array)
243
244         //If the label does not exist create new Category
245     } else {
246         val new_value = 1
247         //Actualize dictionary in memory
248         new_dictionary_entry = (columnFamily.toString, new_value.
                toString, value.toString, timestamp.toString)
249         val new_dictionary_entry_array = Array((columnFamily.toString,
                new_value.toString, value.toString, timestamp.toString))
250         dictionaryArray.union(new_dictionary_entry_array)
251
252     }
253 }
254
255 }
256
257 (out, new_dictionary_entry)
258
259 }
260
261 //Use model to get a prediction of our vector then add to HBase in
    order to see the accuracy
262 val prediction = model.predict(new_dictionary._1)
263 p.add("P(fraud)", "Integer", prediction.getBytes())
264 myTable.put(p)
265
266 new_dictionary
267
268 }
269
270 //Getting variables from the map function
271 val output = new_dictionary.map(i => i.map(i => i._1))
272 val new_entries = new_dictionary.map(i => i.map(i => i._2))
273 val subset_new_dic = new_entries.flatMap(i => i.filter(i => i._1 != "0"))
274
275 //If is not empty means that we have a new entry. Thus, we must update
    the dictionary in HDFS
276 if (!subset_new_dic.isEmpty) {
277     dictionary.union(subset_new_dic)
278     dictionary.saveAsTextFile("hdfs://node001/user/smhernan/DStreams/
        categories_table/categories_table")
279 }
280
281 implicit val conf = HBaseConfiguration.create()
282
283 // Add local HBase conf
284 conf.set(TableInputFormat.INPUT_TABLE, tableName)
285
286 val hBaseRDD = sc.newAPIHadoopRDD(conf, classOf[TableInputFormat],
    classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],
    classOf[org.apache.hadoop.hbase.client.Result])
287
288
289
```

```
290
291     countHbase += hBaseRDD.count().toInt
292     linesTimestamp += System.currentTimeMillis
293     //Display variables in order to plot results
294     println("Lines: " + linesSend.mkString(";"))
295     println("Counted: " + countHbase.mkString(";"))
296     println("Timestamp: " + linesTimestamp.mkString(";"))
297
298
299     } //foreachRDD
300     //streamSize.print()
301
302     //Straming context starts
303     ssc.start()
304     ssc.awaitTermination()
305
306 }
307 }
```