View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Universitatea Politehnică Bucureşti
Facultatea de Automatică şi Calculatoare

Master Thesis

Performance evaluation of Apache Mahout for mining large datasets

Advisors:
prof. dr. ing. Fatox Xhafa
prof. dr. ing. Florin Rădulescu

Student:
Bogza A.G. Adriana-Maria

2016

Thesis information

Title	Performance evaluation of Apache Mahout for mining large datasets
Author	ADRIANA MARIA BOGZA
Date	4th of July of 2016
Advisor	FATOS XHAFA
Department	Computer Science
Co-advisor	FLORIN RADULESCU
Department	Computer Science
Institution	Automatic Control and Computer Science Faculty
	Politehnica University of Bucharest
Master	MASTER IN INNOVATION AND RESEARCH IN INFORMATICS
Faculty	FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
University	UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)
	BarcelonaTech

Members of the court

President JORGE CASTRO RABAL

Secretary GABRIEL ALEJANDRO VALIENTE FERUGLIO

Vocal FERNANDO MARTÍNEZ SÁEZ

Grading

Numerical grade

Descriptive grade

Date 4th of July of 2016

Acknowledgements

To prof. Fatos Xhafa for his continuos help and guidance throughout this project.

To prof. Florin Radulescu for his continuous help and guidance throughout the six year of undergraduate and post-graduate studies.

To Gabriel Verdejo Àlvarez and Iván Balañá Jiménez from the RDLab cluster for their dedication and help with setting up the infrastructure.

To my family and friends for supporting me.

To the people I've met in Barcelona that made this experience one of a kind.

Abstract

Large scale data processing is a subject of interest for many researchers, who approach the problem from various points of view, starting with necessary storage, models, computation, tools. The main purpose of this project is to evaluate the performance of the Apache Mahout library, that contains data mining algorithms for data processing. The datasets used for this evaluation are constructed with the help of the Twitter Streaming API. The environment on which we are evaluating the performance of Mahout is Hadoop MapReduce. In order to convert the Twitter Stream to batches of data for MapReduce a persistent storage layer needs to be included in the design of our system.

The performance is evaluated in terms of computation time, memory used and number of read and written bytes. The accuracy of the results is also evaluated in some of the scenarios.

Contents

1 Introduction **17**

1.1 Objectives 19

2 State of the art **21**

2.1 Big Data and distributed computing 22

2.1.1 Big Data 22

2.1.2 Distributed Computing 24

2.1.3 Apache Hadoop 25

2.1.4 Applicabilities of Apache Hadoop MapReduce for Big Data processing 27

2.2 Data mining, text mining 30

2.3 Batch vs. Stream processing 34

3 Architecture **35**

3.1 Infrastructure 36

3.2 Conceptual model 38

3.3 System architecture 40

4 Stream processing system **43**

4.1 Apache Yahoo! S4 model 45

4.2 Apache S4 vs. Apache Storm 47

4.3 Twitter Stream 49

4.4 Processing the Twitter stream with S4 50

4.4.1 Twitter-adapter 50
4.4.2 Twitter-processor 51

5 Dataset and persistence layer 55
5.1 Dataset 56
5.1.1 Data structure 56
5.1.2 Data size 59
5.1.3 Data receive rate 60
5.2 Persistence Layer 62
5.2.1 HBase 63
5.2.2 CassandraDB 66
5.2.3 MongoDB 68
5.2.4 Hadoop Distributed File System 69

6 Apache Mahout 73
6.1 Data preprocessing 75
6.2 Clustering 79
6.2.1 Centroid generation 79
6.2.2 Mahout kMeans 81
6.2.3 Mahout Fuzzy-kMeans 82

7 Experimental study 85
7.1 Parameters 87
7.2 Proof of concept 89
7.3 Experiment #1: Constant number of nodes, different data sizes 92
7.3.1 Normal datasets 92
7.3.2 Larger datasets 95
7.4 Experiment #2: Constant data size, different number of nodes 97
7.5 Experiment #3: I/O performance evaluation 99
7.6 Experiment #4: Algorithmic accuracy evaluation 101

8 Economic report: planning and costs **105**

8.1 Project planning 105

8.2 Costs 108

8.2.1 Development costs 108

8.2.2 Infrastructural costs 108

9 Conclusions **111**

Appendix 1 - User guide **113**

Bibliography **117**

List of Figures

2.1 Hadoop Architecture 26

3.1 Conceptual model 38
3.2 Enahnced conceptual model 39
3.3 System architecture 41

4.1 Yahoo S4 architecture 46
4.2 Storm processing model 47
4.3 S4 model for processing the Twitter Stream 50
4.4 Data flow in the Twitter Processor S4 APP 52

5.1 Tweet object structure in the S4 application 58
5.2 Changes in the data structure of a tweet 59
5.3 Receive rate of tweet events over 20hours window 61
5.4 HBase datamodel 65

6.1 Data conversion as part of the preprocessing step 78

7.1 Processing time 93
7.2 In-memory usage 94
7.3 kMeans versus fuzzy kMeans 95

8.1 Gantt project diagram 107
8.2 EC2 instances pricing. *Retrieved from: `https://aws.amazon.com/ec2/pricing/`* 110

List of Tables

1.1 Objectives 20

3.1 Cluster processors 37

5.1 HBase vs. Cassandra advantages 68

7.1 Experimental results - 4 nodes, 256MB - Processing time and in-memory usage 89

7.2 Experimental results - 4 nodes, 256MB - I/O operations . . . 89

7.3 Experimental results - 4 nodes, different data sizes - processing time 92

7.4 Experimental results - 4 nodes, different data sizes - in-memory usage 93

7.5 Centroid generation parameters - smaller versus larger datasets 96

7.6 Experimental results - 4 nodes, different large data sets sizes - processing time 96

7.7 Experimental results - 4 nodes, different large data sets sizes - in-memory usage 96

7.8 Experimental results - 160k tweets, different number of nodes - processing time 97

7.9 Experimental results - 160k tweets, different number of nodes - in-memory usage 97

7.10 Experimental results - 320k tweets, different number of nodes - processing time 98

7.11 Experimental results - 320k tweets, different number of nodes - in-memory usage 98
7.12 Experimental results - I/O operations - Data preprocessing . . 99
7.13 Experimental results - I/O operations - Centroid generation . 99
7.14 Experimental results - I/O operations - kMeans 99
7.15 Experimental results - I/O operations - Fuzzy kMeans 100

8.1 Development costs 108
8.2 Hardware costs 110

Chapter 1

Introduction

The quantity of available data worldwide is constantly increasing and so is the need of large datasets processing. It is no longer enough to be able to access data, because the quantity that is available today makes it close to impossible to store it in one place and retrieve relevant information from it.

Large scale data processing is crucial in multiple fields, like medicine, finances, retail industry, telecommunication, law enforcement, corporate surveillance, research analysis, marketing and others. Each one of these has gained valuable insights by mining available data. In this project the focus lies on social media analytics, more precisely on extracting information from public posts on Twitter. Social media analytics is important for retail industry, as it can provide insights about people's need and then they can be targeted with the exact products or services that meet their needs and expectations. Politics is another field that can benefit from it, gathering information about the opinions of society, popularity of a candidate or approval or disapproval of a particular event for example. Human resources departments can find adequate candidates for the available open jobs by following users interested by specific topics, such as computer science or bioengineering.

The infrastructure is an important aspect to consider when processing large datasets and extracting relevant information from them. First of all storage should be taken into account, where is the data stored and which is the format? The format is important because there is no universal storage format and data can be gathered from multiple sources in different structures and some of it may need some pre-processing step to reach a common form that can be used further on. Second, the processing time and required memory needed for the entire dataset is to be taken into consideration when architecturing the infrastructure. A distributed system for processing is likely needed,

as one machine can hardly manage processing for large dataset, given that processing algorithms are both CPU and in-memory intensive.

The type of information that is to be extracted from the dataset is another factor to be considered. Is the information time-critical? For example, in some contexts information needs to be extracted in real time in order to be able to act fastly based on that information. One example would be an Intrusion Detection System (IDS) which needs to be able to automatically detect in real-time any kind of attack over a network [1]. In other cases, the desired information is not that time-critical and extracting it in a few minutes or even hours would be acceptable, depending on the context.

The purpose of this project is to evaluate the performance of Apache Mahout on a large dataset. There are multiple frameworks for which Mahout offers support, one of them being Hadoop MapReduce, which is the one chosen for this project. The dataset used will be formed of tweets published on the Twitter social media platform that are exposed via the Twitter Streaming API [2]. The stream processing system used in this project for constructing the tweets dataset is Apache S4.

The study aims to provide valuable insights regarding the performance of the data mining algorithms implemented in the Mahout library with Hadoop MapReduce as an infrastructural support. It is important to underline that the performance of data mining algorithms can be very different from case to case, depending on the underlying framework, the particularities of the dataset or the physical infrastructure available. All combinations of these parameters cannot be covered in a study like this one. The Hadoop MapReduce framework was chosen because of its popularity in both enterprise and research communities. The dataset has been constructed from tweets because of their particularities as very small text documents and the endless stream that can be processed, which allows creating a dataset with a great number of entries. The number of tweets published each second is estimated to be around 5k, which leads to over 500M tweets published each day [3]. Even if the Twitter Streaming API exposes only 1% of the published tweets and not all of them are in English, which is the language of interest for this project, millions of tweets can be gathered each day and that makes it a great source of data for the large dataset required for this project.

In this paper we've discussed about the motivation of this project so far. In the following section of this chapter (1.1) we will continue to discuss the objectives of this project. Chapters 2-7 are about the technical aspects of the project. In Chapter 2 we discuss about the state of the art of this problem and we consider Big Data, distributed computing, data mining techniques

and batch and stream processing some of the keywords. The architecture designed for the objectives of this paper is presented in Chapter 3 and also the physical infrastructure that was made available by the RDLab of UPC is described. The streaming processing system model, possible solutions and actual implementation are discussed in Chapter 4. In Chapter 5 the particularities of the dataset that is constructed from the Twitter Streaming API are presented and some options for the persistence layer that is to store the tweets are evaluated. Chapter 6 is about the Mahout library, the support if offers for some data preprocessing steps and the algorithms that are evaluated within this project. In Chapter 7 the experiments that were conducted and the obtained performance results are presented. The performance is to be evaluated on three axis, time performance, memory performance and algorithmic accuracy. In Chapter 8 the main tasks and their distribution across time are presented and also the identified costs for development and infrastructure are detailed. In Chapter 9 the conclusions of this study are drawn and the objectives accomplishments are evaluated.

1.1 Objectives

The main objective of this project is to evaluate the performance of Apache Mahout for mining large datasets, that consist of tweets gathered with a stream processing system, like Apache S4. With this goal in mind, there are several other objectives that are to be completed at the end of this project and that are described next.

O1: Setting up the Hadoop environment on the machines given inside the RDLab cluster, with all the configurations needed for running Mahout MapReduce algorithms on top of Hadoop

O2: Construct the dataset: Adopt Yahoo S4 for the Twitter stream processing, develop the S4 applications needed in order to gather the required data and evaluate which information is useful for data mining algorithms and should be passed on to the Mahout algorithms.

O3: Analyze and choose a suitable persistence layer in order to connect the S4 stream processing with the Mahout MapReduce batch processing algorithms; NoSQL DB solutions should be evaluated, but also files on the Hadoop File Systems are to be taken into consideration.

Table 1.1: Objectives

Objective	Description
Main	**Evaluate the performance of Apache Mahout for mining large datasets**
O1	Setup the Hadoop environment
O2	Construct the dataset: Adopt Yahoo S4 for the Twitter stream processing
O3	Analyze and choose suitable persistence layer
O4	Analyze dataset with Mahout algorithms
O5	Evaluate the performance
O6	Self-development a. Learning new technologies: S4, Mahout b. Mastering latex long document writing

O4: Analyze the dataset using Mahout algorithms and evaluate the accuracy of the results considering the given dataset; this objective to answer simple questions: are Mahout clustering algorithms suited for analyzing the content published on Twitter? Is it possible to determine the popular topics among the given tweets?

O5: Evaluate the performance of Mahout algorithms on the given dataset; the performance evaluation should target two main directions: High Performance Computing evaluation and I/O performance evaluation.

O6: Self development on two axis: studying and learning new technologies, related to the project, like S4 and Mahout, and mastering latex and long document writing

Chapter 2

State of the art

Taking into consideration the rapid growth in volume regarding data that is available from different sources (online content, sensor networks), the re-evaluation of data storage and manipulation techniques was a necessary step for further development. Standard database management systems became insufficient. A distributed storage system is a viable solution for a large volume of data that must be analyzed. Such a model offers scalability and allows distributing the computation resources across multiple nodes inside a cluster, which leads to a smaller response time.

In this chapter we will discuss further about big data, storage solutions and high performance computing. We will continue with techniques for extracting relevant informations from a big volume of data, which fall under the data mining umbrella. We will focus our attention to text mining, since this is within the scope of this project. At last, we will discuss two approaches for data processing, batch processing and stream processing and we will provide a short comparison between them.

2.1 Big Data and distributed computing

Big Data became a trending term with the exponential growth of the quantity of available data and the need to quickly extract relevant information from it. It can help businesses make better decisions, discover patterns, enhance their productivity and improve their quality. It gives business the ability to be more agile and adapt based on current trends and estimated future ones.

In this section we will discuss about what Big Data means, about types of distributed computing and we will present a tool for storing and processing large data sets in order to extract relevant information from it. In subsection 2.1.4 some applicabilities of this tool are given as example.

2.1.1 Big Data

Big Data is not a new concept, the first time the term was mentioned was in 1997 in a paper written by scientists at NASA [4] describing a problem with computer graphics. Over the years the term became more and more popular as it was adopted by industry dealing with huge quantities of data. There are multiple existing definitions for Big Data and none is official. In the Oxford English Dictionary Big Data is “data of a very large size, typically to the extent that its manipulation and management present significant logistical challenges”.

Other definition provided by Wikipedia is “an all-encompassing term for any collection of data sets so large and complex that it becomes difficult to process using on-hand data management tools or traditional data processing applications”. Even if variations of this definition are commonly used, terms like “large” or “traditional” are relative and ambiguous. McKinsey Global Institute offered another definition for Big Data in 2011, as “datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze” [5]. However the researchers agree that “this definition is intentionally subjective and incorporates a moving definition of how big a dataset needs to be in order to be considered big data” [5].

Gartner defines Big Data as “high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation” [6]. Most authors take into consideration these three dimensions, volume, velocity and variety, that is why Big Data is commonly defined by the three Vs. Some researchers tried to expand this numbers with other

relevant dimensions.

Volume refers to the amount of data, usually low-density and unstructured data, such as click streams on web or mobile apps, sensor networks or Twitter data feeds. The task of Big Data is to convert this into valuable information. The dimension of velocity takes into account the fast rate at which the data is received and potentially acted upon. All Big Data applications receive data real-time and some of them also have to give near real-time answers with relevant information extracted from that data. Variety derives from the fact that most of the data comes in an unstructured or semi-structured format. Text for example can be considered an unstructured format, from which one has to extract valuable structured information.

Key players from the IT industry developed some of the Big Data technologies and have also given their own definitions for Big Data. Oracle asserts that "Big data describes a holistic information management strategy that includes and integrates many new types of data and data management alongside traditional data" [7]. Four dimensions are taken into consideration: volume, velocity and variety, as the Gartner definition and the fourth one is value. Data has an intrinsic value that must be discovered, for example consumer preferences or sentiments with regards to a particular subject. Besides the technologies needed for storing such a high volume of data and processing it for information extraction, it is argued that another challenge is related to humans learning to ask the right questions.

Microsoft defines Big Data as "the term increasingly used to describe the process of applying serious computing power—the latest in machine learning and artificial intelligence—to seriously massive and often highly complex sets of information" [8]. A study conducted by Jonathan Stuart Ward and Adam Barker gives more definitions on Big Data gathered from multiple sources [9].

Together with the volume, velocity and variety of data great challenges arise and the necessity of high-quality IT infrastructures, platforms and DBs is highlighted. The authors of "High-Perfomance Big-Data Analytics. Computing Systems and Approaches" [10] identify some of the most important challenges as following: infrastructural challenges ("compute, storage and network elements for data capture, transmission, cleansing, storage, pre-processing, management and knowledge dissemination"; clusters, grids, clouds), platform challenges (in order to obtain "end-to-end, easy-to-use and fully integrated platforms for making sense out of big data") and file system and database challenges (addressed by analytical, scalable, distributed and parallel SQL databases, but also by NoSQL or NewSQL DBs, that are more

capable for handling big data).

2.1.2 Distributed Computing

Distributed computing assumes there is a computational system in which multiple interconnected computers share computational tasks assigned to the execution system. There are several distributed computing approaches, such as grid, cluster and cloud computing, we will discuss each of these briefly.

Cluster computing assumes there is a set of loose or tightly coupled computers that work together and each node is set to perform the same task. In most cases, each node uses the same commodity hardware and operating system (however this is not a strong requirement) and is connected with the rest of the cluster via high speed interconnection technologies, such as Gigabit Ethernet or InfiniBand.

Clusters ensure "scalability, availability and sustainability in addition to the original goal of higher performance" [10], as it has a modular architecture made up from basic and simple components, which makes it easy to add new nodes to the cluster to support higher throughput or to replace nodes that become unavailable for example. This type of architecture is horizontally scalable, because adding new processing nodes ensures availability for a higher throughput. Vertical scalability would increase capacity by adding more resources to the existing nodes, such as memory or CPU.

The tradeoffs that comes with this type of computing model (instead of using a supercomputer) is the increased management complexity between components. Load balancing for effective use of resources is a priority in such cases. In the same time, since commodity hardware is used and failures are expected, the software managing the cluster needs to be able to detect and respond to failures, which can bring an increase in complexity.

Grid computing consists of a collection of computers from multiple locations with a common goal. In a grid computing environment each node is set out to perform a different task and is loosely connected with the rest of the nodes.

One clear difference between clusters and grids is that usually in a cluster resources (nodes) are located in the same data center, while in a grid machines can be distributed across a geography. The tradeoff that comes with this for a grid application is that it has to consider the latency and bandwidth tolerance if it requires a geographically located resources.

A similarity between clusters and grids is that resource failures are commons

in both cases. In a grid the resource managers "must adapt their behaviour dynamically and use the available resources and services efficiently and effectively." [10]

Cloud computing is also known as on-demand computing as it based on services that rely on a pay-per-use model. There are three main types of services, Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). The idea behind cluster computing is that it uses a pool of configurable resources that can be rapidly provisioned and released, with minimal management effort.

2.1.3 Apache Hadoop

Hadoop is an open-source Apache software framework used for distributed processing of large datasets across large clusters. It is used for extensive data analytics and high performance computing. Hadoop at its core is made up of a distributed file system (Hadoop Distributed File System - HDFS), a processing framework called MapReduce and a job scheduling and resource manager framework, Hadoop YARN.

The Hadoop Distributed File System is designed to run over commodity hardware and ensures high fault tolerance and high throughput access at low-cost. It's architecture is based on the master-slave model. The master server is called the NameNode, while the slaves are DataNodes. The NameNode is the central machine of the cluster and it contains the file system metadata. Each DataNode manages the data stored on them and the computations over that data are made locally, in order to avoid data movement across the network. Usually files are divided in chunks of 64 to 512 MB (configurable) and the chunks are replicated in order to avoid data loss caused by system or network failures.

MapReduce is a programming model that it designed to be executed in parallel on large clusters over large datasets. The model is made up from two main functions: map and reduce. The map step can be seen as a preprocessing data step, while the reduce step can be associated with the actual computation and aggregation of results. Multiple map or reduce jobs are launched in parallel across multiple nodes in the cluster and each node receives a chunk of the input data to process, usually the one that is actually stored on that node. MapReduce follows the master-slave architecture also. The master node in this context is called the ResourceManager and it manages the existing jobs. Once a job is submitted, it is divided into tasks and the ResourceManager decides where to run each task and ensures continuous communication with

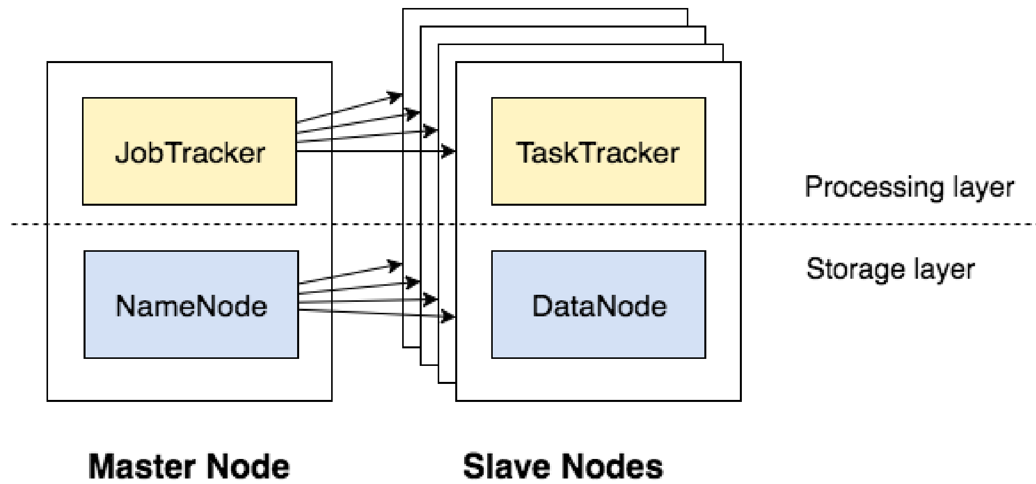


Figure 2.1: Hadoop Architecture

each NodeManager. A NodeManager is associated with the slave and monitors the execution of the tasks on that node (there can be multiple tasks per node) and sends continuous feedback to the ResourceManager.

The Hadoop architecture can be observed in figure 2.1.

Hadoop is designed to be fault tolerant, which means that even if some of the nodes fail, there should be no data loss. This is possible through data replication across DataNodes and through task, job and nodes management. If a task fails, the NodeManager detects the failure, sends a message to the ResourceManager, which later reschedules the task. If an entire DataNode fails, the NameNode and the ResourceManager detect the failure, all the tasks that were running on that node are rescheduled and the data is already replicated on other nodes. The NameNode and the ResourceManager are single point of failures in this architecture, if the master fails all the cluster becomes unavailable, even if the slaves are running.

There are multiple file systems that Hadoop can integrate with. These do not necessarily replace HDFS, but can be a source of data for Hadoop or a destination for Hadoop MapReduce jobs for example. One of them is FTPFS (File Transfer Protocol File Systems) which is a file system that supports access to a FTP server through standard file system APIs. Another example would be Amazon S3 (Amazon Simple Storage Server), which is mostly targeted for Hadoop clusters who are kept on Amazon EC2 (Amazon Elastic Compute Cloud) infrastructure.

Hadoop was designed to be deployed inside a cluster. Such clusters can be

physically located in an on-site datacenter, but can also be deployed in a cloud infrastructure. There are multiple cloud vendors who offer the possibility of deploying a Hadoop cluster without having to acquire any hardware or needing specific setup expertise, like Amazon, Microsoft and Google.

2.1.4 Applicabilities of Apache Hadoop MapReduce for Big Data processing

The need to handle, store and process large dataset is quickly expanding to many sectors, not just to IT-related fields. For example, the amount of patient data gathered over the years can lead in discovering better treatments in the health sector, but it is close to impossible for this data to be processed by human power only.

Hadoop is a commonly used framework for data mining and Big Data processing. There are many different fields in which the usage of Hadoop brings a considerable contribution in extracting relevant information from large dataset. We will discuss some of them briefly.

Log processing

The processes that govern our world are getting more computerized day by day. There are many software applications that replaced human actions. Among many other advantages, like speed of completing a task or easiness of interaction, software application offer the possibility of tracking the actions completed through logs. The problem that arises is that there is a lot of log data generated and it is difficult to extract relevant information from it. Hadoop can be used for large log processing. In the thesis of Daniel Llorente, named "Procesamiento masivo de datos via Hadoop" [11], some experiments of log file analysis with MapReduce jobs were conducted and the results are promising even for large log files (4GB).

Healthcare and Bioinformatics

An article from 2011 [12] estimated that the volume of medical data available worldwide provided by PACS (Picture Archiving and Communication Systems) vendors is around 150 Exabytes and is increasing at an approximate rate of x1.8 each year. PACS are only one of the sources of medical data, that contains different types of scans as images, such as ultrasounds,

magnetic resonance (MR), computed tomography (CT), endoscopy and others. Besides PACS there are a lot more medical data sources, like patient medical history or treatments evaluation. Given this high volume and the importance of medical research for human society, the need of processing Big Data in healthcare is critical.

Several experiments that made use of the Hadoop framework in order to extract relevant medical data where conducted. One example would be the usage of the MapReduce algorithms to identify unproven cancer treatments on the health web, a study of great importance regarding to dealing with the dissemination of false and dangerous information to vulnerable health consumers [13]. Another example would be DNA sequencing, with multiple applicabilities, such as studying “differences in one person’s genome relative to a reference human genome or comparing genomes of closely related species” [14]. CloudBurst is an algorithm for these types of studies that makes use of the MapReduce framework in order to parallelize computation and gather results. The experiment conducted proved that the time it takes to make the computation scales linearly with the number of nodes inside the Hadoop cluster. DNA fragment assembly algorithms have also been implemented using MapReduce with great results, as seen in the paper of Baomin Xu et. all [15].

Text Mining

The discovery of recurrent phrases in documents is an interesting research area in text mining and can be used for document summarization, clustering or topic search in a larger dataset. An experiment was conducted by A. Balkir et. all [16] in order to use the MapReduce framework for developing an algorithm to discover such recurrent phrases. The MapReduce solution proved to scale nicely for this experiment and to be fault tolerant, but the challenge was the maintainance of a large distributed table in HBase that needs to be frequently read by map jobs. The results proved that the MapReduce solution decreased the application runtime up to six times, than a naive distributed implementation over HBase.

Another example would be sentiment analysis on social media posts or spam detection of emails. In the experiments conducted by T. Cohn et. all [17] the MapReduce framework was used to tokenize and detect the language of twitter posts. The language detection especially is challenging in the case of twitter posts because of the small number of words in such a post (on average 10 words per tweet). E. Jain and S K Jain used a twitter dataset used Mahout

over Hadoop MapReduce to clusterize users based on the similarities of their posts [18]. They showed through their experiments how the execution time was bigger for the fuzzy kMeans than the kMeans algorithm, but the number of necessary iterations in order to obtain convergent clusters is higher for kMeans.

2.2 Data mining, text mining

There are several definitions given by different authors for data mining. Bing Liu associates it with knowledge discovery in databases (KDD) and defines it “as the process of discovering useful patterns or knowledge from data sources, e.g., databases, texts, images, the Web, etc. The patterns must be valid, potentially useful, and understandable.” [19]. It also states that data mining “is a multidisciplinary field involving machine learning, statistics, databases, artificial intelligence, information retrieval, and visualization”. Other authors list data mining as the analysis step in the KDD process - after data cleaning and transformation and before results visualization and evaluation.

The data mining process consists of several steps that need to be executed sequentially:

1. **Data collection** - Constructing the dataset by gathering data from existing databases or WWW
2. **Data preprocessing** - Preparing data before applying data mining algorithms to it; this includes:
   (a) *Data cleaning* - Implies iterating through the data set and removing missing values (or replacing them with default ones), smoothing noisy data, removing outliers and inconsistencies. This step is necessary because the data source can be without any restrictions enforced on the data. Imagine data comes from a web form with no required fields and no data validation - anyone can fill in any number of fields, and a missing field adds no value to the final results. Executing a data cleaning step leads to less missing values in the dataset.
   (b) *Data integration* - The integration of data from multiple sources, with possible different data types and structures, is sometimes necessary and also handling of duplicate or inconsistent data. Let's take the case of a dataset which is constructed from two different data sources and user profile is part of both. In the first data set the user has a date of birth associated with it, while in the second the user profile has a record for age. The age can be deducted from the date of birth, so in case of inconsistencies, in a data integration step, one may choose which source is the more accurate for a particular information and remove the inconsistencies.
   (c) *Data transformation* - The data transformation step can be executed in different forms, depending on the dataset at hand. Ex-

amples of actions that may be part of this step are: data normalization, summarizations, generalization or new attributes construction. One example would be when he have the date of birth for a user, but for the data mining algorithm the age is a more relevant attribute and we can derive this from its date of birth.

(d) *Data reduction* - It can be also found under the name of feature extraction. Not all attributes are relevant for the particular algorithm we want to apply. In order to reduce the size of data set and the processing time, only relevant attributes can be extracted and further processed. For example, for a user we might have a record for his first name, last name, middle name, gender, email address, telephone number, date of birth and a list of records of all of his shopping carts, where each item from the shopping cart is described by the name of the article, quantity, price and so on. Most likely, in order to build a recommendation system for shopping, the fields that are of interest for the data mining might be gender, age and the items he has purchased.

(e) *Data discretization* - This step is required for algorithms that can work only with discrete data, in this case the continuous attributes must be replaced with discrete ones from a limited set. If we were to take the age attribute and assign it a discrete value, that value could be young/middle-aged/old for example.

3. **Pattern discovery and extraction** - Applying the data mining algorithm in order to obtain some results.

4. **Visualization** - Necessary to visualize the results for a better understanding and evaluation. The visualization step can be applied to input data also.

5. **Evaluation of results** - In this step the human judgement has the responsibility to decide whether the results are valuable or not, whether the information extracted is useful and new or not. Some of the information extracted might be statistical truths for example.

There are two main data mining methods, predictive and descriptive. The predictive methods use some variables in order to predict the values of other variables. A classification method is a predictive method, which is based on labeled data in order to classify new data. The descriptive methods are applied in order to discover patterns that describe the inner structure of the dataset. One example would be the clustering algorithms that find groups

of similar object in the dataset and can identify the isolated objects, called outliers. For our project, we will use descriptive methods for finding the subjects of interest in our pool of tweets. The clustering algorithms that are taken into consideration are kMeans and Fuzzy kMeans, which is an extension of the first. These algorithms will be explained in detail in Chapter 6. The clustering algorithms we've mentioned can be applied on any type of input, be it points in a multi-dimensional space or text documents. The only requirement is to be able to define a distance function between items in the dataset in order to compute and evaluate if two items are close to each other (similar) or not.

In a similar manner as data mining, text mining can be defined as a "knowledge-intensive process in which a user interacts with a document collection over time by using a suite of analysis tools" and "seeks to extract useful information from data sources through the identification and exploration of interesting patterns" [20]. In the case of text mining, the data sources are document collections and patterns are found in unstructured text data. For this reason, the preprocessing step is essential in the text mining process, because it transforms the unstructured text data into a more explicitly structured format. The concepts the text mining operates with are document collection, document, character, word, term and concept.

A document collection is a grouping of text-based documents. These documents can be grouped by any criteria and usually text mining techniques aim to discover patterns across such collections. A document collection can be either static, if the set of documents doesn't change over time, or dynamic, if documents can be added or updated frequently in that collection. In our case, the document collection would be the collection of tweets we gather via the S4 application and it can be both static or dynamic, depending of the type of algorithm we are running against it. If we were to use regular clustering algorithms, the document collection would have to be static, that means that the stream of tweets received via S4 has to be processed and the tweets stored in some persistent manner in order to be able to retrieve them as a whole to be processed via Mahout algorithms. There are also streaming algorithms that can run on continuous streams of data and in that case the document collection would be dynamic, since new documents can be received constantly. In the second case, the S4 application would probably have to do some pre-processing on the data and then redirect the tweets received to the algorithm that processes them in order to extract valuable information.

A document is the element that has to be processed by the text mining algorithms. It is an ordered collection of words that are usually constructed

by a defined grammar and that make sense together. In our case, a tweet could be considered a document, since it's an entry inside the document collection which has to be processed to add value to the result.

A character is the basic element from a document and can be a letter, a number, a special symbol or a whitespace. One or multiple characters can form words. Words are the element that provides meaning to letter grouped together and delimited by other types of characters. In order to facilitate the document processing step, a document could be represented in a more structured manner as a set of all the words in the document for example. It is important to note however that it is recommended to optimize the set of words generated for that document in order to ignore stop words (common words that bring no value to the meaning of the document), symbolic characters and numerics.

A term can be a single word or a multiword phrase that has a specific meaning within the collection of documents in which is encountered. Let's take the following sentence as an example:

President George W. Bush ended his term at the White House in 2009.

Some terms that could be extracted from it are "President", "House", "term", but also "President George W. Bush" and "White House". Some of these terms have a specific meaning in this context, but in other context they could have a totally different meaning. For example, the word "Bush" can be thought of as a plant, but placed near "President George W. Bush" has a totally different meaning, being the surname of a president of the United States of America.

Concepts are "features generated for a document by means of manual, statistical, rule-based, or hybrid categorization methodologies" [20]. It is not unusual that the concepts describing a particular document collection are not actually frequent words in that collection. For example in a collection of sports articles, a concept describing it could be "competition", but this may not be a frequent word in that collection.

2.3 Batch vs. Stream processing

Batch processing is the execution of a series of programs or jobs over a set of inputs, instead of a single input, without any manual intervention between them. They usually involve high volume repetitive tasks and are suited for generating reports at the end of the business day for example. There are two types of batch processing jobs, one where all input can be processed independently from one another and one where the computation result aggregates all the input data.

Stream processing is another paradigm, equivalent to dataflow or reactive programming. Considering we have a continous stream of events, the main idea is to process each of them as they are received, in real-time. There are two types of stream processing applications, based on the requirements for processing all the events. There are hard systems, where the loss of an event or missing the processing deadline can be considered a total system failure and there are soft systems, where it is acceptable to drop some of the events if the processing queue gets full or it is acceptable to miss some processing deadline.

There are some clear differences between batch and stream processing. In batch processing, the system has access to all the data from the beginning, while in stream processing the system has access to one piece of data at a time. Also in stream processing the amount of data available is considered to be infinite, while in batch processing the amount of data is limited by the size of available containers.

In batch processing, we can compute something big and complex, while in stream processing we can compute a function of a single element or a small window of recent elements. A stream processing system needs to complete computation in near real-time, a few seconds are acceptable, while for a batch processing system the latency is usually measured in terms of minutes or more. For a stream processing system, the computations are independent from one another and are usually asynchronous.

The stream processing applications are fitted for financial products (for algorithmic trading, risk management or fraud detection) or for network and application monitoring (for intrusion detection), among others.

As presented in several articles about stream processing using Yahoo! S4, the MapReduce framework is suited for Big Data batch processing, but not so fitted for Big Data Stream processing [21] [22].

Chapter 3

Architecture

In this chapter the overall architecture designed for the purpose of this project is presented. In the first section (3.1) we will discuss about the physical infrastructure we have at our disposal and then we will present the conceptual model that defines the structure of the system (section 3.2). The actual system architecture is detailed in section 3.3.

Choosing Hadoop MapReduce as the framework on which we are evaluating the Mahout algorithms leads to a batch processing oriented approach. However, the construction of the actual dataset on which we are evaluating the algorithms involves processing the Twitter stream. The architecture of our project needs to take into consideration the necessary conversion from stream to batches of data. In order to be able to support this conversion, an intermediary persistent storage layer is required.

3.1 Infrastructure

In this section we will discuss about the physical infrastructure on top of which we've conducted our experiments. The infrastructure was provided by the Research and Development Laboratory (RDLab)[23] department of UPC. The purpose of this group is to provide comprehensive IT support to the Computer Science department of UPC and other research centers, in order to optimize project's research and development processes. The High Performance Cluster (HPC) from RDLab consists of more than 1000 CPU cores and more than 3TB of RAM memory. These are shared between multiple projects. The execution environment also offers support for Lustre (a parallel distributed file system), Hadoop, SMP and MPI computation and GPU computing, among others.

For the purpose of our project, we have identified two possible approaches. The first one was to use the Hadoop environment already deployed inside the cluster and run the Mahout algorithms straight from it. This came with a series of disadvantages and the main one was regarding data storage. Data needed to be accessed from the Hadoop environment and since that environment is shared across multiple projects, storing data over HDFS was not a good option for us. If we were to choose an independent storage system that could be accessed remote, we would have had to take into account the network latency for I/O operations when evaluating Mahout performance. Instead, we used virtual machines running on top of the RDLab cluster.

We used two virtual machines that have Linux containers over them for developing the project and running the experiments on a small number of nodes. There is one virtual machine with two LXC machines for the stream processing application and one virtual machine with four LXC machines for the Hadoop environment. Each LXC node is assigned 2 CPU cores and 2GB of RAM. For the extra nodes in our Hadoop environment (another 29 nodes) we used pure virtual machines, with the same configuration of 2 CPU cores and 2GB of RAM. The advantages of having several LXC machines over the same virtual machine is that there is no network latency between them. Two separate virtual machines might be deployed on the same physical node, which also eliminates network latency, but they could also be deployed on separate physical machines, which can increase the running time for our Hadoop jobs, because of the network communication time. However, we believe that this is a real life scenario, where the Hadoop nodes are deployed on different machines inside a cluster.

The virtual machines are created by placing a job in the queue of the HPC

Table 3.1: Cluster processors

Processor model	Number of cores	Frequency
Intel(R) Xeon(R) CPU X3230	4	2.66GHz
Intel(R) Xeon(R) CPU 3070	2	2.66 GHz
Intel(R) Xeon(R) CPU X5550	4 (8 threads)	2.66 GHz
Intel(R) Xeon(R) CPU X3363	4	2.83GHz
Intel(R) Xeon(R) CPU X5670	6 (12 threads)	2.93 GHz
Intel(R) Xeon(R) CPU E5450	4	3.00 GHz
Intel(R) Xeon(R) CPU X3220	4	2.40 GHz
Intel(R) Xeon(R) CPU X3350	4	2.66 GHz
Intel(R) Xeon(R) CPU 5130	2	2.00 GHz
Intel(R) Xeon(R) CPU 5160	2	3.00 GHz
Intel(R) Xeon(R) CPU 5110	2	1.60 GHz

environment. The queue is executed on an environment with 52 physical machines that might have different configurations between them. The list of processors that are found on those machines is presented in table 3.1, while the RAM memory can be somewhere between 8 and 64 GB for each physical node. Since this queue is shared with other projects there is no way of controlling on which of the physical machines the virtual machines will be deployed.

The nodes can also be stopped on demand by stopping the job that is executed and they can be accessed remotely via Secure Shell (ssh), which is a cryptographic network protocol.

3.2 Conceptual model

For the purpose of this project, there are three main conceptual entities to consider:

- The **stream processing system** that listens to the Twitter streaming API and processes the events received
- The **persistent storage layer** that gathers the tweets
- The **processing environment** inside which the Mahout algorithms are to be evaluated

These can be observed also in figure 3.1. Next in this section we will discuss each of these entities and present their main requirements we've identified.

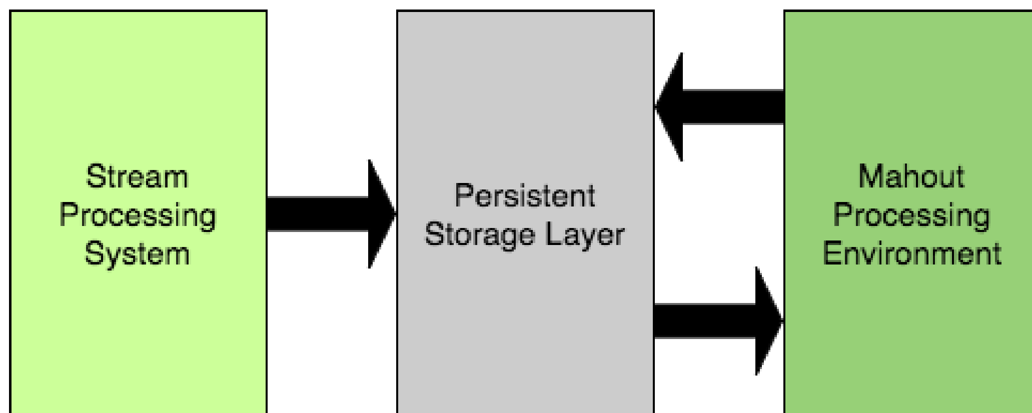


Figure 3.1: Conceptual model

The *stream processing system* needs to be able to listen to the Twitter Streaming API and process the events that are received. Processing these events implies executing some data cleanup operations over the text content of a tweet and write them to the persistent storage layer. It needs to be able to handle the rate of the events as sent by Twitter via the Streaming API. In our case, the stream processing system will be deployed inside a cluster with multiple nodes. One node will received the Twitter events and several others wil process them

The *persistence storage layer* has to store the tweets in a format that is easily readble using the Mahout framework. Since we are handling large datasets, the persistence storage layer should also be scalable and distributed. For the clustering algorithms we will evaluate, the output of the preprocessing steps

should be the TF-IDF vectors in a SequenceFile format stored in HDFS (we will discuss this format into details in section 6.1).

The *processing environment* should be scalable in order to be able to handle larger batches of data. It should also ensure quick access to the persistent storage layer, since the data mining algorithms can be considered I/O intensive. All data must be passed through at least once and intermediary results need to be stored sometimes directly on disk which can lead to a considerable number of read/write operations. Since the system needs to handle large datasets, computation could be divided into independent chunks to be executed in parallel and then the partial results could be merged into a final one. This means that we could use multiple workers to execute the Mahout algorithms on chunks of data.

These entities may not be necessarily indepenendent from one another, for example the persistent storage layer could be part of the actual processing environment for data proximity reasons.

In figure 3.2 the enahnced conceptual model can be observed, based on the notes discussed previously.

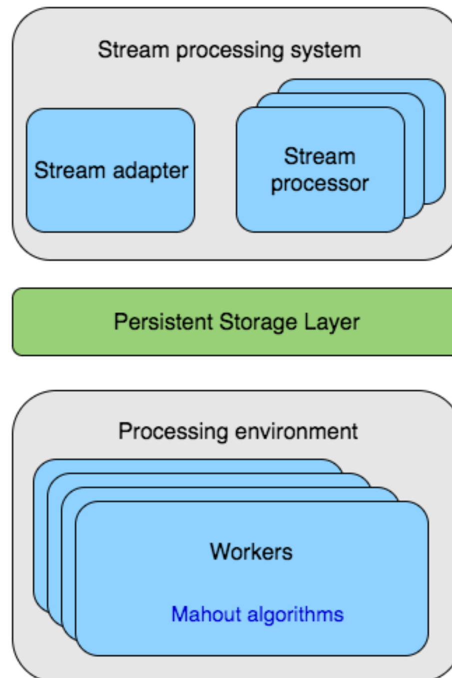


Figure 3.2: Enahnced conceptual model

3.3 System architecture

We will continue discussing about the actual technologies used for each of the entities described in section 3.2 and see how they interact with each other.

For the stream processing system, the Apache Yahoo! S4 solution was chosen. This is a distributed stream processing solution, which uses Apache Zookeeper [24] for clusters management. The architecture of the S4 environment is based on the actor model, where there are several nodes with different responsabilities that communicate with each other via messages, or in our case events. Inside the S4 environment there are two types of nodes which are deployed inside two different clusters. The first one receives events from the Twitter Streaming API and converts them to S4 events that are to be used internally. The second one gathers the S4 events and processes them and then stores them to the persistence storage layer. The S4 solution is presented into more details in chapter 4.

For processing the twitter stream as needed for the purpose of this project, a small number of nodes are required. We used three logical nodes for the S4 environment and deployed all of them on the same physical node. One node acts as a twitter-adapter and two others as twitter-processors. We discovered that a rate of approximately 900 tweets to process per second (as observed through our experiments), having two twitter-processor nodes is enough to handle the load.

For the processing environment, we decided from the begining of this project that we will evaluate Mahout performance over the Hadoop environment. The Hadoop architecture is based on the master-slaves model as described in section 2.1.3. For the persistent storage layer, we evaluated several solutions and chose HDFS for this project. More details about the alternatives we've considered can be found in section 5.2. In HDFS data is stored into chunks and distributed across the nodes inside the Hadoop cluster. This brings the advantage of data proximity when performing computing operations.

Inside our Hadoop cluster, we used a variable number of nodes throughout the experiments we conducted. We started with four nodes (one master and three slaves) and conducted several experiments using 4, 8, 16 and 32 nodes.

The actual architecture of our system is presented in figure 3.3.

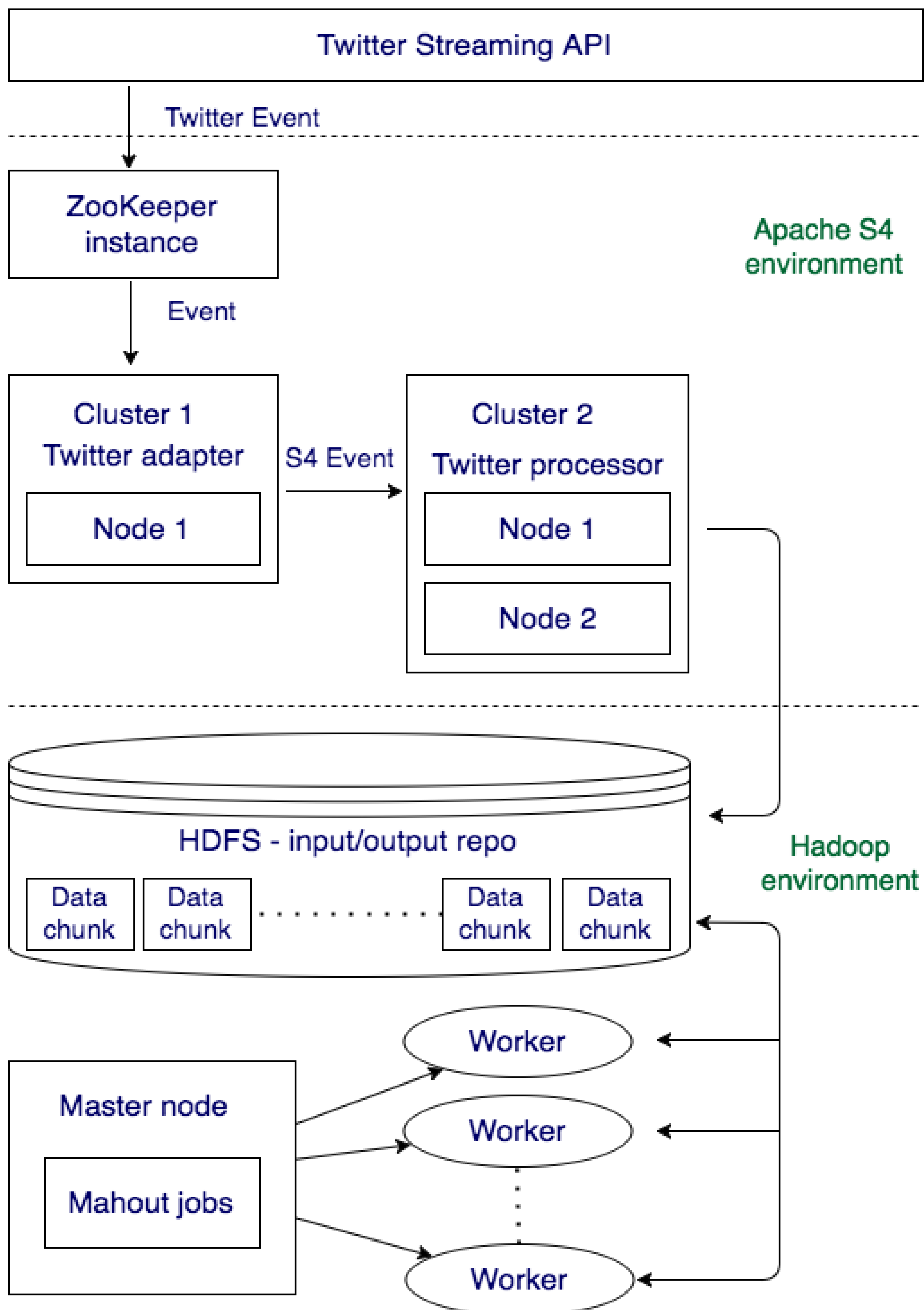


Figure 3.3: System architecture

Chapter 4

Stream processing system

S4 is an open-source project under the Apache umbrella. It's stands for *Simple Scalable Streaming System* and it was released initially by Yahoo in October 2010. Since September 2011 it is an Apache Incubator project. It's main scope is to provide a scalable, extensible and fault-tolerant platform that is to be used by developers for processing continuous unbounded streams of data in their applications. In addition, the main reason S4 was designed in the first place is to "solve real-world problems in the context of search applications that use data mining and machine learning algorithms" [25].

The S4 architecture is based on a cluster that has a decentralized model, in which all nodes are symmetric and there is no single point of failure. This model simplifies deployment and making changes in the cluster configurations. For the cluster management part, S4 is integrated with ZooKeeper, another Apache open-source project, which provides coordination services for distributed applications. ZooKeeper maintains the configuration information and provides distributed synchronization across the nodes in the cluster. S4 ensures scalability through the easiness in adding a new node inside the cluster, via the ZooKeeper management layer.

Most MapReduce platforms, like Hadoop, are highly optimized for batch processing, they operate on a static set of data. In stream computing, the paradigm is to have a stream of events that flow into the system at a given data rate and which you cannot control. The processing system must keep up with the event rate or degrade gracefully by eliminating events.

Applications using S4 can be easily developed and deployed, using a Gradle or Maven integration. The latest version of S4 is 0.6.0 and this is the one that has been used in this project. Since June 2013 there have been no new

releases, but another Apache project for stream processing has been launched around then, Storm.

The S4 framework has been used in other research project, like a flight tracking application[26] that receives data containing information about plane positions and other metadata and computes the list of n nearest airports to the aircraft radius and country over the plane is flying, among others. The processing operations computed with the S4 application for the flight tracking project were mainly distance computations and extracting informations from hardocded data, like list of countries or list of airports. The performance results proved the efficiency of the developed system, being able to update each flight status every 4 seconds on average. In this project, for constructing our Twitter data set, the necessary processing involves mainly string manipulation operations and file writes to HDFS.

In this chapter, the Apache Yahoo! S4 model is presented and a comparison between Apache S4 and Apache Storm is provided. Later on the Twitter Stream is described and in Section 4.4 the actual S4 implementation suited for this project is described.

4.1 Apache Yahoo! S4 model

The framework, written in Java, is designed to consume streams, compute intermediate values and emit other streams in the distributed computing environment. An S4 application can be seen as a graph in which the nodes are Processing Elements (PE) and the edges are streams. Different PEs communicate asynchronously by sending events on streams. A stream is normally used to ensure communication between PEs, but there are also other special type of streams, external streams, which send events outside of the application or receive events from external sources. In order to convert external streams into streams of S4 events a special application can be used, usually called an adapter.

The data processing model is based on Processing Elements (PE), which are the basic computational units in S4. Messages are routed between PEs as data events. Every PE is uniquely identified by its functionality, as defined by the PE class and configurations. Every PE can consume only some types of events, depending on the configuration, and can produce output events.

The processing nodes (PN) are the logical hosts for PEs and are responsible for listening to events, dispatching events via the communication layer and emit output events. The communication layer provides cluster management and automatic failover to standby nodes. It also maps physical nodes to logical nodes, all events are directed to the logical nodes and emitters are unaware of the physical nodes or when the logical nodes are re-mapped due to failures. The coordination layer uses ZooKeeper for this cluster management part and TCP as the transmission protocol.

The figure below (4.1) describes the general architecture of the S4 model in more detail.

In order to develop a new Processing Element to be used inside our application, two main methods need to be considered:

- onEvent() - the input event handler.
- onTime() - output mechanism which is an optional method to output the result of the PE to an external system. This one can be configured to be invoked at regular time intervals or after receiving a specified number of input events.

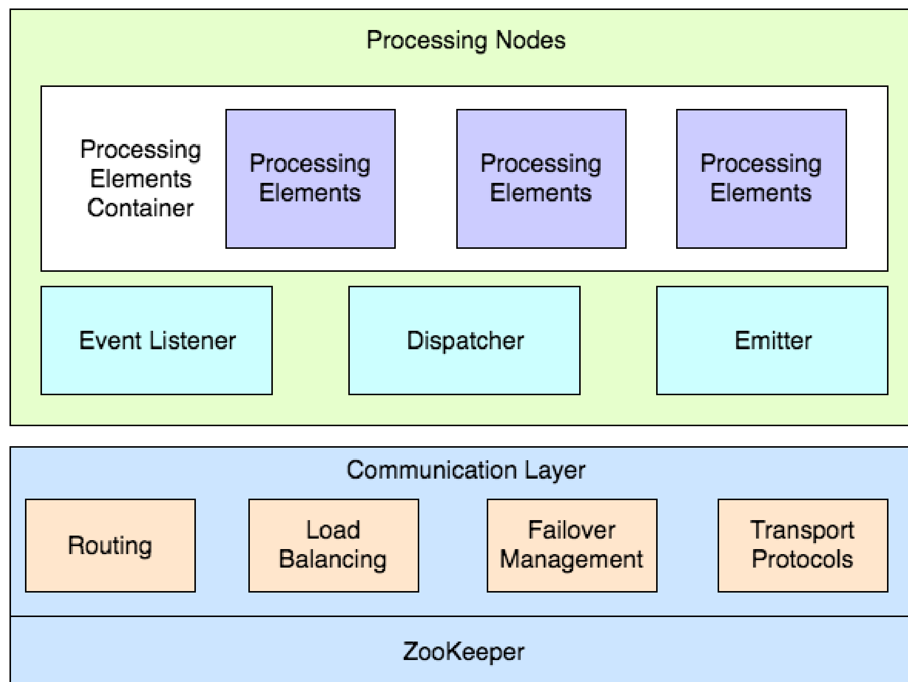


Figure 4.1: Yahoo S4 architecture

4.2 Apache S4 vs. Apache Storm

Apache Storm is another Apache open-source project for reliably processing unbounded streams of data. It was first released by Twitter in 2011 and became a top-level project of Apache in 2014. It is now used by a series of well-known companies, like Twitter, Yahoo, Flipboard.

One Processing Element from S4 can have multiple inputs or multiple outputs. Apache Storm uses spouts and bolts as main entities. A spout can create many streams, while a bolt can consume one or more streams and output multiple streams. The processing model of Storm can be seen in the figure below (fig. 4.2). The spout is also the entity that listens and receives events from external streams. All the processing operations like filtering, aggregation or database communication happen inside the bolts.

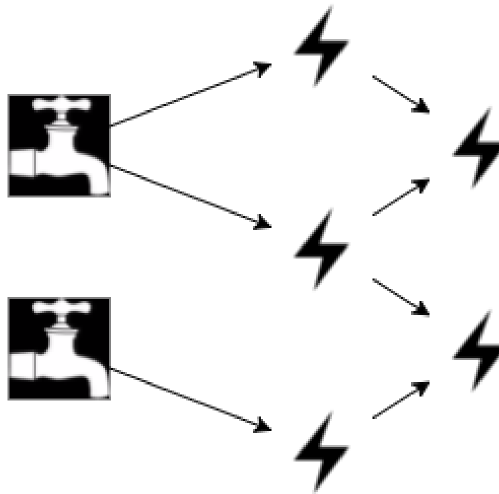


Figure 4.2: Storm processing model

Apache S4 follows the Actors programming paradigm, having the PE as an actor. The logic inside a PE can be very simple, similar to the MapReduce model, where mappers and reducers perform simple operations. Storm doesn't have an explicit programming paradigm, it relies on bolts and spouts to process partitions of the stream.

S4 uses a push model, events are pushed to the output stream as fast as possible and if the receiver buffer gets full, all other events are dropped. This could happen at any stage of the pipeline, from the Adapter to any PE. Storm uses a pull model instead, where each bold pulls data from its source, a spout or another bolt. This has some advantage, since there is only one

place in which event loss could happen, if the spout can't keep up with the external event rate. For our application this is not a valid concern, since there is no impact if any tweet is dropped along way, just that it won't be part of our dataset.

Apache S4 provides fault tolerance via checkpoints. However, the events that are received after a the last checkpoint and before the recovery are lost, so it does not ensure that no events are lost. Storm provides guarantees for the delivering of events. Spouts are responsible to keep events until they are completely processed in order to replay them from the beginning if necessary.

Both projects are implemented in Java. S4 provides no centralized administration mechanism, while Storm has an Admin Interface.

One advantage of Apache Storm is that it provides integrations with multiple external systems and other libraries, such as Apache Cassandra, Apache SOLR, Apache HDFS. These could be used as an intermediary between the stream processor and Mahout MapReduce algorithms implemented on top of Hadoop. S4 does not ensure such integration at the moment, but it can be done by the developer of the application.

Another advantage of the Apache Storm framework is that it can be debugged inside the editor used (Eclipse for example), while the only way of debugging an S4 application is by deploying it inside the cluster and sending mock events by hand to the adapter, so it is slow to debug.

One downside that we discovered while experimenting with S4 is that it is difficult to redeploy an application. There is no straight-forward command to undeploy an application from a cluster.

The advantages of S4 over Storm are that is does automatic load balancing and the framework seems easier for new application development. In his book about another Apache project, Dayong Du considers that "Storm gives you the basic tools to build a framework, while S4 gives you a well-defined framework" [27]. For the purpose of this project in which the loss of some events is not critical, S4 is a good choice for stream processing.

4.3 Twitter Stream

Twitter provides both a RESTful and a Streaming API endpoint in order to retrieve publicly published tweets. The Streaming API endpoint reduces the overhead caused by polling the REST endpoint to see if new tweets were published. The REST APIs are useful when conducting specific searches, reading user profile or posting new tweets. The Streaming APIs are to be used when the newest tweets need to be retrieved as they are published. There are multiple streams offered by Twitter, each with its specific use case:

- Public stream: contains public data on Twitter, useful for following specific users or topics and data mining
- User/site stream: contains approximately all the data associated with the view of a particular user or multiple users

Throughout this project the public stream will be used to gather data from. This stream can be accessed via a HTTP GET request, which returns a small sample data of statuses publicly available on Twitter, or a HTTP POST request, which can be used to filter by specific users, topics or locations. We will use a GET request with no filters, in order to gather all types of tweets and then insert them as input data for clustering algorithms in Mahout.

The throughput of tweets that are sent over via the Twitter Streaming API is around 1% of the throughtput that the Twitter platform actually handles.

4.4 Processing the Twitter stream with S4

S4 is an easy to install and configure platform, it's main dependency is gradle. We used one of the machines in the RDLab for the S4 application deployment. There were few necessary steps for configuration: downloading the source code and then compiling, installing and building the application, via Gradle.

For the Twitter stream processing application, we started with the Twitter example provided in their documentation and modified in order to suits the needs of this project. There are two application which are deployed inside two different clusters:

- twitter-adapter
- twitter-processor

The overall architecture of the Yahoo S4 system configured for the needs of this process can be seen in figure 4.3. We will further discuss the two types of S4 nodes, the adapter and the processor into detail.

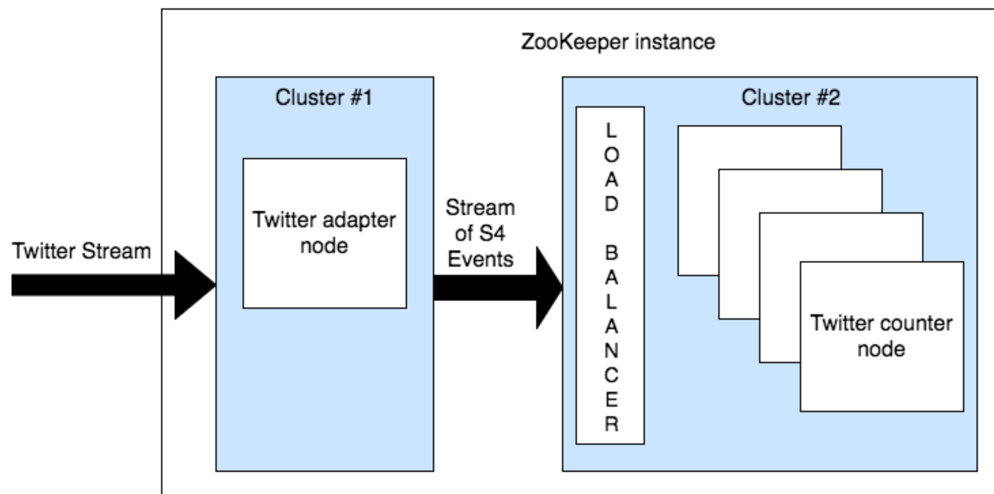


Figure 4.3: S4 model for processing the Twitter Stream

4.4.1 Twitter-adapter

This is the adapter application, used to convert the Twitter stream into a stream of S4 events. Starting with the initial twitter-adapter project, which

sent over to the twitter-processor (initially it was called twitter-counter) application only the text of the tweet, we modified it in order to use the latest version of the twitter4j library and to send more meta information about a tweet. Initially only the text content was sent to the twitter-processor application, but for the purpose of this project we added the language in which it was written, if it has been retweeted and how many times, the list of urls or hashtags which are contained inside the tweet and more information about the author of that tweet, the user id and the number of friends, followers and statuses that he has. This information could be used in a data mining algorithm in order to obtain more relevant information from the collection of tweets, like the most trending ones.

Filtering the tweets that are received by the twitter-adapter app by the language in which they are written could be possible in theory, via the Twitter Stream API, since the request can receive filtering parameters for language and other attributes, but in practice is doesn't seem to work as expected. In our attempt of filtering the Twitter Stream using the *language=en* parameter using the HTTP POST API, the stream continued receiving tweets in japanese or other non-English languages. So we decided to receive all tweets in all languages and do my own filtering before sending the tweets to the twitter-processor application, using the language attribute of the Tweet.

The twitter-adapter application communicates with the twitter-processor one via a remote data stream. The process is pretty straight-forward, when an application inside the ZooKeeper cluster creates a new output stream, it is exposed in ZooKeeper and other applications that define input streams with the same name are automatically connected.

4.4.2 Twitter-processor

The twitter-processor application receives the Tweets as S4 events and uses two Processing Elements in order to forward data to the Hadoop cluster:

- TweetCleanerPE
- TweetWriterPE

The data flow through the twitter-processor application can be seen in figure 4.4 and is described in details next.

The TweetCleaner receives the tweets from the twitter-adapter via the "Raw-Status" stream and processes their content, as part of the data preprocessing

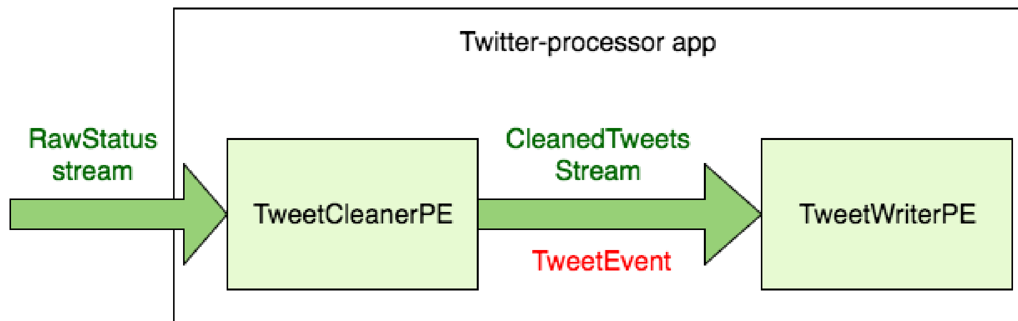


Figure 4.4: Data flow in the Twitter Processor S4 APP

step, more exactly data cleaning. All types of strings that don't add any value regarding the subject of the tweet are removed from the tweet content by this Processing Element. One example would be the references to other Twitter users, which are strings that start with the at-sign character ("@"). In some cases, these references are preceded by the "RT" string, which is a form of showing that the tweet is retweeted from the user that is referenced. The "RT" string can also be removed from the tweet content for the purpose of our project. Another example would be the URLs that are published in a tweet, which we can identify by the substring with which they start, either "http://" or "https://". The URLs published in a tweet are in a shortened version than the original, their form is usually: *https://t.co/[id]*. These can be removed by using a regex pattern matcher. After the content cleaning, the TweetCleanerPE sends TweetEvents to the TweetWriterPE.

A TweetEvent only contains the id of the tweet and the cleaned content, since this is the information needed to be written in HDFS and later processed by the clustering algorithms.

The TweetWriter received the tweets from the TweetCleanerPE, via the "CleanedTweets" stream, which was defined inside the twitter-processor app. This Processing Element is configured to receive events and temporarily store them inside a local queue, until the actual writing operation is triggered by a timer. The timer is configured to trigger the writing operation every 30 seconds and once triggered, the tweets inside the queue are written on disk. The writing operation can be viewed as a batch operation in our case, since it waits for the queue to fill up with multiple tweets and then writes all of them at once. This approach was chosen because of the I/O operations which are expensive and can become blocking for the TweetWriterPE, if the tweets are received at a fast rate and they also need to be written at the same rate. By writing them in batches, the speed at which they need to be written is

practically reduced, without an impact on the capability to receive them, at any speed they might come.

The S4 framework provides a mechanism for creating checkpoints to PEs in order to ensure fault tolerance in case one running node becomes unavailable and another one which was in standby takes its place. For our project, no checkpoints are needed, since the worst case scenario is to skip some of the tweets received and ignore them throughout the data mining algorithms. Every tweet that is received is non-critical and independent of others. Since we can construct our dataset with any tweets.

In order to measure the performance of the S4 applications and the state of the nodes, some metrics were gathered. These metrics are made available by the S4 framework integration with yammer, they just have to be implemented and configured by each application. We are interested in such metrics from the twitter-processor application, since this is the one that filters the tweets and redirects them to the persistent storage layer. For each node inside this cluster, there is a special metrics directory created where relevant information is organised into multiple files, as follows:

- dequeued@RawStatus.csv: which shows how many events were dequeued from the "RawStatus" stream since the start of the node
- dropped@RawStatus.csv: which shows how many events were dropped from the "RawStatus" stream since the start of the node
- received-bytes.csv: which shows how many bytes were received from all streams since the start of the node
- received-events.csv: which shows how many events were received from all streams since the start of the node

There are more files generated, but I consider the ones above the most relevant ones. All of them are in the same format, which is also described in the first row of the .csv file:

time,count,1 min rate,mean rate,5 min rate,15 min rate

The time is expressed in seconds since the start of the node, but the metrics writer can be configured to register information at a given interval, in our case it is one minute. The count column contains the number of events/bytes that are of interest for that particular metric. The mean rate column gives

us the number of items counted (be it events or bytes) since the start of the node divided by the number of seconds that have passed since the start of the node. The 1 / 5 / 15 min rates are similar, they just take into consideration items that were counted in the given time interval, instead of all the items since the start of the node. These kind of metrics are relevant in order to see the send rate of public tweets published in English received via the Twitter Stream API.

Chapter 5

Dataset and persistence layer

A dataset is a collection of data entries. The dataset is a very important aspect of this project since it can influence greatly the results of our experiments. Throughout this project, we chose to construct our own dataset instead of using an already generated one. The main advantage of generating a new dataset is that it can be constructed to suit the exact needs of the projects, in terms of data size, data structure and data storage.

In the first section of this chapter the particularities of data entries and of the dataset, the receive rate of tweets in S4 and the dataset size are discussed. In section 5.2 we will discuss about the persistent data storage solution adopted, but the alternatives considered before making the decision are also presented.

5.1 Dataset

The dataset that we constructed is based on tweets received via the Twitter Streaming API. We've discussed more about this API in section 4.3, but in this section we will discuss about the data that is actually being received and how this data is transformed in order to be used later on by the Mahout data mining algorithms.

5.1.1 Data structure

The tweets received via the Twitter Streaming API contain, besides the text content, a lot of metadata that can provide additional information about the popularity of the tweet or the context in which it was published. Examples of metadata for a tweet are:

- *language* - described in a BCP 47 format or equal to "und" if the language could not be detected
- *coordinates* - the geolocation from which the tweet was published
- *creation date*
- *entities* - special entities which are extracted from the tweet content:
  - *urls*
  - *hashtags*
  - *user mentions*
- *favorited* - true/false, indicates if the tweet has been liked or not
- *favorite counter* - the number of likes this tweet has received
- *retweeted* - true/false, indicates if the tweet has been retweeted or not
- *retweet counter* - the number of times this tweet has been retweeted
- *user* - the profile of the author of the tweet, which contains:
  - *id*
  - *creation time* for the user account
  - *description*

  - *followers counter* - indicates the number of followers the user has
  - *friends counter* - indicates the number of friends the user has (which is equivalent to the number of accounts the user is following)
  - *profile image*
  - *status* - the most recent tweet that the user has published
  - *statuses count* - the total number of tweets that the user has published over time

  among others.

Not all this information is of use for the purpose of our project, so only certain fields will be extracted and stored. The twitter-adapter S4 application receives all the information mentioned above from the Twitter Streaming API. Before passing it further to the twitter-processor app, it constructs own Java objects which store less information. The structure of these objects that are sent over the RawStatus S4 stream is:

- *id* - the tweet id as assigned by Twitter
- *text* - the text content of the tweet
- *lang* - the language in which the tweet was published
- *isRetweet* = "retweeted" from the previously discussed tweet structure
- *retweetCounter*
- *urls* - the list of urls contained in the tweet
- *hashtags* - the list of hashtags contained in the tweet
- *userId* - the id of the author of the tweet
- *userFollowers* - the number of followers that the user has
- *userFriends* - the number of accounts that the user is currently following
- *userStatuses* - the number of tweets that the user has published

Figure 5.1: Tweet object structure in the S4 application

This structure is also represented in figure 5.1.

The id is useful for uniquely identifying a tweet in the dataset, while the text content will be used for text processing and information extraction from published content. The other metadata that is being sent over to the twitter-processor app can be used for determining the most popular tweets or for validations regarding the popularity of a tweet.

Hashtags are very commonly used on multiple social media networks, like Facebook, Instragram and also Twitter. A hashtag is a "type of label or metadata tag used on social network and microblogging services which makes it easier for users to find messages with a specific theme or content" [28]. On Twitter hashtags can be considered a method of expressing more with less words or characters, if we were to consider the length limitation of a tweet. Even though hashtags can be formed from multiple words like *#summervibe*, they can bring valuable insights about the topic of a tweet and should be further taken into consideration while processing the tweet text content.

URLs are references to other web resources, usually websites. The URLs published on Twitter posts are shortened by default by the social media platform. This is a mean of allowing users to posts long URLs while keeping the length restriction on all published tweets. The length of any URL published on Twitter is of 23 characters, even is the original URL had a smaller length. The shortened version of URL has the following form: *https://t.co/{identifier}*.

Twitter also protects its users from malicious websites by checking the original URL against a list of potentially dangerous sites and gives warnings if there are any issues. The URLs found in the actual text content of a tweet don't bring any value regarding the topic of the tweet, so they should be extracted from the content as part of the data cleaning preprocessing step.

In the twitter-processor app there are two processing elements, as we've discussed in section 4.4. In the TweetCleanerPE there are some data cleaning operations executed over the text content and then the tweet id and the cleaned tweet content data is sent further on to the TweetWriterPE. The other metadata information that was extracted from the original tweet content in the tweet objects we previously described reaches the twitter-processor app, but currently it's not persisted anywhere. The project could be further developed and the twitter-processor app could be upgraded in order to store the metadata information to be used in other algorithms than the ones we've evaluated. For the moment the tweet id and the cleaned content is all the necessary information.

Given the simplified data structure we've reached, the information could be stored in a key-value format, where the key is the tweet id and the value is the tweet text content.

Data changes its structure throughout several points in this project before being permanently stored in the persistence layer. The changes we've previously discussed can be seen in figure 5.2.

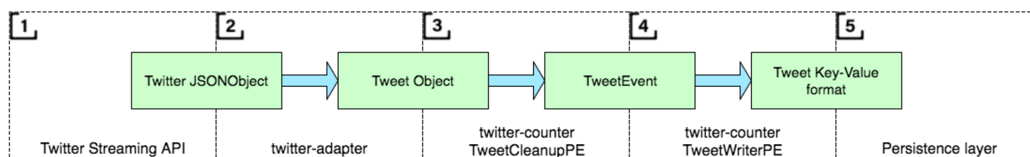


Figure 5.2: Changes in the data structure of a tweet

5.1.2 Data size

Given the length limitation of a tweet content, the size of one data entry is very small. Tweets contain UTF-8 characters and such a character can be represented on 32 bits, which is equal to 4 bytes. A maximum length of 140 characters means a maximum size of 560 bytes. A tweet id can be represented as a long number, so it requires up to 8 bytes. Based on this values, the maximum memory space size required for storing a tweet is aroung 568 bytes.

Taking into consideration the cleanup operations we are performing over the text content of a tweet before storing it, the size of a data entry it most probably smaller than this. So we decided to compute the average size of an actual tweet based on the disk space size the dataset occupies and the total number of stored tweets.

$$avg\ tweet\ size = \frac{used\ disk\ space}{total\ number\ of\ stored\ tweets} \tag{5.1}$$

Using the above formula, the average memory size required for storing a single data entry resulted to be around **91 bytes**. It's around 6 times smaller than the worst case scenario we assumed initially and one of the reasons that explains this is the fact that the urls and user mentions in the tweet content can take more than 50% of the entire text.

After storing the tweet contents in the persistence storage layer, there are other preprocessing steps that are performed before the actual Mahout algorithms are executed. These are discussed in section 6.1 and will generate new data that will be given as input for the data mining algorithms evaluated.

5.1.3 Data receive rate

In figure 5.3 the receive rate of events in the twitter-processor app can be observed. There may be multiple events coming from via the Twitter Streaming API to the twitter-adapter app, but only tweets written in english are passed further on to the twitter-processor app and are taken into consideration for the purpose of our project.

The average number of events per second is *16,21*.

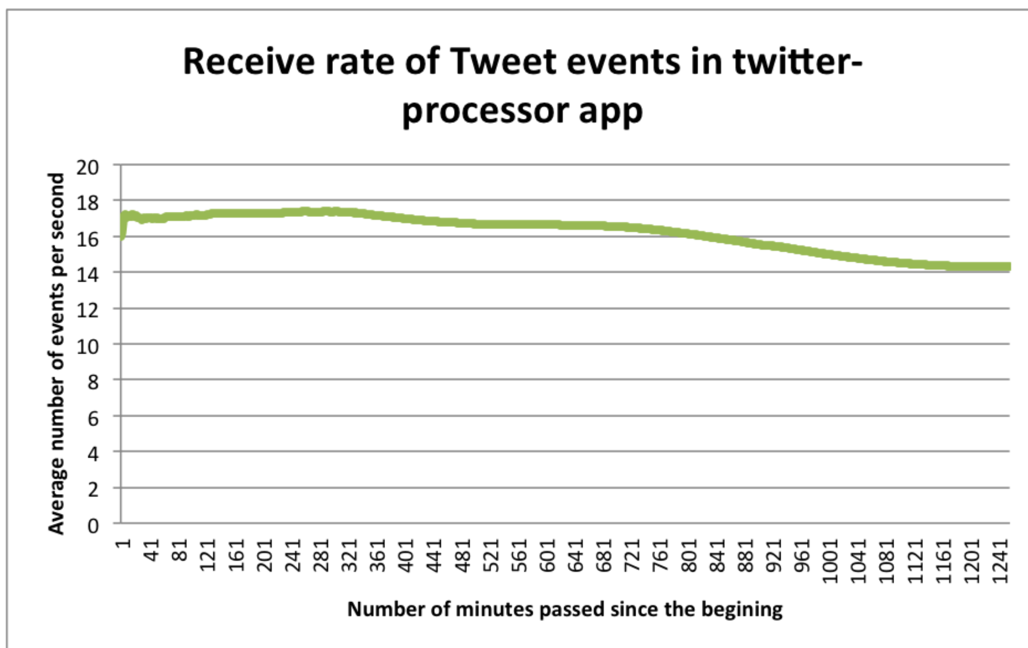


Figure 5.3: Receive rate of tweet events over 20hours window

5.2 Persistence Layer

The persistence layer is the data keeper of our system. Tweets are gathered here in order to be processed later on in batches by the Mahout data mining algorithms.

Data can be persisted usually in files inside a file system or in databases which provide more advanced mechanisms for data storage and retrieval, inteligent memory managed data, indexes, redundancy and concurrent access, among others.

The relational databases commonly used present some challenges when it comes to scalability, high throughput and high volume of data. NoSQL (non-SQL or non-relational or not-only-SQL) is a database model for storing and retrieving data that is modeled differently than in relational databases. NoSQL databases are usually used in big data and real-time web applications, because of their horizontal scalability, performance, simplicity of design or flexibility of data model used (schema-less). There are various approaches for the data model in NoSQL databases. The two that present interest for us are:

- *column-oriented databases* - they manage objects that resemble key-value pairs with three elements:
  - key (or an unique name) used to reference the column
  - value, which is the content of the column
  - timestamp, used to determine which is the most up-to-date value

  Some examples of column-oriented databases would be HBase and CassandraDB.

- *document-oriented databases* - they manage documents, which are usually in a semi-structured format. The document structure is not imposed by the system and the document can be in multiple formats, like XML, JSON or even PDF, depending on the actual database type. One example of document-oriented database is MongoDB.

The CAP theorem presented by Eric Brewer in 2000 [29] is usually mentioned in NoSQL databases contexts. It states that is it impossible for a distributed computing system to simultaneously guarantee all three of the following properties:

- C: Consistency - all nodes have knowledge of the same data in the same time
- A: Availability - every request that comes to the system receives a response, whether it's a success or a failure
- P: tolerance to network Partitions - the system continues to operate despite of arbitrary partitioning caused by network failure

Different NoSQL solutions choose different properties from this theorem for the trade-off.

The persistence layer solutions that were taken into consideration and evaluated for this project are the following:

- HBase
- CassandraDB
- MongoDB
- *Hadoop Distributed File System (HDFS)*

In the following sections we will discuss each of these options into details and compare them in order to see what are the arguments for choosing HDFS for the purpose of this project.

5.2.1 HBase

HBase [30] is an open source NoSQL database. The project was started in 2006 and became a Hadoop subproject in 2008, running on top of HDFS. Since 2010 it is one of the Apache top-level projects. The HBase project was designed based on Google's BigTable paper published in 2006 [31] and it is developed in Java. Many of the key-players from the IT industry use the HBase solution for their projects. As an example, Facebook started using it for it's messaging platform and then extended it in order to better meet it's requirements, to provide fault tolerance and availability even in case a region server fails [32]. Adobe and Netflix [33] have also adopted this solution for their Hadoop based projects. For Netflix, the advantages that HBase brought were the possibility of dynamically growing the cluster and redistributing the load across nodes and being able to execute both real-time HBase queries and batch map-reduce Hadoop jobs over the data stored in HBase.

The main advantage of HBase is that it provides a fault tolerant way for storing large quantities of sparse data. Data can be considered sparse when it has a large number of dimensions and the majority of them have missing values (equal to 0 or null). HBase is a column-oriented database. HBase tables content can be accessed via command line, Java API, but also REST APIs. They can also serve as input or output for MapReduce jobs. HBase provides data replication across clusters and has an automatic failure support.

HBase can be configured to run in standalone, pseudo-distributed or distributed mode. Standalone implies using a single node instance where HBase is deployed. In standalone mode HBase does not require a HDFS instance, it can use the local file system. Pseudo-distributed mode means that there are multiple nodes deployed on the same physical machine and is usually used for testing and prototyping. Distributed (or fully-distributed) mode is when there are multiple nodes deployed on multiple physical machines. Both pseudo-distributed and fully-distributed require the HDFS instance.

The main components in HBase are:

- *MasterServer* - responsible for maintaing the cluster, load balancing, handling schema changes and other metadata information
- *RegionServers* - responsible for handling read and write requests for the regions stored on it, where a region is a table split and spread across the region servers. Each RegionServer contains:
  - WAL - Write Ahead Log, which is a file in HDFS, used to store new data that hasn't been persisted permanently yet in case of a failure
  - BlockCache - the read cache that stores frequently read data in memory, using the LRU (Least Recently Used) algorithm
  - Memstore - the write cache
  - Hfiles - the actual files that store the rows as sorted Key-Value pairs on disk

The data model in HBase contains the following entities:

- *Table* - basically a map of key-value pairs, where the values are the rows
- *Row* - can be associated with the actual object being stored and has a key; similar keys are stored close to each other which is why HBase

offers support for range based scans; a row can also be viewed as a key-value map in which the keys are the columns associated with each row

- *Column* - can be viewed as a key for each row, but not all rows have values for the same columns; the value for a column key can also be viewed as a key-value map in which the key is the timestamp and the value is the actual value of the entry

- *Column families* - group together multiple columns and must be declared at schema definition time; however, the actual columns from a column family may differ from row to row, there is no restriction there; columns from the same column family will be stored close to each other on disk

These can also be seen in figure 5.4.

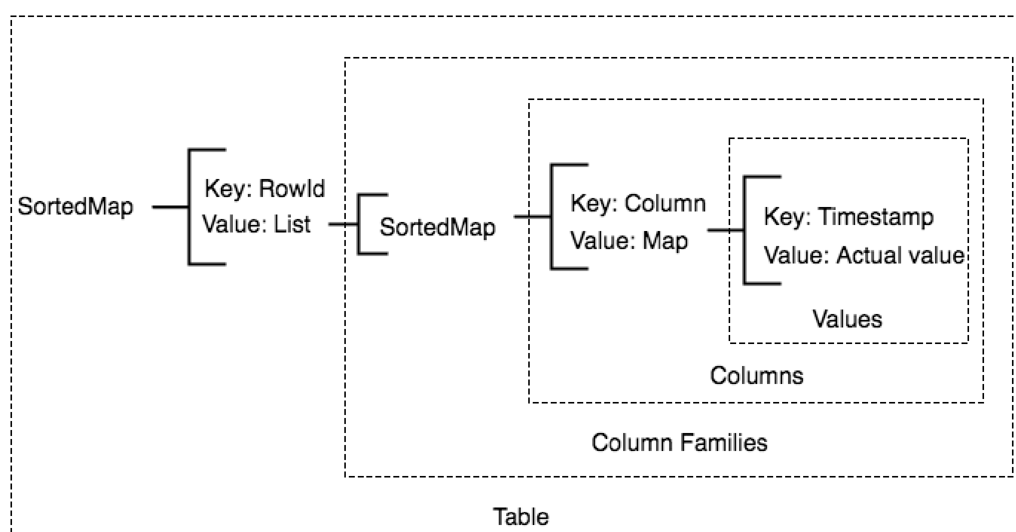


Figure 5.4: HBase datamodel

An important thing to underline about HBase is that is does not support specific data types, everything is considered to be an array of bytes.

HBase provides consistency and partition tolerance (CP) from the CAP theorem.

5.2.2 CassandraDB

Cassandra [34] is another Apache open source project that combines Google's BigTable data model and the distributed system technologies from Amazon's Dynamo. It offers a distributed database management system able to handle large datasets spread across multiple commodity servers.

Cassandra was initially developed at Facebook and then released as an open source project in 2008. In 2009 it became an Apache Incubator project and since 2010 is an Apache top-level project, like HBase. Cassandra was adopted by many companies as a solution for their needs of massive scalability, strong security, always on architecture and high performance. One example would be eBay that uses Cassandra for storing user activity, which is later used by their recommandation engine [35]. Coursera, the online learning platform that features over 1000 courses and over 10 million users, migrated from MySQL to Cassandra because of the higher availability, scalability and performance [36]. SoundCloud has also chosen the Cassandra solution to use in their Activity Feed and real-time statistics, because of the easiness of operating it, scalability and performance on heavy write loads [37]. Other well-known entities that use Cassandra are Spotify, The New York Times, Instagram, Adobe (for its data management platform that can be used to identify the most valuable segments in digital channels, called Audience Manager), GitHub (for their commit activity) or NASA (for handling the vast amount of security data collected and maintained [38]).

Cassandra's architecture has a descentralized model, there is no single point of failure and all nodes inside the cluster have the same role and are indepenendent and interconnected to each other. There is no master, all nodes can service any request. It can support replication in the same datacenter or across multiple data center, which ensures redundancy and failover and disaster recovery mechanisms. It offers elastic scalability, ensring that by adding new hardware, more customers and data can be accomodated. It has a linear-scale performance, the response time is constant if there is a throughput increase and an increase in the number of nodes. Cassandra offers support for Hadoop MapReduce, Apache Pig or Apache Hive. It also comes with its own query language, Cassandra Query Language (CQL), and language drivers for main programming languages like Java, Python, Node.JS and C++.

Regarding data storage, Cassandra has a flexible data storage solution, that can store all data formats: structured, semi-structured or unstructured. It is also able to dynamically handle changes in the data structures.

The datamodel of Apache Cassandra is a hybrid between key-valued and column-oriented database management system. The keyspace in Cassandra could be considered the equivalent of a schema in a traditional RDBMS model where tables are stored. A column family in Cassandra is the alternative for a table in q RDBMS, but it is more dynamic and flexible, since the structure of a column family doesn't have to be declared at schema definition. Each row in a column family is identified by a key. Columns are tuples of name-value-timestamp entries.

The main entities when discussing about Cassandra are:

- Node - the place where data is stored
- Data center - a collection of multiple nodes
- Cluster - container of one or more data centers

Cassandra uses a commit log where every write operation is logged before being executed, as a crash recovery mechanism.

The project is developed in Java and can be monitored via Java Management Extensions (JMX). This allows adding new nodes, decomissioning or removing old ones and extracting metrics about disk usage, latency, garbage collection and so on. Across the cluster, the nodes use a gossip protocol in the background in order to detect faulty nodes. There are two partitioning models of data inside a Cassandra cluster:

- OrderPreservingPartitioner (OPP) - distributes the key-value pairs in a manner that similar keys are close to each other
- RandomPartitioner(RP) - distributes the key-value pairs randomly across nodes

OPP offers the advantage that there are fewer nodes accessed to get some keys, but the data distribution is uneven between nodes in the cluster. Using the RP partitioner guarantees an even data distribution across nodes.

Cassandra provides Availability and Partition-Tolerance (AP) from the CAP theorem.

HBase vs. Cassandra

In table 5.1 some of the advantages of both HBase and Cassandra over each other are presented.

Table 5.1: HBase vs. Cassandra advantages

HBase	Cassandra
CP	AP
-	greater performance for single-row read (as long as eventual consistency are sufficient for the use-case - the quorum reads are slower than HBase reads)
support for data aggregations	-
support for range-based scans	-
-	SQL-like language for operations
optimized for reads	optimized for writes

5.2.3 MongoDB

MongoDB [39] is a NoSQL databases, but it's more document oriented, the data is stored as JSON-like documents with dynamic schemas. It is also free and open-source as HBase and Cassandra, but MongoDB Inc. (the company who developed it) also offers some enterprise/professional paid services, like commercial support. Some of the key-players in different IT-related industries use MongoDB for their services. One of them is Adobe and the Adobe Experience Manager (AEM) product that is a Customer Management System (CMS) indending to accelerate the development of digital experiences in order to increase customer loyalty and engagement. It uses MongoDB in order to store petabytes of data in content repositories [40]. Other projects/companies that use MongoDB as a data storage solution are Foursquare, LinkedIn (for its internal learning platform) or eBay.

One of MongoDB's main features is allowing ad-hoc queries, based on fields, ranges or even regular expression searches. It can also be configured to return a random sample of results from the dataset. MongoDB allows data replication in a replica set and each replica set member can act as primary or secondary replica at any given time. The write and read operations are usually executed on the primary replica, while the secondary replica maintains a copy of the data for automatic failover. MongoDB can scale horizontally using sharding mechanism. The data can be splited by shard keys, range or hash functions across shards. It also offers a dynamic splitting mechanism

that keeps chunks of data from growing too large.

MongoDB provides Consistency and Partition Tolerance (CP) from the CAP theorem.

5.2.4 Hadoop Distributed File System

We presented multiple NoSQL solutions which bring a lot of advantages of using them in a BigData processing context. It is time to focus our attention towards a different type of persistence layer, a distributed file system. We've discussed about Hadoop in general and it's architecture in section 2.1.3, we will continue underlying the advantages of HDFS and some of its use cases.

Built-in redundancy and failover are some of the main advantages of using HDFS. Hadoop's replication and failover mechanisms ensure that even if a node becomes unavailable, the batch processing jobs results are not affected.

Handling Big Data with the characteristics that come with it, variety, velocity and volume (as discussed in section 2.1.1), is the basic idea for which Hadoop and HDFS were design. HDFS improves the batch processing time through the increased send rate of data to the programming layer.

Portability and *cost-effectiveness* of HDFS are other considerable advantages for which many projects choose it as a storage solution for analytics processing.

HDFS vs. NoSQL solutions

For the purpose of our project, we are interested in a solution that integrates easily with Mahout and can act as a source of data for Mahout algorithms, more exactly the implementation that uses MapReduce jobs.

In general, HBase is the most tightly integrated with the Hadoop eco-system from the solutions we've previously discussed, since it's a layer on top of HDFS. Cassandra has its own distributed file system, Cassandra File System (CFS), which is similar to HDFS and can be used for analytics. It also implements the Hadoop File System API, so the commands are similar.

HBase, as most NoSQL databases we've discussed, has the advantage of random access over the data that is being stored. Hadoop can perform only batch processing and data must be read sequentially from HDFS, which means that each MapReduce job has to pass through the entire dataset - which requires a complexity of $O(n)$. HBase uses hash tables which allow

accessing any data entry in a single unit of time - complexity is *O(1)*. HBase provides fast lookup for larger tables, while HDFS does not support fast lookup of individual records.

The Mahout implementation over the MapReduce framework is designed for batch processing, so the fast random access over data advantage that NoSQL solutions bring doesn't bring much value, since the algorithms have to pass throught the entire dataset anyway. In the same time, for the algorithms we are evaluating, there is no need for fast lookup of individual records, all of them can be read sequentially.

In theory, both HBase and Cassandra can act as source of data for any Hadoop MapReduce jobs. In practice, this is true, but the MapReduce jobs implemented for the data mining algorithms from the Mahout library can read and write data only in HDFS. Additional code can be written to change the data source for these Mahout algorithms, but the integration is not that straight-forward.

MongoDB is a very good candidate for storing tweets, as the JSON-like data model is suited for the structure of the tweets. Each tweet could be considered a document (a JSON object) with the same structure. The issue here is that the main point of interest from the tweet is the text content, which is very small because of the length restrictions. If we were to store these tweets in MongoDB we would have a very large number of very small documents, which might not be the best approach for batch-oriented processing algorithms. Storing them in batches in HDFS files has the potential to improve the processing performance since they are close to each other and reads are sequential.

HDFS and NoSQL databases are different solutions for the persistence layer that can be used in different contexts. HDFS, as the entire Hadoop system, is more of a batch-oriented processing approach for BigData. NoSQL databases are more suited for real-time data processing, since they are optimized for fast reads and writes of data. These two can also be deployed in parallel, in systems where some preliminary results are needed real time and then data can be moved in batches in Hadoop for more precise and complex analytics, like generating recommendations or performing predictive analysis. In our case, in order to do real-time data processing with Mahout, an additional layer over Hadoop would need to be used in order to support optimized reads/writes.

Given the context and particularities of our project, we choose HDFS as the persistence layer solution, because of the easy integration with Mahout

MapReduce jobs and distributed storage across the nodes in the cluster.

Chapter 6

Apache Mahout

The Apache Mahout project is an open source project under the Apache umbrella, which provides a framework for building scalable algorithms and also offers built-in algorithms that can be run on top of Hadoop MapReduce, but also on top of Apache Spark, H2O or Flink. Our main focus will be the MapReduce algorithms that are implemented in Mahout. For testing purposes, these can be run in-memory on a single machine, but for large datasets they need to be executed in a Hadoop environment.

The latest version of Apache Mahout is 0.12.0, starting with April 2016, but the one used in our project is 0.11.0. In latest releases there is a clear shift in focus from Hadoop MapReduce implementations to more comprehensive platforms. From version 0.12.0 the MapReduce clustering algorithms became deprecated and the project seems to be oriented towards Apache Flink, which is a streaming dataflow engine that provides data distribution, communication and fault tolerance for distributed computations over data streams. On a first look, the Apache Flink project integrates stream processing and batch processing and would suit nicely with the purpose of our project. However, being released in April 2016 gives us little time to prepare the necessary infrastructure in order to change the implementation. Despite this, evaluating the performance of Mahout's implementation over the MapReduce framework can provide valuable insights for the great number of projects that already have the necessary infrastructure in place, given the popularity of Hadoop MapReduce.

Apache Mahout provides multiple types of algorithms: recommendations, clustering, classifications and others. The algorithms that are of interest for the purpose of this project are the clustering ones, kMeans and fuzzy kMeans. In order to run these algorithms on the twitter dataset, some additional pre-

processing steps are required and these can be executed using other Mahout algorithms.

In the first section of this chapter we will discuss about converting the raw text stored in HDFS in TF-IDF (Term Frequency - Inverse Document Frequency) vectors, operation which converts our data into a more structured format. In section 6.2, the clustering algorithms used in this project are presented. Another pre-processing step is also discussed, the initial centroid generation, which uses the Canopy algorithm in order to compute a set of points to be used as initial centroids for the clusters computed via kMeans and fuzzy kMeans. Some methods of computing distances between the previously obtained TF-IDF vectors are evaluated and the most suited one for the particularities of our twitter dataset is chosen.

6.1 Data preprocessing

Preprocessing data is a necessary step for all data mining processes and it consists of multiple actions of data manipulation in order to prepare it for applying the actual algorithms. The kMeans and fuzzy kMeans algorithms implemented in Mahout take as input a list of TF-IDF vectors and vectorizing the text content is a pre-processing step for the data mining algorithms. Within this chapter we will use the term document in order to describe a tweet.

The volume of text content found in a digital form is estimated to go well beyond the petabyte range [41] and the problem is that most text content is considered to be in an unstructured format. Human languages can present a serious challenge for interpreting them and extracting the relevant meaning. Several key players in the IT industry are conducting research in this direction. One example would be Apple with Siri, an artificial intelligent robot integrated in their mobile phone that can sustain small conversations and answer basic questions like *How is the weather today?*. It is not the purpose of this thesis to get into the machine learning and artificial intelligence processes that lead to this possibility of sustaining conversation with a computer machine, but the important idea is that bringing structure to text content is important for extracting the meaning.

The purpose of vectorizing documents is to obtain a set of all the words used across the dataset and some measures for determining the most relevant words for each document. A vector space model (VSM) is a common format obtained after this operation. Each word is assigned one dimension in this model and its value is determined by the frequency within a document and across the entire collection. Since the number of English words, text vectors are usually considered to have infinite dimensions. The problem with this model is that is considers words to be independent of each other and the occurrence of one word is non-deterministic when looking at the other words. This is not necessarily true, for example the word York will be associated with New to form New York most of the times, so these words aren't completely independent from each other. There are other models that try to consider word dependencies also, like Latent Semantic Indexing (LSI). Another alternative would be to consider sequence of words, called n-grams, where n is the equal to the number of words in the sequence. Mahout offers support for generating n-grams, passing them through a log-likelihood test to determine if they are relevant or were obtained by change and then apply TF-IDF weighting on the result. Since a tweet can be viewed as a very

short document with few words, it is essential to identify the key-words and n-gram generation wouldn't bring much value to this. This is the reason why throughout this project we will use TF-IDF weighting on single words only.

We will continue discussing which of the actions described in section 2.2 with regards to data preprocessing need to take place for our experiment.

Cleaning the text content of the tweets is the first step of our data preprocessing. Tweet contents may contain hashtags or urls which bring no value to our knowledge of the context of the tweet. These have to be eliminated before further processing the content, because they may lead to inaccurate results. In other words, Salvador Garcia et. all consider that "if a high proportion of data is dirty, applying a data mining process will surely result in an unreliable model" [42]. For example, most urls inside a tweet will start with the string "http://" or "https://", which may cause a text evaluator to consider the url strings similar, even if the context is different. Hashtags also have to be removed, because some of the times a hashtag is formed from multiple words placed together (e.g. *#sunnybarcelona*), which may confuse the text evaluator. One could say that we could configure our text evaluator to look for similarities inside of the words (for example if in our dataset we would have texts that contain *sunny*, *Barcelona* and *#sunnybarcelona*, we could say that *#sunnybarcelona* is similar to both *sunny* and *Barcelona*). The problem is that this could lead to bigger problems regarding irrelevant associations. For example, if we were to have the words allocate and cat, these could end up being associated, even if their related topic is totally different. The best solution for us is to ignore hashtags before applying clustering algorithms over the text content of the tweets. The elimination of hashtags and urls from the tweet text content is an action which in our case takes place at the S4 processing layer, more exactly it is handled by the twitter-processor application.

In order to convert the raw text content to TF-IDF vectors, which can be considered overall as a data transformation step, there are multiple actions that need to be performed. The first one is generating the dataset dictionary, with the Document Processor implemented in Mahout and using the Lucene Standard Analyzer. A Lucene Analyzer is used in order to extract indexable tokens (or words) from texts. There are multiple analyzers provided by the Lucene library: Standard, Whitespace, Stop or Snowball.

The Lucene Standard Analyzer is capable of handling names, email addresses, special characters like punctuation marks. It also contains a default list of stopwords and eliminates them if encountered in the document that is being analyzed. A stopword is a commonly used word which brings no context

to the overall meaning of the document. For example, connection words like “and”, “or”, prepositions, pronouns and so on. One can provide the analyzer with its own set of stopwords or add new words to the default set. This is useful if there is a specific subject treated in the dataset of documents (e.g. if the dataset contains scientific articles, words like *research* don't bring any value regarding the overall meaning of the document, but if the dataset contains Reuters articles, the word *research* can provide some insights regarding the content of the article). The Lucene Standard Analyzer first converts each word to lowercase and then eliminates the stopwords.

The Document Processor from Mahout launches MapReduce jobs over Hadoop. A document tokenization job involves only map actions and at the end it generates the tokenized content (each tweet is associated with a set of tokens or words which are part of that tweet).

After the tokenized documents are obtained, the term-frequency (TF) vectors need to be generated. Term-frequency vectors can be viewed as a map in which the key is a word found in the document and the value is the number of occurrences of that word in the document. The DictionaryVectorizer from the Mahout library takes as arguments the maximum size of ngrams, which in our case will be one, since we want to take into consideration independent words only.

The Document Frequency Convertor from Mahout also launches MapReduce jobs over Hadoop and generates the dictionary of the dataset and the Document-Frequency (DF) Vector. The document frequency vectors contains for each word the number of documents in which the word is present. This is useful in order to select the words that appear in a larger number of documents and to assign them a smaller weight when it comes to clustering, because their meaning doesn't bring many insights to the topic of each document. The number of times a word appears in a document is not taken into consideration when computing the document frequency value. After the DF value is computed for each word, the Inverse Document Frequency (IDF) value is generated, according to the following formula:

$$IDF = \log(\frac{N}{DF}) \tag{6.1}$$

where: N represents the number of documents in the collection,
DF represents the Document-Frequency value previously described

The multiplication with N is used for normalizing the values. This IDF value is used in order to assign smaller weights to more frequent words across the

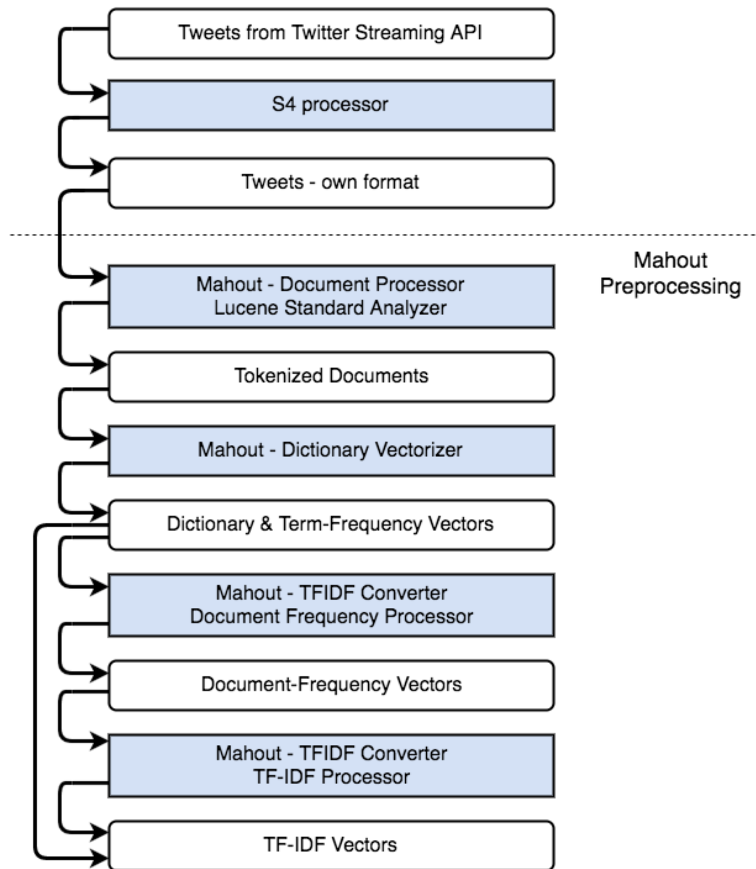


Figure 6.1: Data conversion as part of the preprocessing step

collection. It will not influence however the frequent words inside the same document.

The TF-IDF processor simply computes the TF-IDF values starting with the TF and IDF values, based on the following formula:

$$TD - IDF = TF * IDF = TF * \log(\frac{N}{DF}) \tag{6.2}$$

The preprocessing steps discussed can be observed in the figure 6.1.

6.2 Clustering

Clustering or cluster analysis is the process of grouping a set of items in a manner that similar items are in the same group. Another way of describing it is having items in the same group more similar to each other than to items in other groups. Clustering is a form of unsupervised learning and it focuses on finding some structure in a collection of unlabeled data.

Clustering algorithms can be applied on many different types of data. It can be applied to text documents, based on the similar words found inside or based on their citations. It can also applied on image data, for example in astronomy in order to classify starts. DNA-sequences can also be clustered as part of biomedical research. Structured data, such as shopping history or product descriptions can serve as input for a clustering algorithm in order to extract relevant informations for sales people or for shop bots. All of these (text document, images, DNA-sequences, structured data) are items that can be characterized by a large number of feature vectors or dimensions.

6.2.1 Centroid generation

The clustering algorithms that are used for our experiments, kMeans and fuzzy kMeans, also take as input an initial set of centroids for the clusters. These can be computed using another clustering algorithm implemented in Mahout, Canopy [43]. This is often used as a pre-processing step for kMeans algorithms on large datasets. Datasets can be considered large based on several features: large number of entries in the data, large number of dimensions of the data and large number of clusters that can be derived. The canopy algorithm is proven to reduce the clustering computation time in each of these cases by an order of magnitude with no impact on the accuracy of the results [44]. In our experiments, the dataset can be considered to be large from all these three viewpoints, since there is a large number of tweets retrieved, the number of dimensions is considered infinite since we are dealing with text documents and there is a large number of clusters that can be derived since there are many topics that are being discussed on the Twitter platform.

The Canopy algorithm can use an approximate distance measure method for quickly distributing data across approximate canopies. A canopy is a collection of items that are relatively similar to one another. It is important to note that one item from the dataset can be part of multiple generated canopies. There are two threshold used within this algorithm, T1 and T2, where T1 $>$ T2. If the distance between a canopy (which can be viewed as

the center of a cluster) and an uncategorized point is smaller than T1, the probably that point is part of the canopy, but might be part of others too. If the same distance is smaller than T2, then the point is definetly part of the canopy and there is no need to try to place it in other canopies too. The recommandation is to use an approximate and cheap distance measure to evaluate if the points are at distance smaller than T1 from the canopy and then use a more advanced one to evaluate if that distance is actually smaller than T2. By doing this, most of the distance computations will be done using the cheaper distance measure and then the accuracy of the results can be improved by using a more advanced distance measure for part the second part.

There are multiple distance measures that could be used, we will discuss about each briefly and see which is most suited for our dataset. We will also present the formulas based on which the distances between two items are computed. We will assume that the two items can be represented as vectors in n-dimensional space:

$$v1 = (a_1, a_2, ..., a_n) \tag{6.3}$$

$$v2 = (b_1, b_2, ..., b_n) \tag{6.4}$$

The **Euclidean distance measure** is the equivalent of measuring distance between two points using a ruler. It is computed based on the following formula:

$$d = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2} \tag{6.5}$$

The **Squared Euclidean distance measure** is the square value of the Euclidean one. The formula is:

$$d = (a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2 \tag{6.6}$$

In the **Manhattan distance measure** the distance between any two points is the sum of the absolute difference of their coordinates. The formula based on which this distance is computed is:

$$d = |a_1 - b_1| + |a_2 - b_2)| + ... + |a_n - b_n| \tag{6.7}$$

For the **Cosine distance measure** the items are considered vectors from the origins to the specific coordinates. Between two such vectors, there is a θ angle formed. If the angle is small, the cosine value for that angle is close

to 1. In order to compute the cosine distance, the cosine value of that angle is substracted from 1. This way, we obtain a measure in which a smaller value means that the items are closer to each other. In order to compute this distance we can apply the following formula:

$$d = 1 - \frac{(a_1b_1 + a_2b_2 + ... + a_nb_n)}{\sqrt{(a_1^2 + a_2^2 + ... + a_n^2)}\sqrt{(b_1^2 + b_2^2 + ... + b_n^2)}} \tag{6.8}$$

The **Tanimoto distance measure** combines the Cosine and the Euclidean distance measures. The Cosine distance measures doesn't take into account the length of the vectors, just their direction. The Euclidean distance doesn't take into account the direction in which they point. Both can work well for some datasets, but if both the length and direction are relevant in order to compute the distance between points, the Tanimoto distance is the most suited. The formula for the Tanimoto distance is:

$$d = 1 - \frac{(a_1b_1 + a_2b_2 + ... + a_nb_n)}{\sqrt{(a_1^2 + a_2^2 + ... + a_n^2)}\sqrt{(b_1^2 + b_2^2 + ... + b_n^2)} - (a_1b_1 + a_2b_2 + ... + a_nb_n)} \tag{6.9}$$

For the Euclidean and Manhattan distance measures there is also the possibility to assign weights to each dimension, in order to be able to control which coordinates are more relevant when computing the distance.

For the purpose of this project, since the data can be represented in a n-dimensional space, where n converges to infinity (considering the size of dictionary as the number of dimensions), the cosine distance measure seems the most appropiate.

6.2.2 Mahout kMeans

The kMeans algorithm is one of the most commonly used clusterization algorithms, because of its simplicity. Most implementations of kMeans take as input the following:

- Set of points that are to be distributed into clusters
- Set of initial centroids for the clusters or expected number of clusters (depending on the implementation)
- Distance measure method

- Maximum number of iterations to be performed
- Convergence delta, which is an indicator that the clusters have been identified and no more iterations are needed

The algorithm consists of two steps that are executed multiple times, until the clusters have converged (based on the convergence delta) or until the maximum number of iterations has been reached. It starts using the initial centroids received as input or randomly chooses n centroids. The two steps that are performed repeatedly are:

1. Assign all points to the cluster with the nearest centroid
2. Recompute the centroid for each cluster

The complexity of this algorithm is $O(nktd)$, where n = number of items in the dataset, k = number of clusters, d = number of dimensions for each item and t = number of iterations.

The main advantage for using kMeans is that is fast and easy to understand. It also leads to accurate results when the data points can be grouped into well separated clusters. A disadvantage would be that if the clusters cannot be well defined over the dataset, then the results might not be very accurate, since it allows a point to be part of only one cluster. Another distadvantage is that it requires previous knowledge about the number of clusters that are to be identified, which on large datasets is difficult to estimate and usually these are identified via other faster algorithms (like Canopy). kMeans is also unable to identify noisy data or outliers, precisely because it is based on a static number of clusters.

6.2.3 Mahout Fuzzy-kMeans

The fuzzy kMeans algorithm is an enhancement of kMeans and the main difference is that it allows a data point to be part of multiple clusters. It assigns to each point probability values for being part of every cluster and then the centroids of the clusters are computed based on the positions and the probabilities of each point. Assuming that each point is defined as a vector of coordinates $v_i = (a_1, a_2, a_3, ..., a_n)$ and that the probability values for that point belonging to each cluster are expressed in a matrix where u_{ij} is the probability of the point v_i to belong to cluster c_j, the centroid of each cluster is computed based on the following formula:

$$c_j = \frac{\sum_{i=1}^{n} u_{ij}^m v_i}{\sum_{i=1}^{n} u_{ij}^m} \tag{6.10}$$

where: m represents the accepted level of fuzzyness, m $>$ 1
n represents the number of points in the cluster

After computing new centroids for each cluster, the probabilities matrix is recomputed based on the new centroids.

$$u_{ij} = \frac{1}{\sum_{k=1}^{c} \left(\frac{|x_i - c_j|}{|x_i - c_k|} \right)^{\frac{2}{m-1}}} \tag{6.11}$$

where: c represents the total number of clusters

Chapter 7

Experimental study

For Mahout evaluation, several tests were conducted in order to extract quantifiable metrics. The experiments were conducted over the infrastructure described in section 3.1.

When discussing about the performance evaluation for the Mahout library, we have to consider the following main indicators:

- processing time - the CPU time required for executing the operations
- in-memory usage - the RAM memory usage during execution
- I/O efficiency - the number of bytes read from and written to disk
- algorithmic accuracy - evaluated based on context similarity between top words of each cluster

Other questions we've tried to answer regarding the performance of Mahout throughout these experiments is if Mahout scales with large datasets and if the required processing time increases linearly with the data set size.

Throughout the experiments, we were able to evaluate several steps from the process of clustering data. The three steps that we've evaluated are:

- data preprocessing - converting raw text to tf-idf vectors
- centroid generation - using the Canopy algorithm in order to generate some centroids from the data set to be used by other algorithms
- actual data mining clustering algorithms - kMeans and fuzzy kMeans were chosen, since they seemed the most applicable for the constructed dataset, given its unstructured format.

The actual experiments were conducted by alternating the number of processing nodes in the Hadoop environment and the data set size, in order to be able to see how these two parameters influence the required processing time and in-memory usage. The I/O performance is evaluated by alternating the data set size and observing the read and writes sizes and number of operations. The accuracy is hard to evaluate on large datasets and the nature of our problem, clustering text documents, is rather subjective. We decided to evaluate accuracy on the small datasets and also make a comparison between the kmeans and fuzzy kmeans clustering results.

This chapter is structured as follows:

- Section 7.1 - discussion about the parameters which might influence the behaviour, performance and accuracy of the experiments, for each of the evaluated scenarios
- Section 7.2 - the first experiment based on which we changed the initial approach
- Section 7.3 - experiments with constant number of nodes and variable dataset size
- Section 7.4 - experiments with variable number of nodes and constant dataset size
- Section 7.5 - experiments for I/O performance evaluation
- Section 7.6 - experiment for algorithmic accuracy evaluation

7.1 Parameters

Before discussing about the actual experiments, it is important to present the parameters that can influence the results for each scenario evaluated. Tweaking these parameters can improve or degrade algorithmic accuracy and performance. The full list of parameters for each scenario is provided in the appendix.

Preprocessing The parameters taken as input by the preprocessing step with direct influence over the results are:

- *minimum support for tf-idf vectors* (Integer)
- *number of reducers to use for tf-idf vectors* (Integer)
- *sequential access* - indicates if the output of this step should be written in a SequentialFile format (Boolean)

The *minimum support for tf-idf vector* directly influences the number of dimensions for each point in the dataset, because it establishes if a certain word becomes part of the dictionary of the dataset or not, depending of its number of appearances. Having less dimensions for each point can improve the overall performance because there are less computations, but it can degrade the accuracy because relevant words which provide context might get trimmed away, while common words that do not bring any value to the context of the text become part of the dictionary, because of thier frequent usage.

Centroid generation There are two main parameters for the centroid generation algorithm (Canopy), which are the two thresholds discussed in subsection 6.2.1:

- *T1* - the distance threshold which assigns a point as part of the canopy, but that point can also be part of other canopies
- *T2* - the distance threshold which assigns a point as part of a single canopy

The centroid generation step is configured to use the Cosine distance measure method. This means that T1 and T2 should be greater than 0 and less than 1, because a Cosine distance value cannot be greater than 1.

kMeans For the kMeans algorithm the following parameters are to be taken into consideration:

- Δ = *convergence delta*, used to determine if the clusters have converged (they have not moved a distance greater than this value in the last iteration); the default value is 0.5
- *maximum number of iterations*, this value is independent from the convergence delta and the algorithm stops when reaching this limit even if the clusters have not converged

Fuzzy kMeans The fuzzy kMeans algorithm has similar parameters as the kMeans one. There is also an extra one, the coefficient of normalization or the fuzziness factor (m, as presented in subsection 6.2.3).

7.2 Proof of concept

For the processing time, in-memory usage and I/O operations our first approach was to consider the actual file sizes and start the experiments from 128MB actual disk size of tweets and progressively increase until we reach 32GB, or stop earlier if the performance were to degrade considerably. However, after conducting the first experiment with 256MB and 4 nodes in the Hadoop cluster, we decided that this might not be the best approach. The main observation here is that in 256MB of data, there are almost three million tweets (2.927.916 more precisely) and even if the size of each is very small, the clustering algorithms have to iterate through all the points and compute their distance to the centers of the cluster, given that each point has a great number of dimensions (considering the order of magnitude, the number is higher than thousands). The complexity of these algorithms is influenced by the number of entries, so we decided to take into consideration the number of entries instead of the actual file sizes we used.

In tables 7.1 and 7.15 the results from the first experiment are presented. The experiment used input size of 256MB and four nodes inside the Hadoop cluster, which were four LXC containers deployed on the same physical machine, so there was no network latency.

Table 7.1: Experimental results - 4 nodes, 256MB - Processing time and in-memory usage

Evaluated step	Processing time	In-memory usage
Data preprocessing	73 min	168 GB
Centroid generation	294 min	6.17 GB
kMeans	27.28 hours	2.93 GB

Table 7.2: Experimental results - 4 nodes, 256MB - I/O operations

Evaluated step	Read size	Write size	# read operations	# write operations
Data preprocessing	2.06 GB	1.54 GB	9136	1304
Centroid generation	325.42 MB	33.33 MB	15	2
kMeans	847.42 MB	43.02 MB	8398	8

The data preprocessing step consists of several MapReduce jobs, as there are multiple operations required for transforming raw text into tf-idf vectors, as

described in section 6.1. The processing time presented in our experiments for this step is the sum from all the MapReduce jobs in this step. Since these preprocessing actions must be completed sequentially, as they are not independent from one another, the value for in-memory usage should be chosen as the maximum value from all the preprocessing actions, we believe that this would be a better indicator than the sum of all virtual memory consumed from all jobs. The read and write size as well as the number of operations are computed as a sum for each substep. The kMeans step also consists from multiple MapReduce jobs and the values for processing time and in-memory are presented similarly. Observing the processing time necessary for the kMeans algorithm, the fuzzy kMeans algorithm was not applied on this dataset.

An important note here is that these values are accumulated from all Map or Reduce tasks performed for each of the jobs. Given the fact that some of them were executed in paralel on different nodes, the actual wait time for the results is smaller. The in-memory usage is also computed as a sum of all virtual memory used across all nodes.

Another note is that each map phase consists of multiple actual operations, like record reading, the actual map functions, some local aggregations can be performed optionally and then partitioning the results for the reduce phase. The reduce phase also consists of multiple actual operations. These are shuffle, sort and actual reduce function. The processing time returned by Hadoop is the sum of the processing times of all operations from all tasks.

The parameters for which these values were obtained are as follows:

- Data-preprocessing
  - minimum support = 10
  - sequential access = true
- Centroid generation
  - T1 = 0.9
  - T2 = 0.8
- - convergence delta = 3
  - maximum number of iterations = 10

Based on the obtained results, several issues were discovered. First of all, the minimum support value of 10 is very small taking into consideration the size

of the dataset of almost 3 millions tweets. In order to trim non-English words, abbreviations that are not common or other random strings, the minimum support should be set to some higher value in percentage compared to the size of the dataset. Second, generating the output of tf-idf in sequential files format slows down the kMeans algorithm, based on our experimental results. Besides this, sequential files are not necessary for our project. The T1 threshold for canopies could be set to a higher value, since we are operating with text vectors that have few similarities between them and the associations could be made more loosely.

The Hadoop environment configurations have been modified from their defaults values in order to increase the RAM memory available for map/reduce jobs. More exactly these properties were added in the *mapred-site.xml* file:

```
<property>
         <name>mapreduce.map.memory.mb</name>
         <value>2048</value>
  </property>
  <property>
         <name>mapreduce.reduce.memory.mb</name>
         <value>2048</value>
  </property>
  <property>
         <name>mapred.child.java.opts</name>
         <value>-Xmx2048m</value>
  </property>
<property>
         <name>mapreduce.task.io.sort.mb</name>
         <value>512</value>
  </property>
```

Some properties were also changes in the *yarn-site.xml* file in order to set the number of cores to two for each task:

```
<property>
       <name>
           yarn.scheduler.minimum-allocation-vcores
         </name>
       <value>2</value>
</property>
```

7.3 Experiment #1: Constant number of nodes, different data sizes

7.3.1 Normal datasets

For this experiment we've used the same four nodes as in the one described earlier, because they come with the advantage of being on the same virtual machine and there is no network latency to interfere with the results. The purpose of this experiment is to see how the data size influences the processing time and the in-memory usage.

In table 7.5 we can observe the processing time required for each step, while in table 7.4 the in-memory usage is presented.

Table 7.3: Experimental results - 4 nodes, different data sizes - processing time

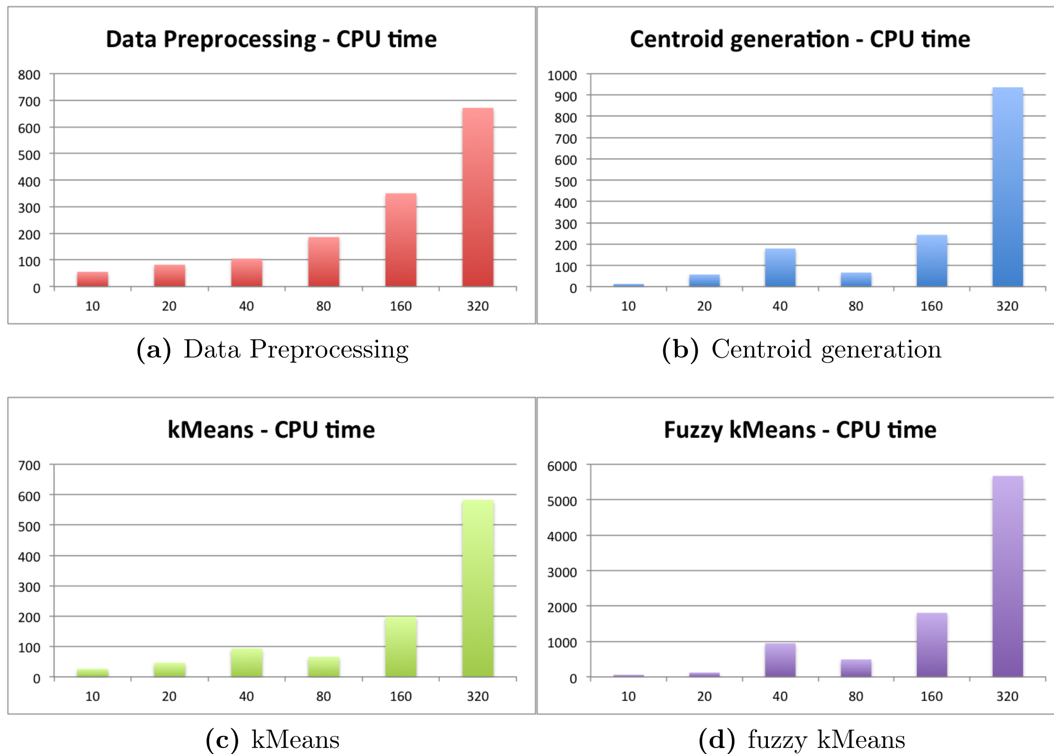
# of tweets	Data preprocessing	Centroid generation	kMeans	Fuzzy kMeans
10k	55.5 sec	12.9 sec	27 sec	1 min
20k	1.37 min	56.7 sec	46.7 sec	2.08 min
40k	1.75 min	2.98 min	1.56 min	15.82 min
80k	3.1 min	1.1 min	1.1 min	8.24 min
160k	5.84 min	4.05 min	3.32 min	30.15 min
320k	11.19 min	15.6 min	9.71 min	94.52 min

This data is also represented in figures 7.1 and 7.2, in order to better observe how the CPU time and in-memory usage scales with the number of entries in the data set. The x-axis represents the thousands of entries in the dataset, while the y-axis represents either the processing time expressed in seconds, either the in-memory usage expressed in GigaBytes. We can see that the CPU time tends to increase non-linearly with the growth of the data set size, while the in-memory usage tends to increase linearly.

We can also observe that there can be outliers, depending on the actual content of the data set. The actual tweets from each experiment are different from each other and this can influence the results we have gathered. Some data sets can be more expensive than others, depending on the actual content. An example from this experiment would be the data set with 40k tweets, where we can observe an unexpected growth in CPU time for the centroid

Table 7.4: Experimental results - 4 nodes, different data sizes - in-memory usage

# of tweets	DP	C	K	F
10k	1.39 GB	1.27 GB	1.36 GB	2.9 GB
20k	1.4 GB	2.68 GB	1.68 GB	3.34 GB
40k	1.72 GB	4.16 GB	2.54 GB	3.61 GB
80k	4.95 GB	3.58 GB	2.58 GB	4.8 GB
160k	13.82 GB	4.3 GB	3.4 GB	4.79 GB
320k	28.83 GB	5.48 GB	4.03 GB	5.02 GB



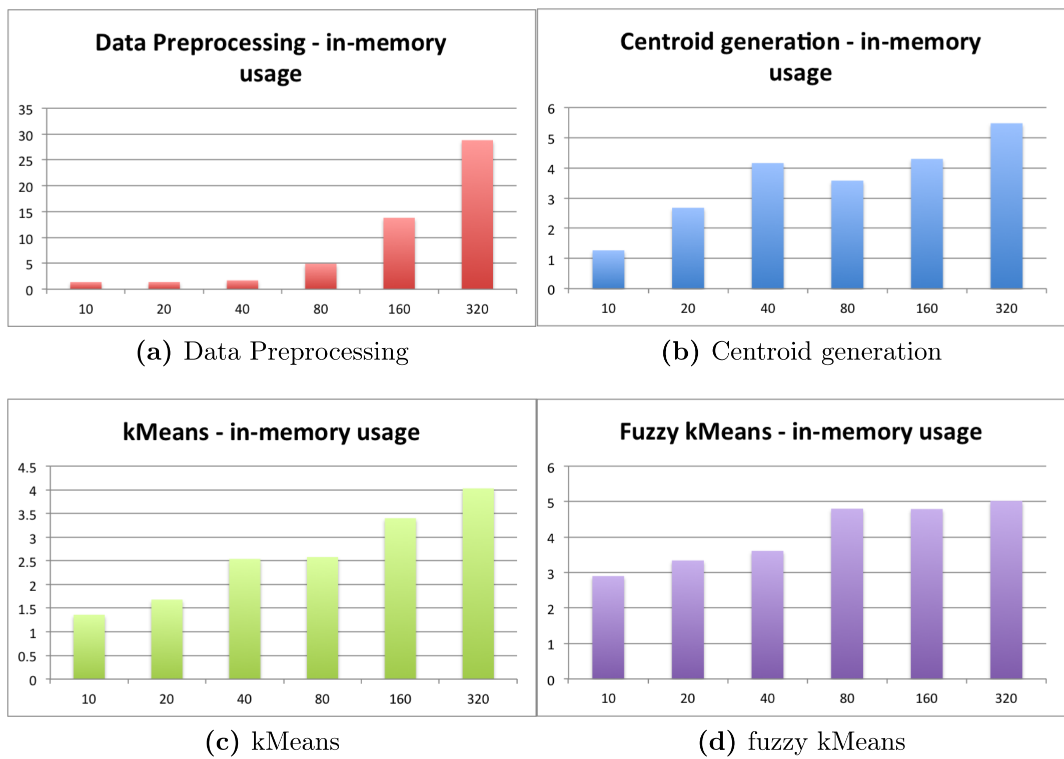
(a) Data Preprocessing

(b) Centroid generation

(c) kMeans

(d) fuzzy kMeans

Figure 7.1: Processing time



(a) Data Preprocessing

(b) Centroid generation

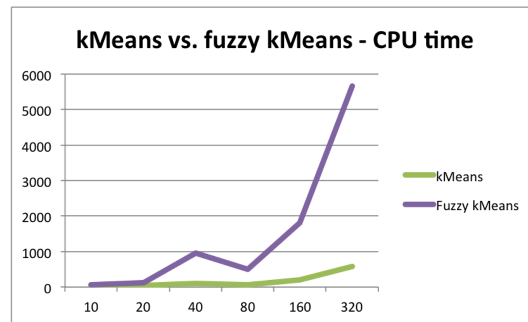
(c) kMeans

(d) fuzzy kMeans

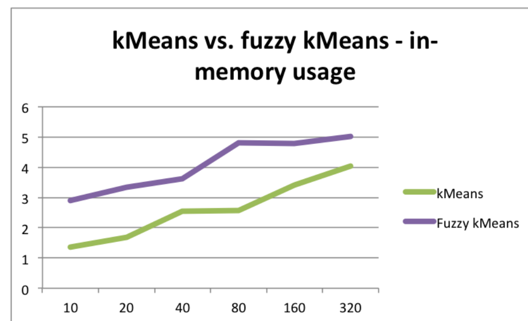
Figure 7.2: In-memory usage

generation, kMeans and fuzzy kMeans steps.

We can also observe that the kMeans algorithm has a faster execution time than fuzzy kMeans and also uses less virtual memory. In figure 7.3 we can better observe these differences.



(a) Processing time



(b) In-memory usage

Figure 7.3: kMeans versus fuzzy kMeans

7.3.2 Larger datasets

Larger datasets were also evaluated throughout this project. In order to obtain reasonable processing time, some parameters were modified compared to the previous experiment. The minimum support for the preprocessing step was increased as the size of the dataset increased. The T1 and T2 parameters from the centroid generation scenario were also modified as follows:

These parameters influence the number of clusters that are to be generated and a higher number of clusters leads to a larger processing time required. We can observe that the processing time for the 640k input size is smaller for almost all phases (except data preprocessing) than for the 320k input size.

Table 7.5: Centroid generation parameters - smaller versus larger datasets

	T1	T2
Smaller datasets (<=320k)	0.9	0.8
Larger datasets (>=640k)	0.99	0.9

Table 7.6: Experimental results - 4 nodes, different large data sets sizes - processing time

# of tweets	Data preprocessing	Centroid generation	kMeans	Fuzzy kMeans
640k	15.52 min	13.58 min	4.80 min	57.72 min
1.28M	32.71 min	13.10 min	18.89 min	2.11 hours
2.56M	63.85 min	43.22 min	26.35 min	3.25 hours
5.12M	2.25 hours	33.64 min	15.89 min	1.71 hours

Table 7.7: Experimental results - 4 nodes, different large data sets sizes - in-memory usage

# of tweets	Data preprocessing	Centroid generation	kMeans	Fuzzy kMeans
640k	36.39 GB	3.99 GB	4.31 GB	5.11 GB
1.28M	86.20 GB	6.21 GB	4.77 GB	4.61 GB
2.56M	169.13 GB	5.84 GB	3.52 GB	4.02 GB
5.12M	797.92 GB	5.08 GB	3.87 GB	4.35 GB

The faster execution time for the 5.12M dataset compared to the 2.56M can be caused by the duplication of some entries in that particular dataset, which makes it easier to clusterize, since identic items are discovered easily and places in the same cluster.

7.4 Experiment #2: Constant data size, different number of nodes

For the next experiment, we've measured the performance of the Mahout library MapReduce jobs with different number of nodes inside the Hadoop cluster. We've used the four nodes as in the previous experiments and added other nodes which were deployed on separate virtual machines, which means that the results were influenced by network latency. Another important note is that the data sets used were not made of the exact same tweets, even though if the number of tweets is the same. The content of the tweets can also influence the results.

The experiment was conducted for 160k and 320k data set sizes. The results can be found in tables 7.8, 7.9, 7.10 and 7.11.

Table 7.8: Experimental results - 160k tweets, different number of nodes - processing time

# of nodes	Data preprocessing	Centroid generation	kMeans	Fuzzy kMeans
4	5.84 min	4.05 min	3.32 min	30.15 min
8	4.54 min	2.83 min	3.16 min	18.65 min
16	7.27 min	3 min	2.56 min	16.08 min
32	4.91 min	2.33 min	2.27 min	18 min

Table 7.9: Experimental results - 160k tweets, different number of nodes - in-memory usage

# of nodes	Data preprocessing	Centroid generation	kMeans	Fuzzy kMeans
4	13.82 GB	4.3 GB	3.4 GB	4.79 GB
8	8.07 GB	4.52 GB	3.24 GB	3.77 GB
16	16.01 GB	4.05 GB	1.87 GB	2.97 GB
32	10.17 GB	3.51 GB	1.78 GB	2.83 GB

We can observe that the processing time tends to decrease as new nodes are added inside the cluster, for all the four steps we've measured. The in-memory usage also tends to decrease as new nodes are added to the cluster, especially for the kMeans and fuzzy kMeans algorithms.

Table 7.10: Experimental results - 320k tweets, different number of nodes - processing time

# of nodes	Data preprocessing	Centroid generation	kMeans	Fuzzy kMeans
4	11.19 min	15.6 min	9.71 min	94.52 min
8	8.85 min	10.83 min	11.42 min	108.61 min
16	9.64 min	8.45 min	7.24 min	57.44 min
32	9.3 min	9.86 min	6.75 min	74.92 min

Table 7.11: Experimental results - 320k tweets, different number of nodes - in-memory usage

# of nodes	Data preprocessing	Centroid generation	kMeans	Fuzzy kMeans
4	28.83 GB	5.48 GB	4.03 GB	5.02 GB
8	18.97 GB	6.35 GB	4.62 GB	4.83 GB
16	21.34 GB	5.58 GB	2.56 GB	4.26 GB
32	20.89	5.73 GB	2.41 GB	4.59 GB

7.5 Experiment #3: I/O performance evaluation

In this experiment we evaluated the number of read/write bytes and the number of read/write operations performed for the same setup as in subsection 7.3.2. We used the same number of nodes and different large data sets and observed how the data set size influences the I/O performance.

Table 7.12: Experimental results - I/O operations - Data preprocessing

# of tweets	Read size	Write size	# read operations	# write operations
640k	384.56 MB	271.24 MB	1820	272
1.28M	765.80 MB	583.78 MB	4284	624
2.56M	1.5 GB	1.05 GB	8320	1200
5.12M	2.78 GB	1.89 GB	15912	2284

Table 7.13: Experimental results - I/O operations - Centroid generation

# of tweets	Read size	Write size	# read operations	# write operations
640k	54.38 MB	11.57 MB	15	2
1.28M	107.78 MB	12.23 MB	15	2
2.56M	212.07 MB	11.74 MB	15	2
5.12M	367.73 MB	1.63 MB	15	2

Table 7.14: Experimental results - I/O operations - kMeans

# of tweets	Read size	Write size	# read operations	# write operations
640k	183.85 MB	3.88 MB	3130	8
1.28M	253.86 MB	4.51 MB	3306	8
2.56M	463.22 MB	5.09 MB	3306	8
5.12M	743.82 MB	1.33 MB	1114	8

Table 7.15: Experimental results - I/O operations - Fuzzy kMeans

# of tweets	Read size	Write size	# read operations	# write operations
640k	166.94 MB	11.58 MB	3130	8
1.28M	277.05 MB	12.23 MB	3306	8
2.56M	482.20 MB	11.75 MB	3306	8
5.12M	743.74 MB	1.63 MB	1114	8

The kMeans and fuzzy kMeans are similar in terms of I/O performance. As expected, the data processing step is the most I/O operations consumer, as it generates a lot of intermediary data before the TF-IDF vectors.

7.6 Experiment #4: Algorithmic accuracy evaluation

The accuracy of the results is hard to evaluate since we are dealing with large data sets. Even for the smallest one we've used, that it contains 10.000 entries, the accuracy is hard to evaluate by just looking at the entries. The indicator for accuracy we've chosen are the top terms from each cluster. We believe that if the top terms from each cluster are connected in meaning from one another, then the cluster is composed of similar tweets. The problem we've encountered is that there are still a lot of common words that don't bring any relevant meaning regarding the topic of the tweet that are not considered stopwords by the Lucene Standard Analyzer. Such words would be "you", "me", "would" and so on. The problem is that they are frequent and influence the top-term results and they can be found in multiple clusters.

For the kMeans algorithm we were able to identify some relevant clusters in the results, where the top-terms are as below:

```
Top Terms:
      best        => 3.716218249551181
      live        =>  1.0115189675627083
      tomorrow    =>  0.8420717962856951
      gates    =>  0.6387904594684469
      kevin    =>  0.6239757866695009
      museum      =>  0.5702077808051274
      bar         =>  0.5502037919800857
      tea         =>  0.5202414085125101
      romper      =>  0.4889811400709481
      museumbar   =>  0.4889811400709481
```

In this example, Kevin Gates is an artist who performed live in Museum Bar.

```
Top Terms:
      finding     =>   3.043842699620631
      dory        =>  2.5815749849591936
      1        =>    2.40844236101423
      see         =>  1.4709655216761999
      going    =>  1.3759261540004186
      nemo        =>  0.9098956430113161
      vs       =>  0.7973385476446772
      money    =>   0.691977308942126
```

```
    kids       =>  0.5816698507829146
    stare    =>  0.4278777977088829
```

In this example, Finding Nemo is an animation movie for children and Dory is a character from that movie.

```
Top Terms:
    dead       =>   3.825956185658773
    tuesday    =>  1.1254327827029758
    left       =>  0.8918552928500705
    shot       =>  0.8233299255371094
    walking    =>  0.6348007122675577
    rock       =>  0.6286250352859497
    cavs       => 0.45017311308119035
    crowd    => 0.43777627415127224
    killer     => 0.43777627415127224
```

In the last example, people wrote on Twitter about the Orlando shooting that took place on the 12th of June of 2016.

We consider these as positive example for clustering. There are also some less relevant clusters in the kMeans output. One example would be:

```
  Top Terms:
    i     =>  0.7902157702524448
    you      =>   0.750680629338172
    my    =>   0.474589636851623
    me    =>  0.3943572662000711
    have     =>  0.3532798025980531
    so    =>  0.3343220786422271
    get      =>  0.2990408585732995
    when     =>  0.2822529022404744
    like     => 0.28091229510646454
    what     => 0.27387167187966044
```

This is an example of a cluster with the most relevant terms being common words that don't bring any value to the topic of the tweets in the cluster. A solution for fixing this would be to construct our own set of stop words and use it in the preprocessing step. Another option would be to eliminate by default words that have a length less than 3 for example, but we might loose some relevant terms, like well-known abbreviations.

The fuzzy kMeans algorithm generated more clusters with common terms, because of the fact that this algorithm allows a tweet to be part of multiple clusters, which means that the most popular one will be in most of the clusters.

Chapter 8

Economic report: planning and costs

In this chapter the main developing tasks identified for this project and their distribution across time are presented in the first section, while in the second one the costs of development and infrastructure are detailed.

8.1 Project planning

In order to achieve the objectives mentioned in section 1.1, the main tasks have been identified and estimations were given for each of them in order to evaluate the necessary time for the development of the project. The estimations were expressed in number of days required to complete them. Other approach would have been to use the Scrum Agile methodology [45] and to identify the main stories and tasks for each story and assign abstract story points to each story and identify the corellation between story points and time intervals. Giving estimations in story points is a better option when the tasks or stories are not very well defined or when estimations in an absolute scale are difficult. For the purpose of this project, even though some unplanned work was to be expected, giving a time estimation for each task seemed more relevant in order to be able to evaluate the progress. In figure 8.1 the Gantt diagram with the initial planification of this project is presented.

The project was started in February 2016, when the Erasmus exchange programme started between UPB and UPC. The data mining oriented research conducted at UPB and the familiarity with the Hadoop environment have

made it possible to finalize it in time.

The first step of the project was the requirement analysis and familiarization with the UPC cluster. The initial idea was to run the Mahout algorithms on top of the Hadoop environment already configured over the RDLab infrastructure. This could have presented some integration issues between Hadoop MapReduce jobs and the persistence layer in which data is stored. A decision was made to develop the project inside an isolated environment on top of the RDLab infrastructure using virtual nodes on top of multiple physical machines.

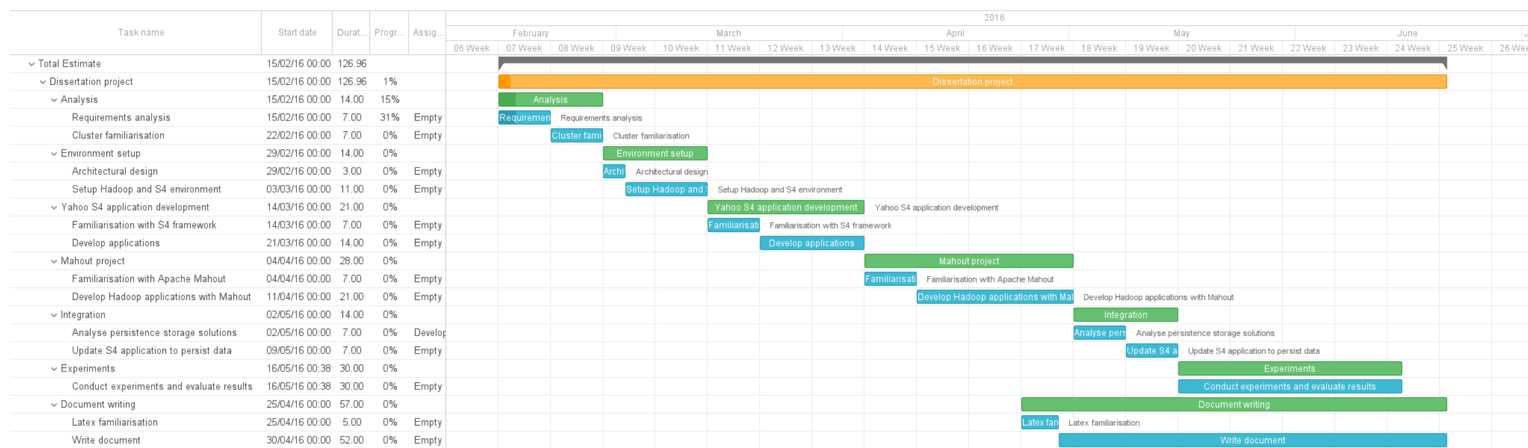
The next step was designing the architecture of the project and setting up the main components, configuring the Hadoop and S4 clusters. The development of the project consists of two phases:

- Development of the stream processing application
- Development of the Hadoop application using the Mahout algorithms

Both of these involved familiarizing with the technologies and developing the actual applications. In order to use the data gathered via the strean processing application with the Mahout MapReduce algorithms an intermediary persistence storage layer had to be configured. The evaluation of multiple storage solutions was necessary. After the decision was made, the S4 application was updated to write data into the new data source from were the Hadoop application would read it.

The final practical step of this project was conducting the experiments and evaluate the Mahout performance in multiple scenarios, with different data sizes and different number of nodes.

The writing of this paper began before finalizing all these research and development tasks, because it required more time and it could be done in parallel.



Task name	Start date	Durat...	Progr...	Assig...
Total Estimate	15/02/16 00:00	126.96		
Dissertation project	15/02/16 00:00	126.96	1%	
Analysis	15/02/16 00:00	14.00	15%	
Requirements analysis	15/02/16 00:00	7.00	31%	Empty
Cluster familiarisation	22/02/16 00:00	7.00	0%	Empty
Environment setup	29/02/16 00:00	14.00	0%	
Architectural design	29/02/16 00:00	3.00	0%	Empty
Setup Hadoop and S4 environment	03/03/16 00:00	11.00	0%	Empty
Yahoo S4 application development	14/03/16 00:00	21.00	0%	
Familiarisation with S4 framework	14/03/16 00:00	7.00	0%	Empty
Develop applications	21/03/16 00:00	14.00	0%	Empty
Mahout project	04/04/16 00:00	28.00	0%	
Familiarisation with Apache Mahout	04/04/16 00:00	7.00	0%	Empty
Develop Hadoop applications with Mahout	11/04/16 00:00	21.00	0%	Empty
Integration	02/05/16 00:00	14.00	0%	
Analyse persistence storage solutions	02/05/16 00:00	7.00	0%	Develop
Update S4 application to persist data	09/05/16 00:00	7.00	0%	Empty
Experiments	16/05/16 00:38	30.00	0%	
Conduct experiments and evaluate results	16/05/16 00:38	30.00	0%	Empty
Document writing	25/04/16 00:00	57.00	0%	
Latex familiarisation	25/04/16 00:00	5.00	0%	Empty
Write document	30/04/16 00:00	52.00	0%	Empty

Figure 8.1: Gantt project diagram

8.2 Costs

The costs of this project are divided between development costs and infrastructural costs, which means both software and hardware.

8.2.1 Development costs

The tasks identified in section 8.1 are mainly engineering tasks, to be performed by a Software Engineer. The requirements analysis could be performed by a Business Analyst, but it could also be performed by the Software Engineer himself. A Software Engineer should not be limited to technical expertise, but it is encouraged to develop a broad skill set, that includes aspects like time management, requirement analysis, communication and technical writing among others. We will consider next that all the identified tasks will be performed by a Software Engineer.

According to the Payscale platform [46], the average salary for an Entry-Level Software Engineer working in Barcelona at the moment of the writing of this thesis is around €2416 per month, for a 8 hours/day programme. Considering this and the estimated time for the development of this project as presented in section 8.1, the development cost would be as follows:

Table 8.1: Development costs

Salary per month	Number of months	Total
€2416	4	€9664

8.2.2 Infrastructural costs

The infrastructural costs are divided between the software and the hardware costs.

In the case of this project, all the software used is freeware, so the costs are zero. A list of the software products used can be found below:

- Apache S4
- Apache Hadoop
- Apache Mahout

- IntelliJ IDEA Community Edition - Java IDE
- texmaker - for Latex document writing

The actual hardware costs of this project were also zero because of the RDLab infrastructure. Next we will discuss the hardware costs in a scenario in which the RDLab infrastrucure would not be available.

There are 34 nodes used in this project and each one of them is configured to use 2GB of RAM and 2 CPU cores. One of the nodes is for the S4 application, in which several virtual nodes are deployed, but they are on the same physical node. The other nodes are part of the Hadoop cluster, but not all of them are required all the time. Four nodes is the minimum required for any experiments and the number increases per need. These nodes could be provided by another IaaS (Infrastructure-as-a-Service) provider. The most popular one is AWS (Amazon Web Services), which offers Amazon EC2 (Elastic Compute Cloud), a web service that provides compute capacity in the cloud that can be resizable, per need. There are multiple well known service providers that rely their infrastructure on Amazon so they can focus on improving their services, without investing much time in hardware maintenance or setup. Such examples would be Netflix, Airbnb or Adobe.

The costs for EC2 instances depends on the required configurations for the instances. In figure 8.2 the pricing for all instances can be observed, as retrieved from the official EC2 pricing page in June 2016.

The configuration of 2 CPU cores and 2 GB of RAM is not offered, but for the purpose of this project the t2.medium configuration could be used, with 2 CPU cores and 4 GB of RAMs, since the in-memory usage is high for the Mahout data mining algorithms implemented over MapReduce. The actual hardware costs estimation rely on the planning estimation in order to see how much time do we actually need the nodes. In table 8.2 the estimated costs are detailed, based on the time estimation for node usage.

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
t2.nano	1	Variable	0.5	EBS Only	$0.0065 per Hour
t2.micro	1	Variable	1	EBS Only	$0.013 per Hour
t2.small	1	Variable	2	EBS Only	$0.026 per Hour
t2.medium	2	Variable	4	EBS Only	$0.052 per Hour
t2.large	2	Variable	8	EBS Only	$0.104 per Hour
m4.large	2	6.5	8	EBS Only	$0.12 per Hour
m4.xlarge	4	13	16	EBS Only	$0.239 per Hour
m4.2xlarge	8	26	32	EBS Only	$0.479 per Hour
m4.4xlarge	16	53.5	64	EBS Only	$0.958 per Hour
m4.10xlarge	40	124.5	160	EBS Only	$2.394 per Hour
m3.medium	1	3	3.75	1 x 4 SSD	$0.067 per Hour
m3.large	2	6.5	7.5	1 x 32 SSD	$0.133 per Hour
m3.xlarge	4	13	15	2 x 40 SSD	$0.266 per Hour
m3.2xlarge	8	26	30	2 x 80 SSD	$0.532 per Hour

Figure 8.2: EC2 instances pricing. *Retrieved from:* *https://aws.amazon.com/ec2/pricing/*

Table 8.2: Hardware costs

Scope of the instance	Estimated time	Cost
S4 node - development	1 month - 8 hours / working day	9.152$
S4 node - tweet gathering	2 weeks	17.472$
4 Hadoop nodes - development	1 month - 8 hours / working day	36.608$
32 Hadoop nodes - experiments	1 month - 8 hours / working day	292.864$
Total	-	356.096$

Chapter 9

Conclusions

Apache Mahout is a powerful tool for automatically extracting relevant information from large quantities of data, where manual processing would be close to impossible. We have chosen Hadoop MapReduce as the framework on top of which to evaluate Mahout, because of its popularity among research projects and enterprise area.

Within this project, we've managed to construct a dataset of tweets gathered via the Twitter Streaming API. We used Apache Yahoo! S4 as a stream processing system. We also evaluated several alternatives for persisting tweets in order to be later read in batches by the Mahout MapReduce algorithms and we have chosen HDFS, as it best meets our requirements. The main objectives, as presented in section 1.1, can be considered acomplished.

We have conducted several experiments and we've calibrated our expectations after the first results. Initially we planned on conducting the experiments over larger datasets, taking into account the actual file sizes that are to be used as input. Giving that in a 256MB file, there are three million tweets and the processing time takes several hours, we have decided to change our mindset and conduct the next experiments focusing on the number of entries in the dataset. We have evaluated the processing time and the in-memory usage for each scenario. We also evaluated I/O performance for different data sizes.

We have also identified some outliers in our results, since we've ran each experiment only once and the results can be influenced by multiple factors, such as network latency or input dataset. A solution for the future would be to run the same experiment over multiple datasets for multiple times and extract the average processing time and in-memory usage.

From the result evaluation, we have come to the conclusion that this type of dataset is not the most suited for the MapReduce implementation of the Mahout algorithms, because the entries are very small, but the size of the problem is exponentially bigger because of the dictionary. So each entry in the data set has a high number of dimensions and for many of them the value is 0, since a tweet is restricted in length and contains few words. A better option of evaluating Mahout on this type of data seem to be the Apache Flink or Apache Spark framework, which offer support for direct stream processing, the tweets would be evaluate as they arrive.

In our experiments we focused on clustering algorithms, kMeans and fuzzy kMeans. It would have been interesting to evaluate other types of algorithms from the Mahout library, such as classification ones. We chose the clustering ones because they seemed the most suited for our dataset. We could also try running such experiments on different datasets.

Regarding the infrastructure of this project, all nodes inside the Hadoop cluster were configured to 2GB of RAM, which proved to be insufficient for running experiments on larger datasets. These would either throw OutOfMemory exceptions or would enter long garbage collector cycles. The particularities of the dataset must be taken into consideration when considering the infrastructure. In this case, the high number of dimensions for each item increases the in-memory usage.

Even though the main purpose of the project was evaluating Mahout performance, we also focused on constructing our own dataset, since this offered us the flexibility of structuring it to fit our needs and also removing unecessary data.

The results of this experiments can provide some estimations about the Mahout performance over the MapReduce framework, for the projects that are considering of using this library for extracting relevant information from their data.

Appendix 1 - User guide

S4 application deployment

```
// Start a ZooKeeper clean instance
./s4 zkServer -clean

// Define two clusters: one for the twitter-adapter app with
   one node and one for the twitter-processor app with two
   nodes
./s4 newCluster -c=cluster1 -nbTasks=2 -flp=12000
./s4 newCluster -c=cluster2 -nbTasks=1 -flp=13000

// Start two nodes for the twitter-processor app
./s4 node -c=cluster1
./s4 node -c=cluster1

// Start one node for the twitter-adapter app
./s4 node -c=cluster2

// Build an deploy the twitter-processor app
./s4 s4r -b=‘pwd‘/test-apps/twitter-processor/build.gradle
   -appClass=org.apache.s4.example.twitter.TwitterProcessorApp
   twitter-processor

./s4 deploy -appName=twitter-processor -c=cluster1
   -s4r=‘pwd‘/test-apps/twitter-processor/build/libs/twitter-processor.s4r

// Build and deploy the twitter-adapter app
./s4 s4r -b=‘pwd‘/test-apps/twitter-adapter/build.gradle
   -appClass=org.apache.s4.example.twitter.TwitterInputAdapter
```

```
    twitter-adapter

./s4 deploy -appName=twitter-adapter -c=cluster2
    -s4r=`pwd`/test-apps/twitter-adapter/build/libs/twitter-adapter.s4r
    -p=s4.adapter.output.stream=RawStatus
```

Mahout operations

Examples for running the mahout jobs for all the four steps discussed can be found below. These example assume that the input path is the root of a folder which has the following structure:

```
root_folder   -> input -> sequence files with tweets content
           -> output -> the results for each step will be
              saved here
           -> metrics -> some execution metrics will be stored
              here, like processing time
```

Preprocessing

```
hadoop jar /path/to/preprocess-jar-with-dependencies.jar
    /hdfs/path/to/folder/to/process minimum_support
    number_of_reducers
```

Centroid generation

```
hadoop jar /path/to/canopygeneration-jar-with-dependencies.jar
    /hdfs/path/to/folder/to/process T1 T2 number_of_nodes
    override_results
```

kMeans

```
hadoop jar /path/to/executekmeans-jar-with-dependencies.jar
   /hdfs/path/to/folder/to/process convergence_delta
   max_iterations number_of_nodes
```

Fuzzy kMeans

```
hadoop jar
   /path/to/executefuzzykmeans-jar-with-dependencies.jar
   /hdfs/path/to/folder/to/process convergence_delta
   max_iterations m number_of_nodes
```

Bibliography

[1] W. Lee, S. J. Stolfo, P. K. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop, and J. Zhang, "Real time data mining-based intrusion detection," in *In DARPA Information Survivability Conference and Exposition II*, 2001, pp. 85–100.

[2] Twitter, "Twitter streaming api," 2016. [Online]. Available: https://dev.twitter.com/streaming/overview

[3] "Internet live stats," 2016. [Online]. Available: http://www.internetlivestats.com/

[4] M. Cox and D. Ellsworth, "Application-controlled demand paging for out-of-core visualization," in *Proceedings of the 8th conference on Visualization '97 (VIS '97)*, R. Yagel and H. Hagen, Eds. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997.

[5] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big data: The next frontier for innovation, competition and productivity," *McKinsey Global Institute*, 2011.

[6] Gartner, "Big data," 2016. [Online]. Available: http://www.gartner.com/it-glossary/big-data/

[7] Oracle, "Big data," 2016. [Online]. Available: https://www.oracle.com/big-data/index.html

[8] Microsoft, "The big bang: How the big data explosion is changing the world," 2013. [Online]. Available: https://news.microsoft.com/2013/02/11/the-big-bang-how-the-big-data-explosion-is-changing-the-world/

[9] J. S. Ward and A. Barker, "Undefined by data: A survey of big data definitions," *arXiv preprint arXiv:1309.5821*, 2013.

[10] P. Raj, A. Raman, D. Nagaraj, and S. Duggirala, *High-Performance Big-Data Analytics. Computing Systems and Approaches*. Springer International Publishing Switzerland, 2015.

[11] D. Llorente, "Processamiento masivo de datos via hadoop," Master's thesis, Facultat d'Informatica de Barcelona, Universitat Politecnica de Catalunya, 2014.

[12] G. Hudges, "How big is 'big data' in healthcare?" 2011. [Online]. Available: http://blogs.sas.com/content/hls/2011/10/21/how-big-is-big-data-in-healthcare/

[13] Y. Aphinyanaphongsa, L. D. Fua, and C. F. Aliferisa, "Identifying unproven cancer treatments on the health web: Addressing accuracy, generalizability and scalability," *IMIA and IOS Press*, 2013.

[14] M. C. Schatz, "High performance computing for dna sequence alignment and assembly," Master's thesis, Faculty of the Graduate School of the University of Maryland, College Park, US, 2010.

[15] B. Xu, J. Gao, and C. Li, "An efficient algorithm for dna fragment assembly in mapreduce," *Biochemical and biophysical research communications*, vol. 426, no. 3, pp. 395–398, 2012.

[16] A. S. Balkir, I. Foster, and A. Rzhetsky, "A distributed look-up architecture for text mining applications using mapreduce," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–11.

[17] D. Preotiuc-Pietro, S. Samangooei, T. Cohn, N. Gibbins, and M. Niranjan, "Trendminer: An architecture for real time analysis of social media text," *International AAAI Conference on Web and Social Media*, 2012. [Online]. Available: https://www.aaai.org/ocs/index.php/ICWSM/ICWSM12/paper/view/4739/5087

[18] E. Jain and S. K. Jain, "Using mahout for clustering similar twitter users. performance evaluation of k-means and its comparison with fuzzy k-means," *5th International Conference on Computer and Communication Technology (ICCCT)*, 2014.

[19] B. Liu, *Web Data Mining, Exploring Hyperlinks, Contents, and Usage Data*. Springer, 2011.

[20] R. Feldman and J. Sanger, *The Text Mining Handbook. Advanced Approaches in Analyzing Unstructured Data.* Cambridge University Press, 2006.

[21] F. Xhafa, V. Naranjo, L. Barolli, and M. Takizawa, "On streaming consistency of big data stream processing in heterogenous clutsers," *2015 18th International Conference on Network-Based Information Systems*, 2015.

[22] F. Xhafa, V. Naranjo, and S. Caballe, "Processing and analytics of big data streams with yahoo!s4," *IEEE 29th International Conference on Advanced Information Networking and Applications*, 2015.

[23] "Rdlab," 2016. [Online]. Available: https://rdlab.cs.upc.edu/index.php/en/

[24] "Apache zookeeper," 2016. [Online]. Available: https://zookeeper.apache.org/

[25] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *2010 IEEE International Conference on Data Mining Workshops*, Dec 2010, pp. 170–177.

[26] F. Xhafa, V. Naranjo, S. Caballe, and L. Barolli, "A software chain approach to big data stream processing and analytics," *9th International Conference on Complex, Intelligent, and Software Intensive Systems*, 2015.

[27] D. Du, *Apache Hive Essentials.* Packt Publishing, 2015.

[28] "Hashtag," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Hashtag

[29] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, pp. 7–. [Online]. Available: http://doi.acm.org/10.1145/343477.343502

[30] "Apache hbase," 2016. [Online]. Available: https://hbase.apache.org/

[31] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings*

of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323

[32] Z. Fong and R. Shroff, "Hydrabase – the evolution of hbase@facebook," 2014. [Online]. Available: https://code.facebook.com/posts/321111638043166/hydrabase-the-evolution-of-hbase-facebook/

[33] Y. Izrailevsky, "Nosql and netflix," 2011. [Online]. Available: http://techblog.netflix.com/2011/01/nosql-at-netflix.html

[34] "Apache cassandra," 2016. [Online]. Available: http://cassandra.apache.org/

[35] "ebay and cassandra," 2016. [Online]. Available: http://www.planetcassandra.org/blog/post/5-minute-c-interview-ebay

[36] "Coursera and cassandra," 2016. [Online]. Available: http://planetcassandra.org/blog/interview/coursera-migrates-to-the-top-of-the-class-moves-to-cassandra-for-an-always-on-on-dema

[37] O. Aladini, "Soundcloud and cassandra," 2016. [Online]. Available: http://planetcassandra.org/blog/soundcloud-activity-feed-and-real-time-stats-powered-by-apache-cassandra/

[38] C. Keller, "Nasa and cassandra," 2016. [Online]. Available: http://planetcassandra.org/blog/5-minute-c-interview-nasa/

[39] "Mongodb," 2016. [Online]. Available: https://www.mongodb.com/

[40] "Adobe and mongodb," 2016. [Online]. Available: https://www.mongodb.com/press/mongodb-delivers-multi-petabyte-data-store-option-adobe-experience-manager

[41] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in Action*. Manning Publications Co., 2011.

[42] S. Garcia, J. Luengo, and F. Herrera, *Data Preprocessing in Data Mining*. Springer International Publishing Switzerland, 2015.

[43] "Canopy algorithm," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Canopy_clustering_algorithm

[44] A. McCallum, K. Nigam, and L. H. Ungar, "Efficient clustering of high-dimensional data sets with application to reference matching," in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '00. New York, NY, USA: ACM, 2000, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/347090.347123

[45] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Agile manifesto," 2001. [Online]. Available: http://www.agilemanifesto.org/

[46] "Payscale," 2016. [Online]. Available: http://payscale.com/