University of Salerno

ADVANCED OPERATING SYSTEM

Clustering and K-Means: a report on parallelization optimization with Apache Spark

Authors:

Cioffi Daniele

Consalvo Mario

Secondulfo Andrea

Professor:

Prof. Giuseppe Cattaneo

Academic Year 2019/2020

Contents

1. Introduction

In everyday life we often and subconsciously gather objects according to a certain resemblance: from clothes in the closet, from food in the shelves to the supermarket or shops, as well as Google groups by keywords all the documents it contains in its huge database.

What we're doing is we're grouping together the objects that we need to relate sooner or later, for some future use or just to find the objects more easily when we need them. This activity, called clustering is one of the most important techniques of grouping non-labeled objects.

Clustering can be performed using different methodologies. One of the most popular is based on the k-means algorithm, which you will see better continuing reading in the article. In addition to k-means, there are of course other clustering algorithms, which sometimes allow to emphasize the architecture used.

The goal of the experimentation is carried out to classify a collection of scientific papers downloaded from ArXiv.org. To achieve that, it is necessary to implement all the pre-processing stages (from raws type-b PDFs to string-based data) required to conduct a k-means algorithm.

For future works, there is the objective of implementing two other different clustering algorithms: spherical and bi-sec k-means applied to document clustering (on the same PDFs dataset). The most common and traditional implementations on the Internet are based on Hadoop, using a MapReduce architecture.

The work we propose represents a next step. It is aimed at using the Apache Spark framework that provides performance up to 100 times higher than more famous paradigms like MapReduce.

2. Preliminary analysis

2.1. Apache Spark

Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since.

Apache Spark has its architectural foundation in the Resilient Distributed Dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. The Dataframe API was released as an abstraction on top of the RDD, followed by the Dataset API.

Spark and its RDDs were developed in 2012 in response to limitations in the MapReduce cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs: MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results on disk. Spark's RDDs function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory.

Spark facilitates the implementation of both iterative algorithms, which visit their data set multiple times in a loop, and interactive/exploratory data analysis, the repeated database-style querying of data. The latency of such applications may be reduced by several orders of magnitude compared to Apache Hadoop MapReduce implementation. Among the class of iterative algorithms are the training algorithms for machine learning systems, which formed the initial impetus for developing Apache Spark.

Apache Spark requires a cluster manager and a distributed storage system. For cluster management, Spark supports standalone, Hadoop YARN, Apache Mesos or Kubernetes. For distributed storage, Spark can interface with a wide variety, including Alluxio, Hadoop Distributed File System (HDFS),MapR File System (MapR-FS),Cassandra, OpenStack Swift, Amazon S3, Kudu, Lustre file system, or a custom solution can be implemented.

2.2. K-means

K Means clustering is a non-hierarchical approach of grouping items into different number of clusters/groups. The number of clusters/groups is defined by the user which he chooses based on his/her use-case and data in question. K-Means works by forming cluster of data points by minimizing the sum of squared distances between the data points and their centroids. A centroid is a central point to a group of data points in the dataset. There are various ways of choosing initial centroid, but in many cases, it is done using random allocation. The algorithm is as follows:

- Firstly, select randomly chosen ‘k’ cluster centroids.
- Cluster Assignment: In this step, assign each of the data points in the dataset to one of the centroids, selecting centroid which is closest to the data-point.
- Centroid Movement: For each centroid, compute the average of all the data-points that are allocated to each centroid. This computed average is the new value of the centroid.
- Calculate the sum of square of distance that each centroid has moved from its previous value, repeat steps 2 and 3 until this value is not less than or equal to threshold value (usually 0.01) or the number of iterations reaches maximum iterations specified, either of which is satisfied.

2.3. Dataset

Having a good amount of data is essential for the best result from the experiment.

ArXiv.org provide more than 1 mln (2017 dataset) of scientific papers, all of them in PDF type-B format and are classified in 8 categories, each category has it owns sub-category.

Files weight goes from 0.5 MB to 3.2 MB so the entire dataset is quite large, but due to bandwidth limitations our work uses only ~50 GB (download the entire dataset could have required more than a month).

2.4. Pre-processing techniques

Before applying any data analysis technique it's necessary to pre-process the collection of documents. In our word we have used the following pre-processing techniques:

- *extract plain text*, the process of extracting manipulated text from a data file in PDF format. This is the first stage of pre-processing and it is therefore essential not to make mistakes.
- *tokenization*, the process of breaking a text into tokens. A token is a non-empty sequence of characters, excluding spaces and punctuation. A token is a non-empty sequence of characters, excluding spaces and punctuation;
- *lowercase conversion* which converts all terms into lower cases;
- *special character removal* which removes all the special characters and digits. This operation allows a substantial reduction of "space" within each document.
- *stop word removal*, removes word that are meaningless, such as articles, conjunctions and preposition. Like the previous operation, this one also allows a reduction of "not important" text within the document.
- *stemming*, namely the process of removing inflectional affixes of words reducing the words to their stem. The goal of stemming is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form.
- *lemmatization*, the process of grouping together the different inflected forms of a word so they can be analyzed as a single item. An algorithm that converts a word to its linguistically correct root is called a lemmatizer. A lemma in morphology is the canonical form of a lexeme.
- *document representation*, after the above listed filters, each document is encoded and stored as a standard vector of term weights.

The preprocessing phase is extremely useful in relation to the next step, which is the clusterization. Making a homogeneous text is crucial for retrieving a good result with KMeans.

2.5. Feature selection

Feature selection, also known as variable selection, attribute selection or variable subset selection, is the process of selecting a subset of relevant features for use in model construction. Depending on if the class label information is required, feature selection can be supervised or unsupervised.

Unsupervised feature selection has been selected for our implementation.

The central premise when using a feature selection technique is that the data contains some features that are either redundant or irrelevant and can thus be removed without incurring much loss of information. Redundant and irrelevant are two distinct notions, since one relevant feature may be redundant in the presence of another relevant feature with which it is strongly correlated.

The process of feature selection ultimately permits the text to be vectorized as described above, in the document representation phase.

2.5.1. Term Frequency

Term frequency (TF) is used in connection with information retrieval and shows how frequently an expression (term, word) occurs in a document.

Term frequency indicates the significance of a particular term within the overall document. This value is often mentioned in the context of inverse document frequency IDF. The term frequency value is consulted, among other things, for the calculation of keyword density.

The function can be broken down into two factors: The first factor of the function is the number of terms present in the document. This number is usually divided by the length of the document itself to avoid giving preference to longer documents.

$$\mathrm{tf_{i,j}} = \frac{n_{i,j}}{|d_j|}$$

where n i,j is the number of i occurrences of j, while the denominator |dj| is simply the size, expressed in number of terms, of the j document.

2.5.2. Term Frequency Inverse Document Frequency

TF-IDF (term frequency-inverse document frequency) is a statistical measure that evaluates how relevant a word is to a document in a collection of documents. This is done by multiplying two metrics: how many times a word appears in a document, and the inverse document frequency of the word across a set of documents.

TF-IDF was invented for document search and information retrieval. It works by increasing proportionally to the number of times a word appears in a document but is offset by the number of documents that contain the word.

So, words that are common in every document, such as this, what, and if, rank low even though they may appear many times, since they don't mean much to that document.

However, if the word Bug appears many times in a document, while not appearing many times in others, it probably means that it's very relevant.

This function indicates the general importance of the term i in the collection:

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : i \in d\}|}$$

where |D| is the number of documents in the collection, while the denominator is the number of documents that containing the term i.

2.6. Techniques applied

Our input set is composed by PDF (type B) documents collected over arXiv. We arrived to collect over 50GB of papers (about a million units) from 8 different categories. We have applied the following pre-processing techniques to our original data set:

- plain text extraction from PDF documents;
- tokenization;
- stop word removal;
- stemming;
- lemmatization;
- searching in vocabulary for synonyms of terms.

The choice of these techniques and the application on the input data, determine a correct operation of the execution, always considering both the load balance (on the cluster) and the determination of the expected results.

It is not excluded, however, the probability of using in the future, the implementation of this project, with a greater amount of input data than that used in this experiment.

2.6.1. Libraries used during the sequential implementation

NLTK Python NLP library

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.

Thanks to a hands-on guide introducing programming fundamentals alongside topics in computational linguistics, plus comprehensive API documentation, NLTK is suitable for linguists, engineers, students, educators, researchers, and industry users alike. NLTK is available for Windows, Mac OS X, and Linux. Best of all, NLTK is a free, open source, community-driven project.

Natural Language Processing with Python provides a practical introduction to programming for language processing. Written by the creators of NLTK, it guides the reader through the fundamentals of writing Python programs, working with corpora, categorizing text, analyzing linguistic structure, and more.

Scikit-learn

Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

2.6.2. Libraries used during the spark implementation

PDFBox

The Apache PDFBox library is an open source Java tool for working with PDF documents. This library to allow creation of new PDF documents, manipulation of existing documents and the ability to extract content from documents.

We used PDFBox to Extract Text: this function to allow extract unicode text from PDF files, to use for next operation.

Spark NLP: State of the Art Natural Language Processing

Spark NLP is a Natural Language Processing library built on top of Apache Spark ML. It provides simple, performant & accurate NLP annotations for machine learning pipelines that scale easily in a distributed environment. Spark NLP comes with 160+ pretrained pipelines and models in more than 20+ languages. It supports state-of-the-art transformers such as BERT, XLNet, ELMO, ALBERT, and Universal Sentence Encoder that can be used seamlessly in a cluster. It also offers Tokenization, Part-of-Speech Tagging, Named Entity Recognition, Dependency Parsing, Spell Checking, Multi-class text classification, Multi-class sentiment analysis, and many more NLP tasks.

This library was very helpful for the steps of:

- tokenization;
- stop word removal;
- stemming;
- lemmatization.

In particular, the ease of use has led us to obtain excellent results without errors.

Apache Spark MLlib

MLlib is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:

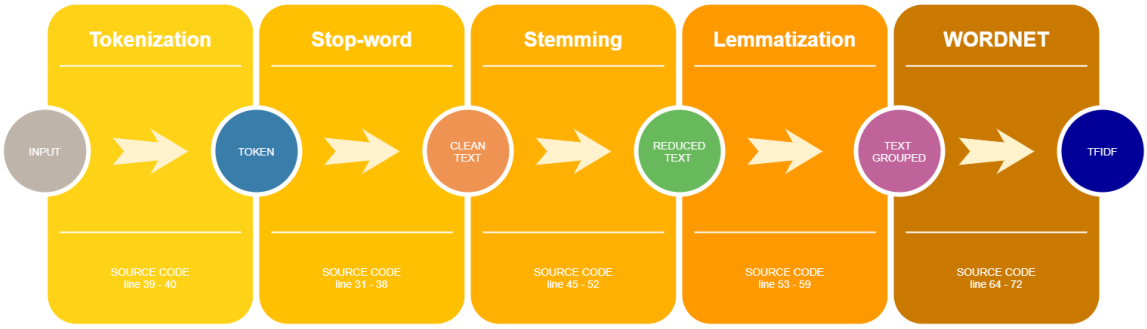
- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- Persistence: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc.

3. Sequential implementation

Our "sequential" program is implemented by defining a pipeline of sequential executions of the processes explained in Chapters 2.1 and 2.4. Starting from our input, the data gradually undergo changes and improvements. The pipeline is developed in such a way that every output of a phase is the input of the next one, and that the correctness of the operation of the process is preserved.

3.1. Sequential steps

The following are the main steps of sequential implementation. This representation serves to facilitate the understanding of the phases that pass out turns and to accentuate the direct dependence between the various phases (pipeline).



3.1. Sequential implementation pipelines

3.2. Source code

Below is the code of the preprocessor.py file written in python that allows the execution of our sequential algorithm. All the code lines are reported so as to provide a complete overview of the various stages of execution.

The various phases described theoretically in the previous chapters have been implemented one after the other. The execution of them will lead in a completely sequential way to obtain the desired output.

```
import os
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
from nltk import pos_tag
from nltk.corpus import wordnet as wn
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import matplotlib.cm as cm

def check_pos(tag):
    if tag.startswith('J'):
        return wn.ADJ
    elif tag.startswith('V'):
        return wn.VERB
    elif tag.startswith('N'):
        return wn.NOUN
    elif tag.startswith('R'):
        return wn.ADV
    else:
        return ''

def preprocessing(line):

    stop_file = open("content/stopwords.txt")

    #stopwords
    stopwords = stop_file.readlines()
    stopwords = [x.strip() for x in stopwords]

    #file retrieving
    read = line

    read = read.lower()
    #tokenization
    tokens = word_tokenize(read)

    #removing stopwords
```

```
    filtered = [w for w in tokens if not w in stopwords]

    #stemming
    porter = PorterStemmer()
    stemmed = []
    for x in filtered:
        stemmed.append(porter.stem(x))

    #pos tagging
    tags = pos_tag(stemmed)
    #Lemmatizing
    wordnet_lemmatizer = WordNetLemmatizer()
    lemmatized = []
    for x in tags:
        v = check_pos(x[1])
        if(v != ''):
            lemmatized.append(wordnet_lemmatizer.lemmatize(x[0], pos=v))

    #real words retrival
    realwords = []
    frequencies = {}
    for x in lemmatized:
        syns = wn.synsets(x)
        if syns:
            realwords.append(x)
            for syn in syns:
                if syn in frequencies:
                    frequencies[syn] += 1
                else:
                    frequencies[syn] = 1

    strreal = (" ".join(realwords))

    return strreal

list = os.listdir("input/")

files = []
for f in list:
    s = open("input/" + f)
    files.append(s.read())
print("i've read the files")

vectorizer = TfidfVectorizer(preprocessor=preprocessing)
vectors = vectorizer.fit_transform(files)
feature_names = vectorizer.get_feature_names()

model = KMeans(n_clusters=8).fit(vectors)
clusters = model.predict(vectors)
order_centroids = model.cluster_centers_.argsort()[:, ::-1]

for i in range(8):
    print("Cluster %d:" % i)
    for ind in order_centroids[i, :10]:
        print(' %s' % feature_names[ind]),
```

```
    print

pca = PCA(n_components=2).fit_transform(vectors.todense())
tsne =
TSNE().fit_transform(PCA(n_components=50).fit_transform(vectors.todense()))

label_subset = [cm.hsv(i/max(clusters)) for i in clusters]

plt.scatter(tsne[:, 0], tsne[:, 1], c=label_subset)

plt.show()
```

3.3. Testing

The execution of the sequential implementation was carried out on a simple laptop (commercial device) owned by us to verify its usefulness and practicality of use.

3.3.1. Test configuration

We describe the testing phase, which took place using a private machine.

The machine used have the following configuration:

- CPU: Intel i5-3320M 2 core 4 thread
- Memory RAM: 8.192 MB LPDDR3
- Hard disk: 256 GB SSD
- Operative System: MacOs 10.13.6

The execution was done locally, and the input files were previously downloaded directly from the source.

3.3.2. Execution Time

We started the sequential test of our program on 30 GB of PDF papers. First, we performed the extraction of the text from the pdf.

The execution took place by launching the following command:

- java -jar pdfpreprocessor.jar <input_dir> <output_dir>

Everything was executed by launching the following command (source code):

- python preprocessor.py <input_dir>

Once we got our text files, we gave them in input, to the sequential implementation of our algorithm for calculating k-means.

Operation	***Time***
PDF to text	3,09 hours
k-means	6,28 hours
Total	9 hours about

3.1. Execution time table

PDF extraction	***Size***
Input size	30 Gb
Output size	1,33 Gb

3.2. Size extraction table

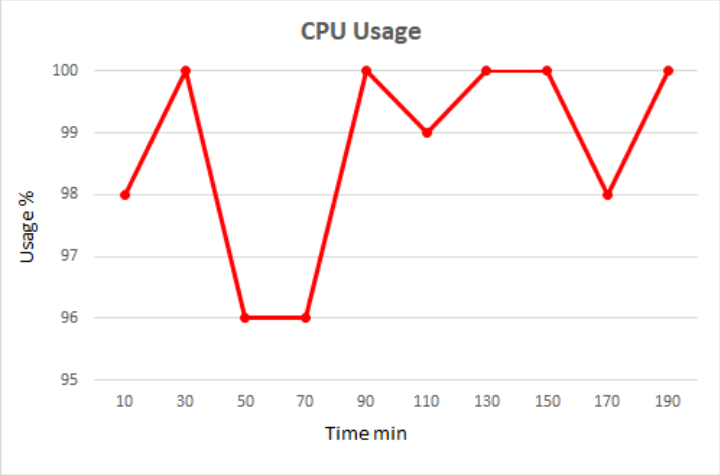
k-means	***Size***
Input size	1,33 Gb
Output size	581 megabytes

3.3. K-means table

3.3.3. Benchmark

In the following graphics we show the use of CPU and memory of our sequential execution.

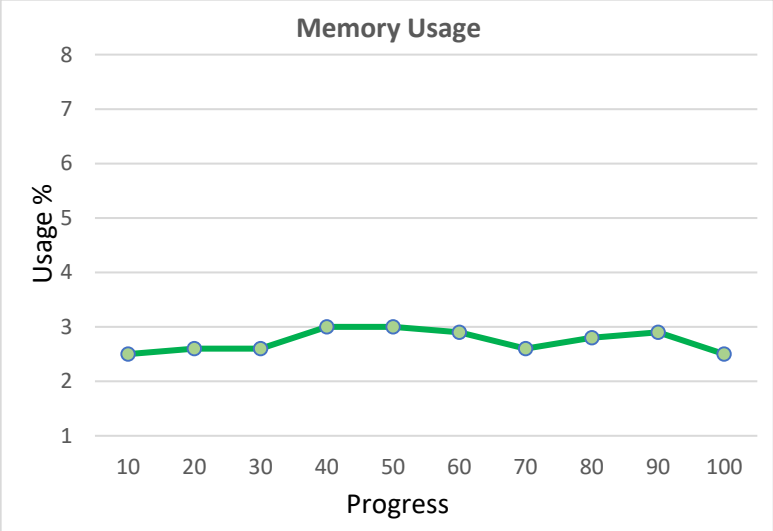
CPU usage for k-means



3.2. Sequential CPU chart

Although the processor used for the computation is quad core, the execution is carried out using only one thread and then only one core. The graph shows the use of the CPU referring to that core.

Memory usage



3.3. Sequential Memory chart

For the execution has been made available all the memory possible minimizing the processes in progress on the system.

3.3.4. Results

The k-means algorithm needs an input parameter representing the number of groups in which to select the data.

We carried out numerous tests providing different k parameters to the k-means algorithm, and we observed that the best result corresponded to the number of categories provided by the dataset.

Keyword extraction

Cluster0	**Cluste1**	**Cluster2**	**Cluster3**	**Cluster4**	**Cluster5**	**Cluster6**	**Cluster7**
al	graph	Phi	Magnet	Theorem	Algebra	Network	Model
Star	Vertex	Quantum	Phi	Proof	Group	Node	Data
Mass	Algorithm	Eq	Spin	Lemma	Theorem	Learn	Time
Stellar	Theorem	Rev	Rev	Function	Map	Train	Function
Model	Set	Mass	Fig	Rn	Proof	User	Fig
Ga	Proof	Field	Filed	Set	Lemma	Model	Algorithm
Cluster	Connect	Model	Phase	Exist	Space	Neural	Method
Ray	Path	Function	Electron	Bound	Subgroup	Algorithm	Number
Redshift	Tree	Quark	Quantum	Problem	Set	Neuron	Al
Fit	Lemma	Term	Band	Space	Action	Data	Set

3.4. Results k-means table

3.4. Implementation problems

It seems obvious to report blocking problems verified during sequential execution. The development of the proposed solution, performed entirely on a single processor, creates an inevitable dependence between the various tasks and to increase the execution time whenever an unexpected delay occurs in a task. In any case, a solution that leads to a parallel execution, where the described tasks can be executed at the same time, seems inevitable also in view of the size of the dataset.

4. Implementation on APACHE SPARK

Due to the problems described above, we found that a parallelization of the execution was the best method to minimize the very high execution time we ran up against during the sequential implementation.

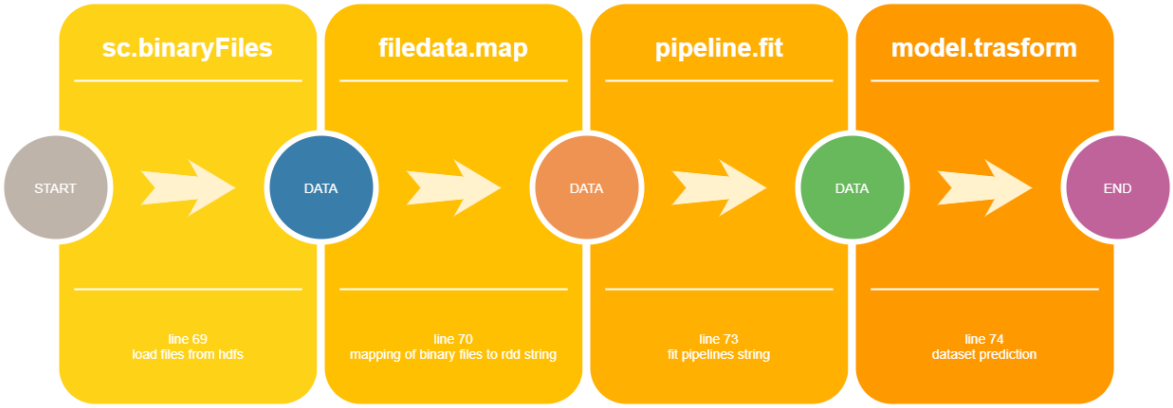
We used Apache Spark as the base framework for the following reasons:

- Spark also allows us to cache the data in memory, which is beneficial in case of iterative algorithms such as those used in machine learning.
- Traditional MapReduce engines are suboptimal for these applications: an application must run as a series of distinct jobs, each of which reads data from stable storage (e.g. a distributed file system) and writes it back to stable storage. They incur significant cost loading the data on each step and writing it back to replicated storage.
- Apache Spark defines the parallelization processes by itself in the best way achievable, so the manual division of the high data volumes isn't necessary.
- Thanks to HDFS and Spark's parallelization capabilities, the program can be executed on multiple machines concurrently, further optimizing data handling and execution times.

4.1. Implementation details

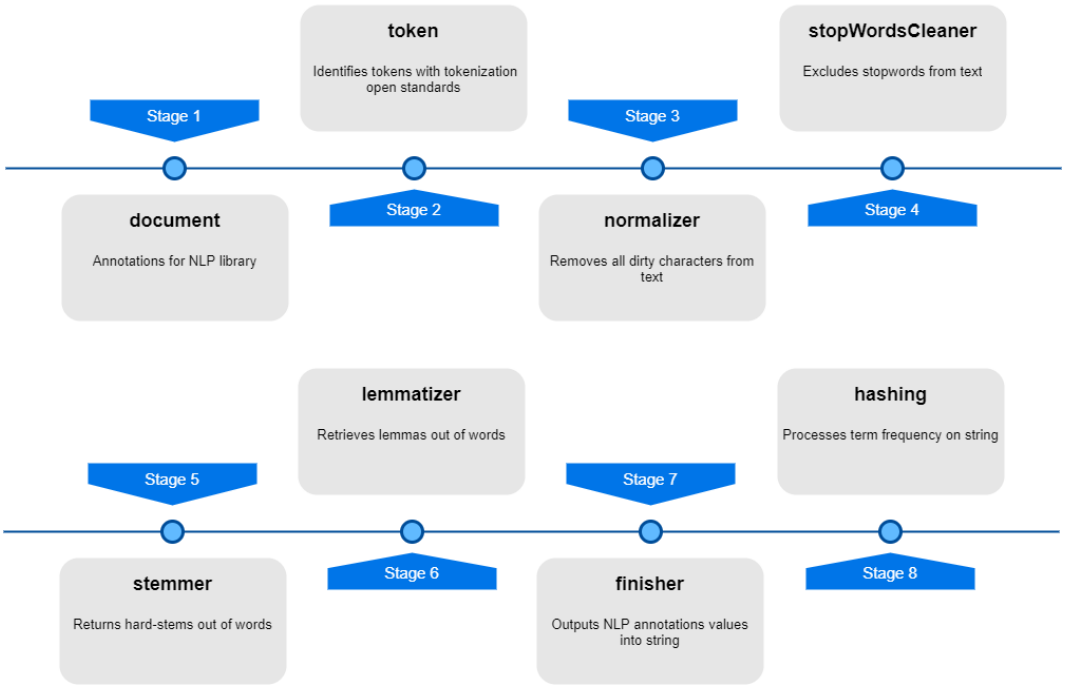
Spark divides the execution into 3 main jobs that are further divided on the cluster machines. Spark automatically optimizes concurrency, avoiding waitings as much as possible.

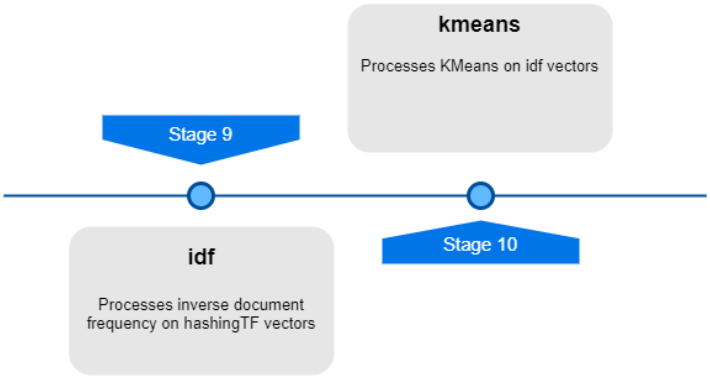
First, the PDFs are retrieved as binaryFiles rdds, then the files are converted into rdd strings through a map function. The rdd strings, converted into a Data Frame, are fitted inside a Spark MLlib Pipeline.



4.1. Parallel steps

ML Pipelines provide a uniform set of high-level APIs built on top of DataFrames that help users create and tune practical machine learning pipelines. MLlib standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline, or workflow.





4.2. Parallel pipeline stages

4.1.1 Why Dataframe?

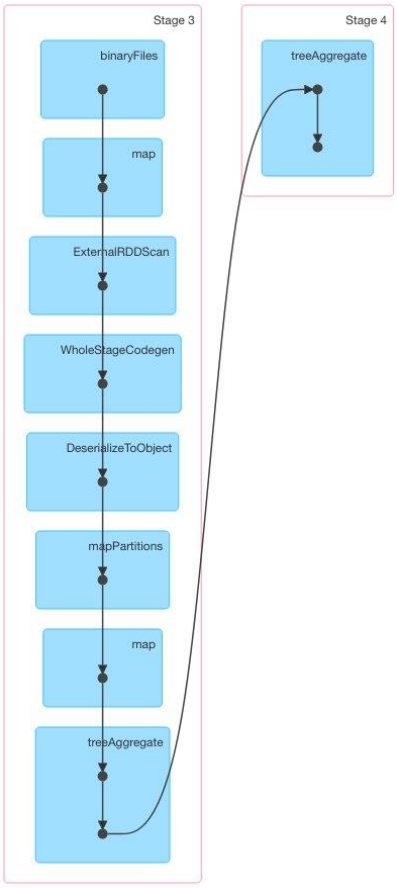
Apache Spark provides to developers APIs either for RDD and DataFrames. RDD was the primary user-facing API in Spark since the 1.0 version. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API. Like an RDD, a DataFrame, introduced in version 1.6, is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier.

Since Apache Spark v. 2.0 DataFrames was adopted by many libraries and becomes the standard also for Apache Spark's libraries as spark.mllib. In our project we used either mllib and Spark NLP which is built on top of spark.mllib (that provide a uniform set of high-level APIs built on top of DataFrames that help users create and tune practical machine learning pipelines). During this study, we have evaluated the use of RDD instead of DataFrames, but due to the fact that all of the libraries that we used are DataFrame-based we have decided to continue with DataFrames. Moreover, as of Spark 2.0, the RDD-based APIs in the spark.mllib package have entered maintenance mode. The primary Machine Learning API for Spark is now the DataFrame-based API in the spark.ml package.

During our tests we have notice some kind of overload on the cluster, maybe caused by use of DataFrame instead of RDD-pairs. To evaluate our solution, we should have implemented a RDD pairs-based software but, due the incompatibilities with the libraries (Spark NLP in particular), we understood it could have taken weeks or months to obtain the same result.

4.2. JOB 1 - Text extraction and data preprocessing

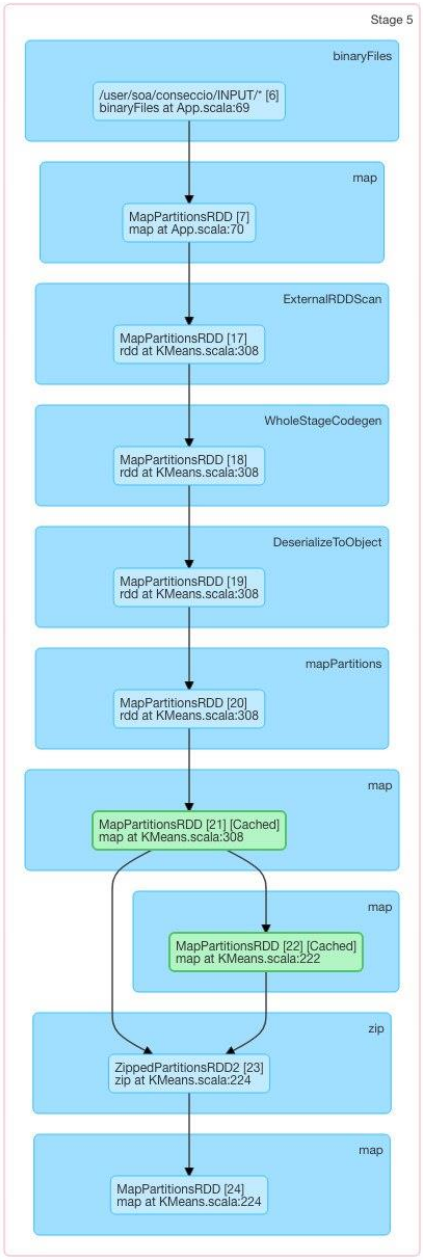
The first main job executed performs the extraction of the text from the Pdfs and executes the pre-processing pipeline. The data obtained from the execution of this job will be used by the next main job. Below is the image of the main job DAG (Directed Acyclic Graph) 1.



4.3. DAG job 1

4.3. JOB 2 - K-means execution

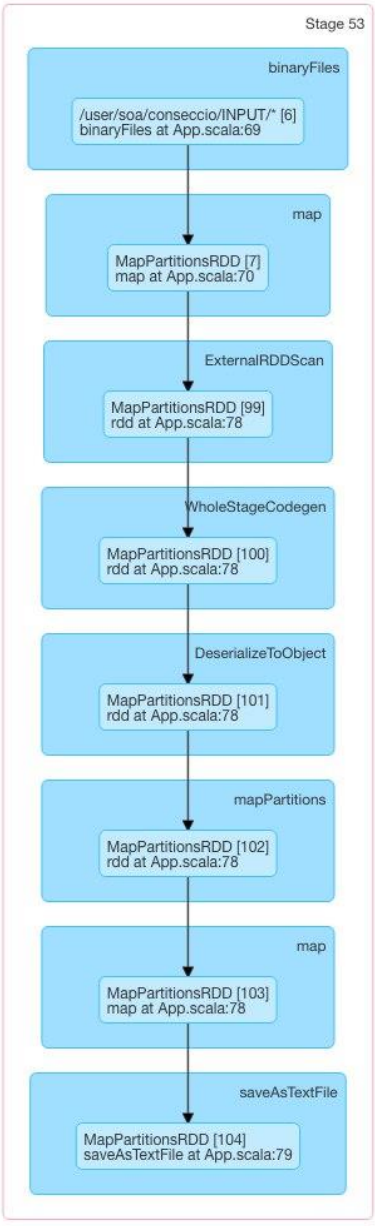
The second main job terminates the pre-processing phase started in main job 1 and effectively performs the k-means. Having obtained our groupings, the execution is ready to carry out the main job 3. Below is the image of the main job DAG 2.



4.4. DAG job 2

4.4. JOB 3 - Export as text

Main job 3 completes the execution of our program by exporting the data obtained from the previous phase as text. Below is the image of the main job DAG 3.



4.5. DAG job 3

4.5. Source code

```
package it.unisa.soa

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.sql._
import org.apache.spark.input.PortableDataStream
import org.apache.spark.ml._
import org.apache.spark.ml.feature.{HashingTF, IDF}
import org.apache.spark.ml.clustering.KMeans
import org.apache.spark.ml.evaluation.ClusteringEvaluator
import org.apache.pdfbox.pdmodel.PDDocument
import org.apache.pdfbox.text.PDFTextStripper
//import org.apache.tika.metadata._
//import org.apache.tika.parser._
//import org.apache.tika.parser.pdf.PDFParser
//import org.apache.tika.sax.WriteOutContentHandler
import com.johnsnowlabs.nlp.base._
import com.johnsnowlabs.nlp.annotator._

import java.io._

val stopwordSourceText = "/user/soa/conseccio/stopwords.txt"
val lemmatizerModelSoruce =
"/user/soa/conseccio/lemma_antbnc_en_2.0.2_2.4_1556480454569"
val outputDestinationFolder = "/user/soa/conseccio/output"

object App {
  def main(args: Array[String]) {
    //val start = System.nanoTime
    // create Spark context with Spark configuration
    val sc = new SparkContext(new SparkConf().setAppName("wordcount"))
    val spark = SparkSession.builder().config(sc.getConf).getOrCreate()
    import spark.implicits._
    val path = args(0)
    //Initialize pipes
    val stopwords = sc.textFile(stopwordSourceText).collect()
    val document = new DocumentAssembler()
      .setInputCol("value")
      .setOutputCol("document")
    val token = new Tokenizer()
      .setInputCols("document")
      .setOutputCol("token")
    val normalizer = new Normalizer()
      .setInputCols("token")
      .setOutputCol("normal")
    val stopWordsCleaner = new StopWordsCleaner()
      .setInputCols("normal")
      .setOutputCol("clean")
      .setStopWords(stopwords)
```

```
51        .setCaseSensitive(false)
52      val stemmer = new Stemmer()
53        .setInputCols("clean")
54        .setOutputCol("stem")
55      val lemmatizer = LemmatizerModel.load(lemmatizerModelSoruce)
56        .setInputCols("stem")
57        .setOutputCol("lem")
58      val finisher = new Finisher()
59        .setInputCols("lem")
60      val hashingTF = new HashingTF()
61        .setInputCol("finished_lem")
62        .setOutputCol("raw_features")
63      val idf = new IDF()
64        .setInputCol("raw_features")
65        .setOutputCol("features")
66      val kmeans = new KMeans()
67        .setK(9)
68        .setSeed(42L)
69        .setFeaturesCol("features")
70        .setPredictionCol("prediction")
71      val pipeline = new Pipeline().setStages(Array(document, token, normalizer,
72 stopWordsCleaner, stemmer, lemmatizer, finisher, hashingTF, idf, kmeans))
73
74      //Execute processing
75      val fileData = sc.binaryFiles(path)
76      val pdfRdd = fileData.map{f => pdfRetrieval(f)}
77      val df = pdfRdd.toDF
78      //Get values
79      val model = pipeline.fit(df)
80      val result = model.transform(df)
81      //Print the predeictions on files
82      val kmeansdframe = result.select("prediction")
83      val pairs = kmeansdframe.rdd.map{row => (row(0))}
84      pairs.saveAsTextFile(outputDestinationFolder)
85    }
86
87    //PDF retrieving
88    def pdfRetrieval(a: (String, PortableDataStream)) : String = {
89      var document : PDDocument = null
90      val pdfTextStripper = new PDFTextStripper
91      var string = ""
92      try {
93        document = PDDocument.load(a._2.toArray)
94        string = pdfTextStripper.getText(document)
95      } catch {
96        case e : Exception => e.printStackTrace
97      }
98      if(document != null) {
99        document.close
100     }
101     return string
102   }

  }
```

5. Test and benchmarking

In this section we want to show the performance results of our performance on SPARK. We want to focus on the basic performance parameters that can provide a general overview of CPU, Memory, disk and network performance. Finally, our goal is to provide a comparison in terms of performance with the results obtained by sequential execution.

The graphs below are generated by Ganglia, a scalable distributed monitoring system for high-performance computing systems implemented on the cluster used.

5.1. First execution

Cluster configuration

The configuration of the cluster used to run our parallel implementation on SPARK has the following configuration:

- 8 slaves
- 7 out of 8 cores per slave
- number of executors 8
- 27,5 over 32 Gb of memory per slave
- bandwidth 1 Gbit

This command was used to launch the execution:

spark-submit --class it.unisa.soa.App --master yarn --deploy-mode cluster --num-executors 8 --executor-cores 7 --executor-memory 27,5G --conf "spark.app.id=wordcount" target/app-1.0-jar-with-dependencies.jar '/user/soa/conseccio/INPUT/*’

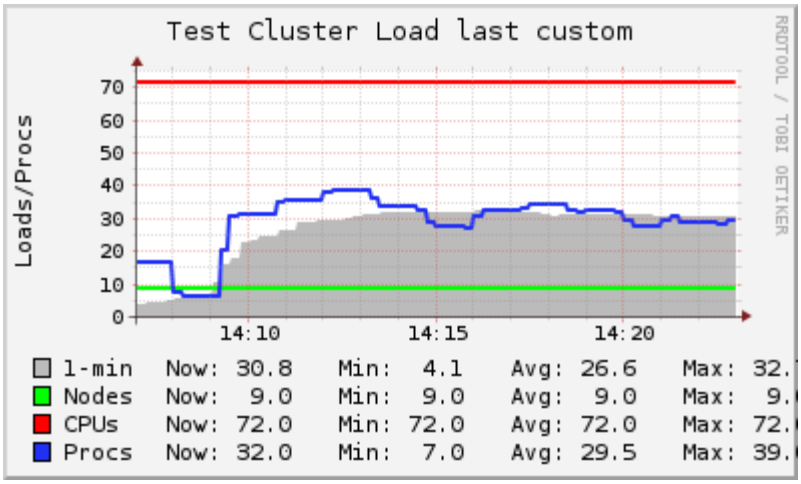
5.1.1. Results

Program execution began at 14:07 and ended after 58 minutes (15:05). The three main Jobs are widely visible from the 3 hills created in a graphical way lasting about 15 minutes each.

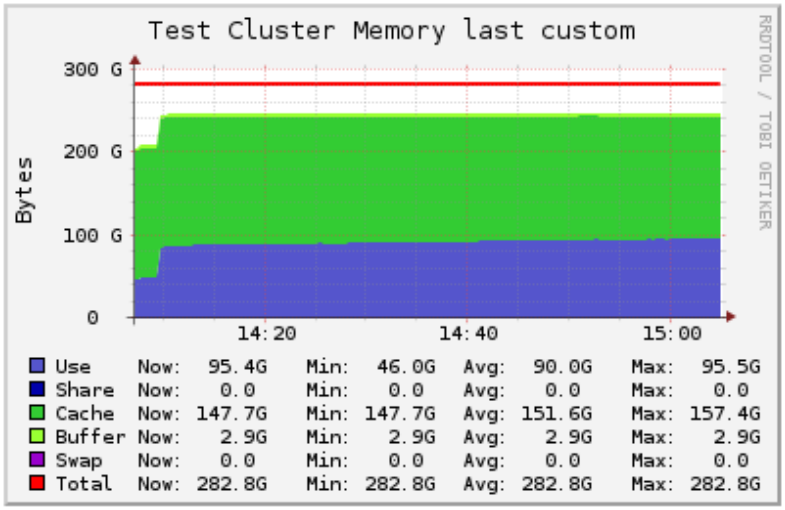
Time	*Job1*	*Job 2*	*Job 3*
58 minutes	20 minutes	20 minutes	18 minutes

5.1. Execution parallel table - 1

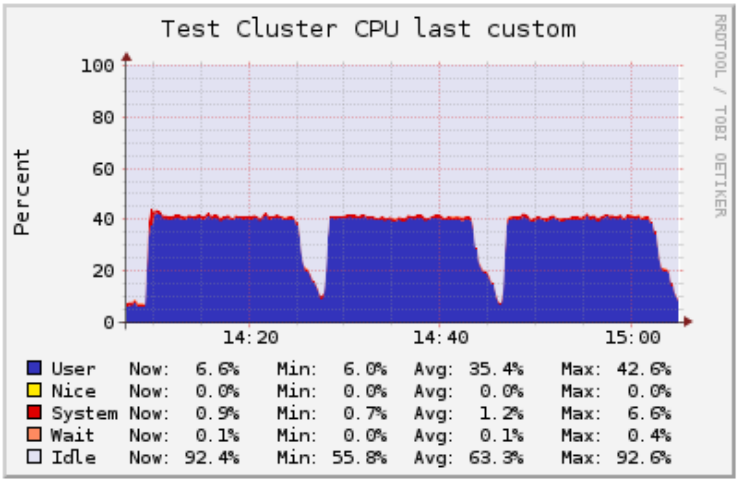
The following graphs are related to cluster usage. These graphs tend to show how and how much the cluster used comes during our execution and how our program makes the most of the features and performance of it.



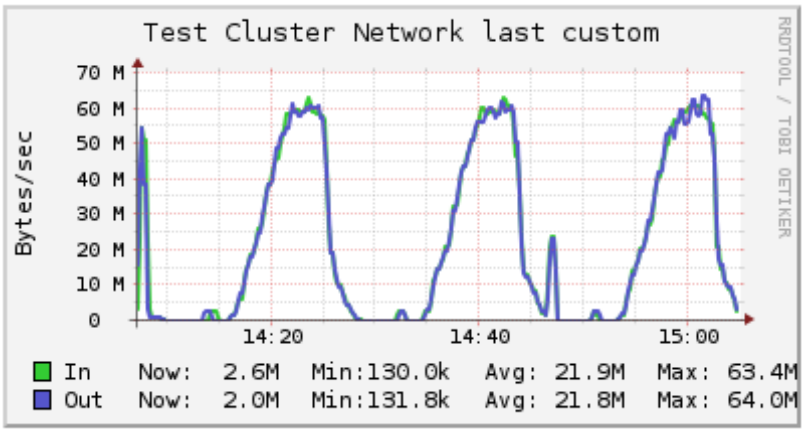
5.1. Memory loads cluster - 1



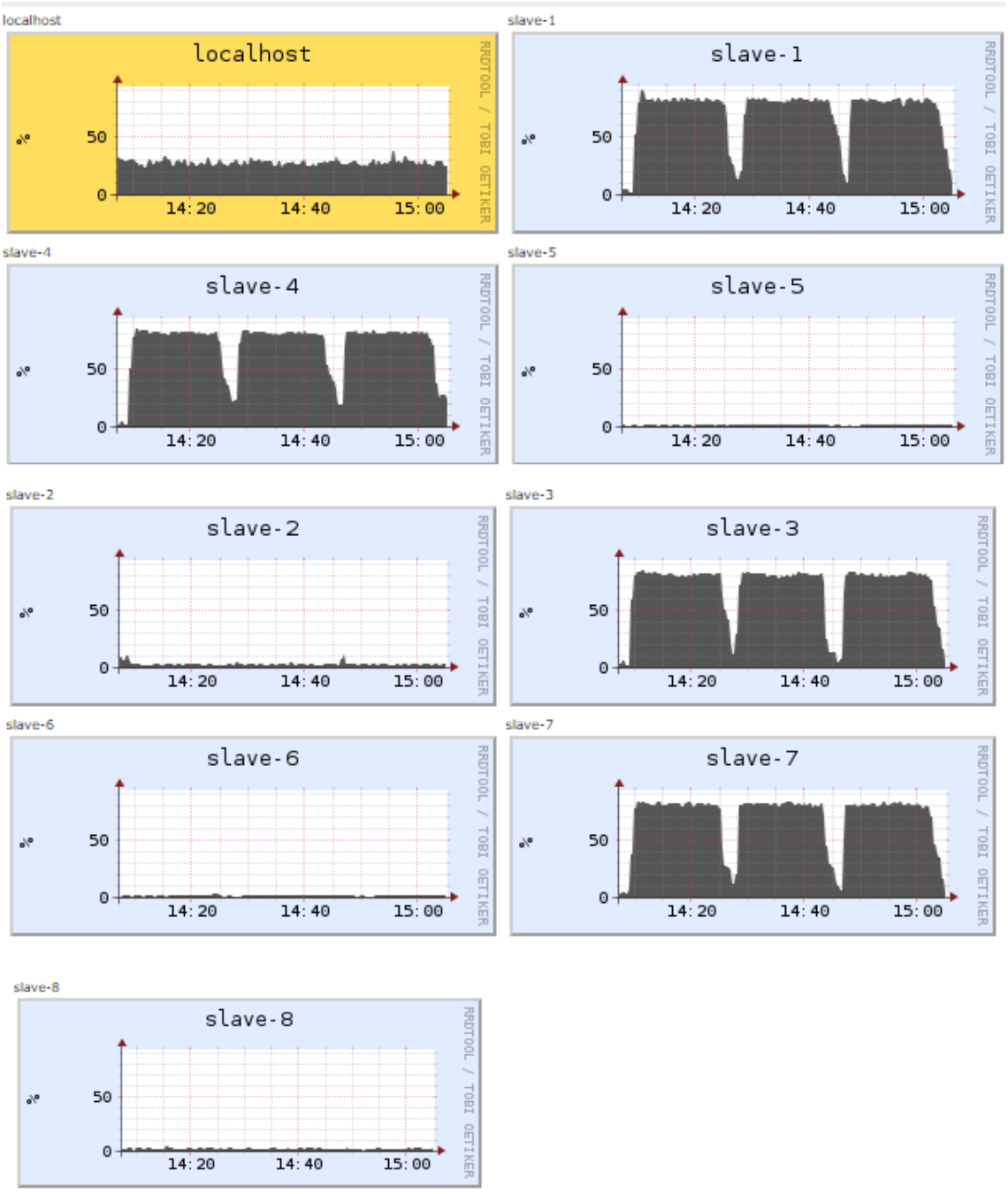
5.2. Memory usage cluster - 1



5.3. Network usage cluster - 1



5.4.CPU usage -1



5.5.CPU usage nodes - 1

5.1.2. Comments

The results of the execution of our program with an input of 40 Gb with the configuration described in chapter 5.1 detected an anomaly: the number of executors in running was 4 out of 8.

We can imagine that this depends on the size of the input (too small for this configuration).

For this reason, we considered it necessary to perform a second execution of our program, always with an input of 40 Gb, but with a different configuration. Our choice, based on tests carried out, fell on memory reduction from 27.5 to 16 Gb.

5.2. Second execution

Cluster configuration

The configuration of the cluster used to run our parallel implementation on SPARK has the following configuration:

- 8 slaves
- 7 out of 8 cores per slave
- number of executors 8
- 16 over 32 Gb of memory per slave
- bandwidth 1 Gbit

This command was used to launch the execution:

spark-submit --class it.unisa.soa.App --master yarn --deploy-mode cluster --num-executors 8 --executor-cores 7 --executor-memory 16G --conf "spark.app.id=wordcount" target/app-1.0-jar-with-dependencies.jar '/user/soa/conseccio/INPUT/*’

5.2.1. Results

Program execution began at 8:55 and ended after 42 minutes (9:37). The three main Jobs are widely visible from the 3 hills created in a graphical way lasting about 15 minutes each.

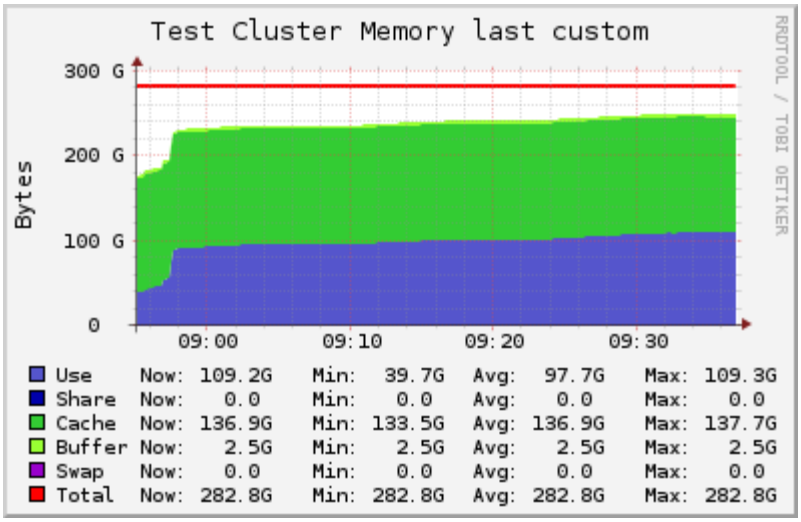
Time	***Job1***	***Job 2***	***Job 3***
42 minutes	15 minutes	12 minutes	15 minutes

5.2. Execution parallel table - 2

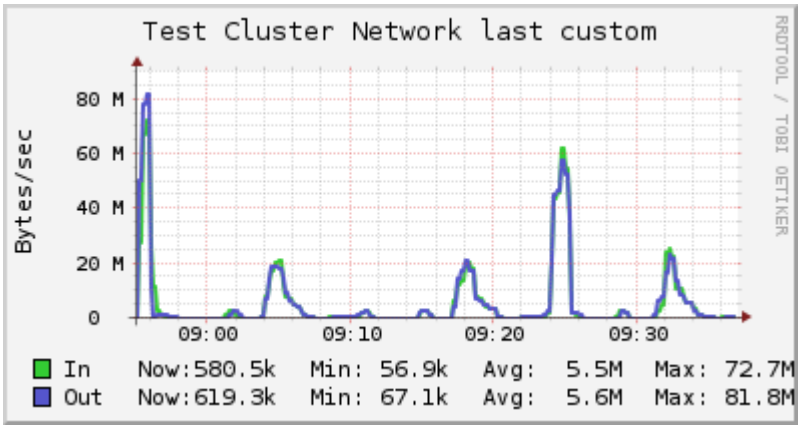
The following graphs are related to cluster usage. These graphs tend to show how and how much the cluster used comes during our execution and how our program makes the most of the features and performance of it.

From the graphs of the execution are clearly visible hillocks. They are since the slaves, despite having the same workload and having the same characteristics, do not finish the jobs at the same time, and a short waiting phase occurs (of time influencing with the total performance).

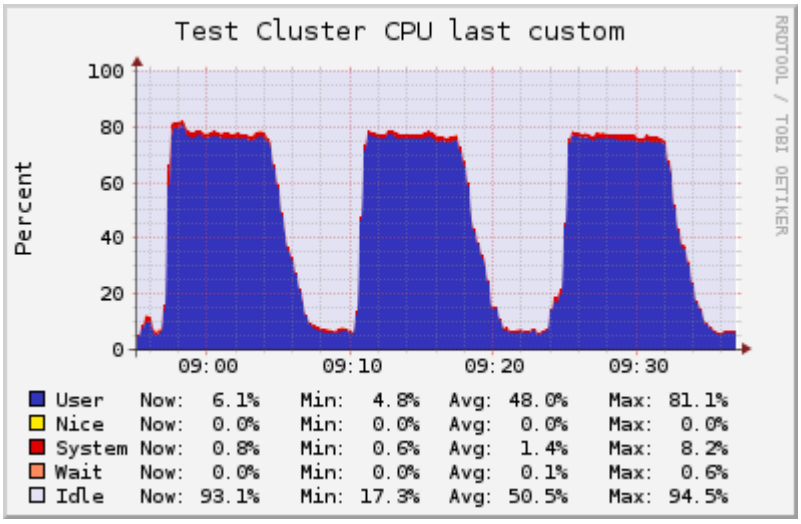
General graph



5.6. Memory usage cluster - 2



5.7. Network usage cluster - 2

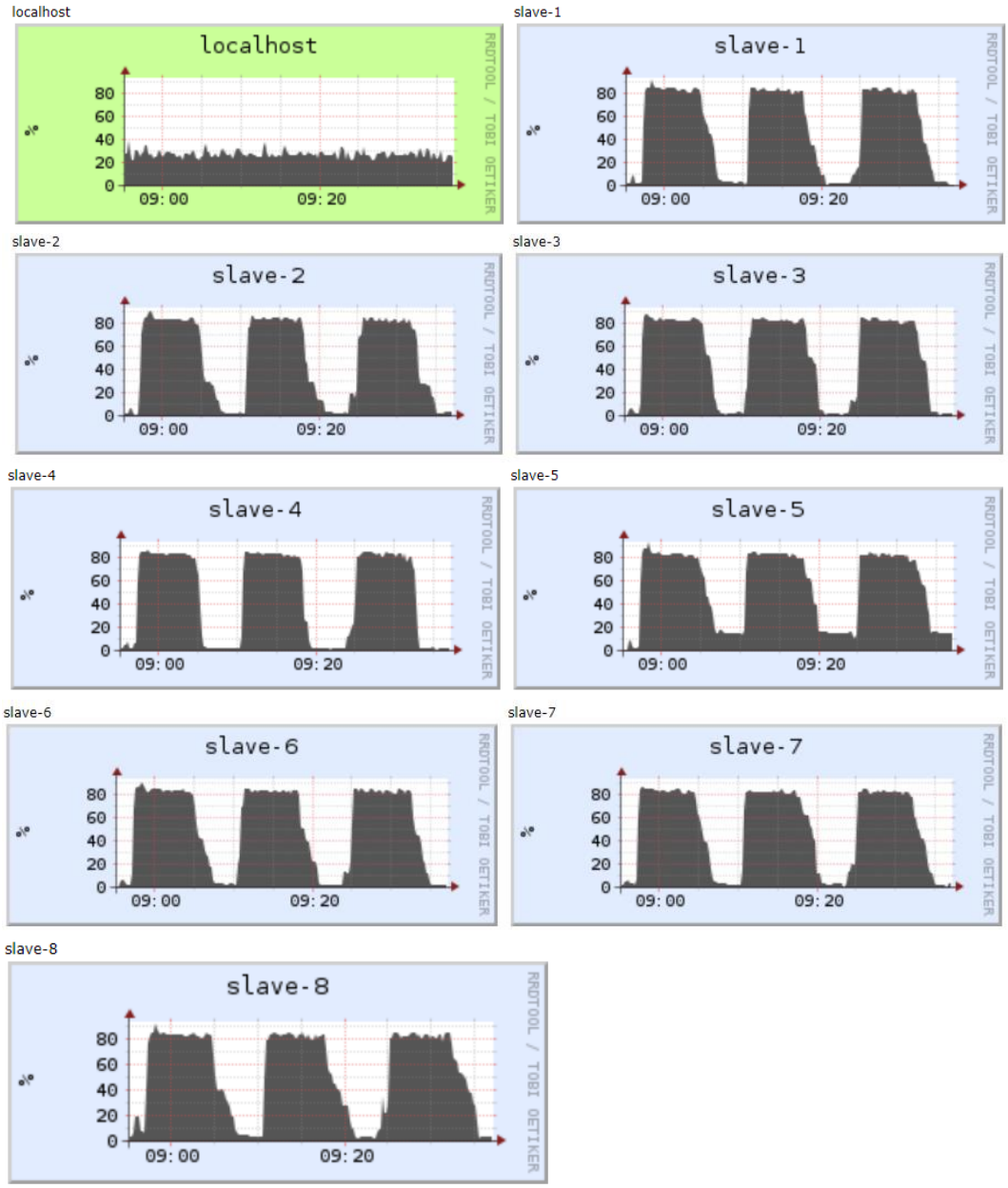


5.8.CPU usage - 2

5.3. CPU

CPU usage is constantly around 80 % for each slave. This data confirms our hypothesis regarding a correct division of the workload between the various nodes and a correct parallelization of the elaboration, decreasing to the maximum the dependencies between the various tasks.

View of all the nodes

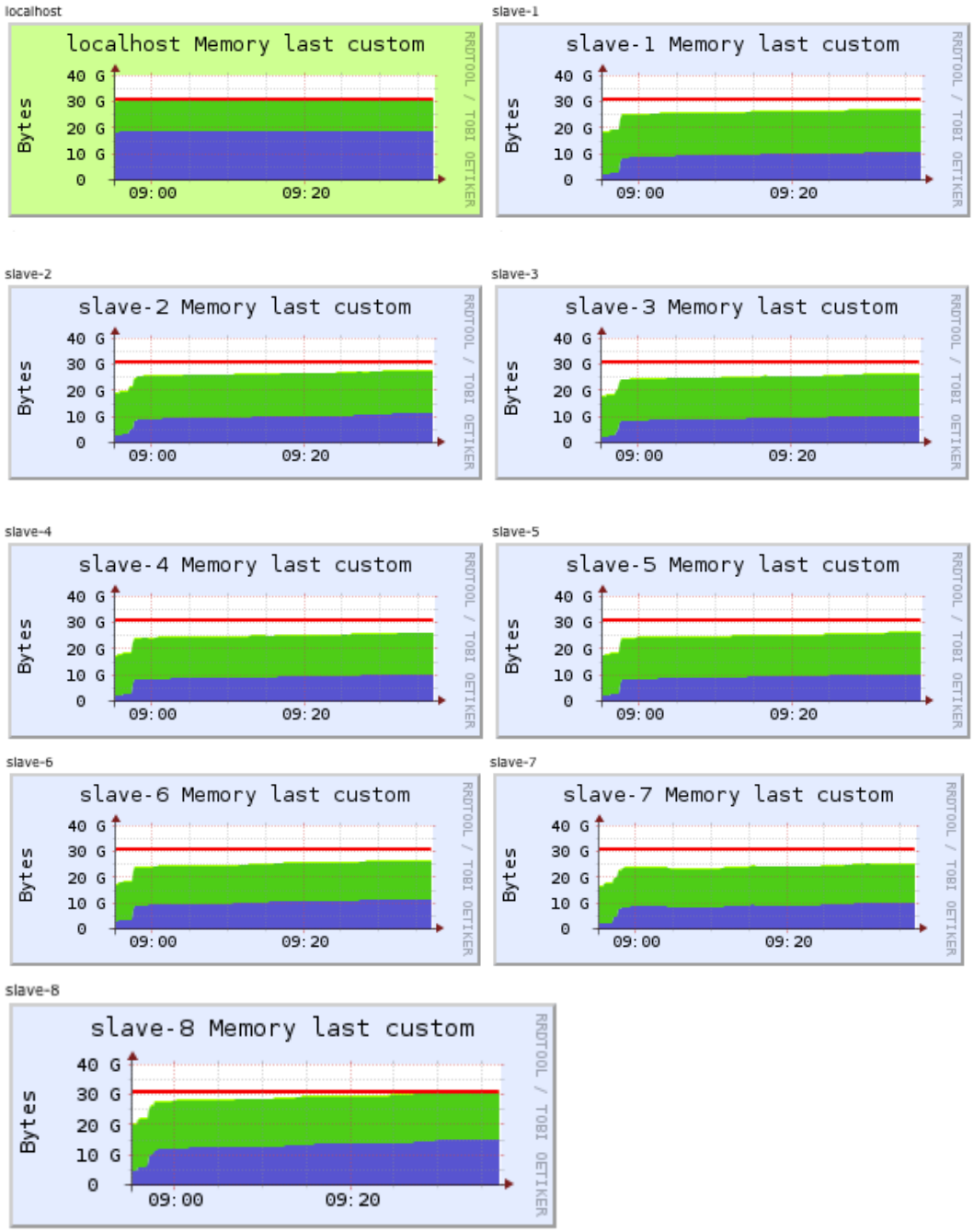


5.9. CPU usage nodes

5.4. Memory

The memory usage appears constant around 10 Gb for each node validating a proper load distribution.

View of all the nodes

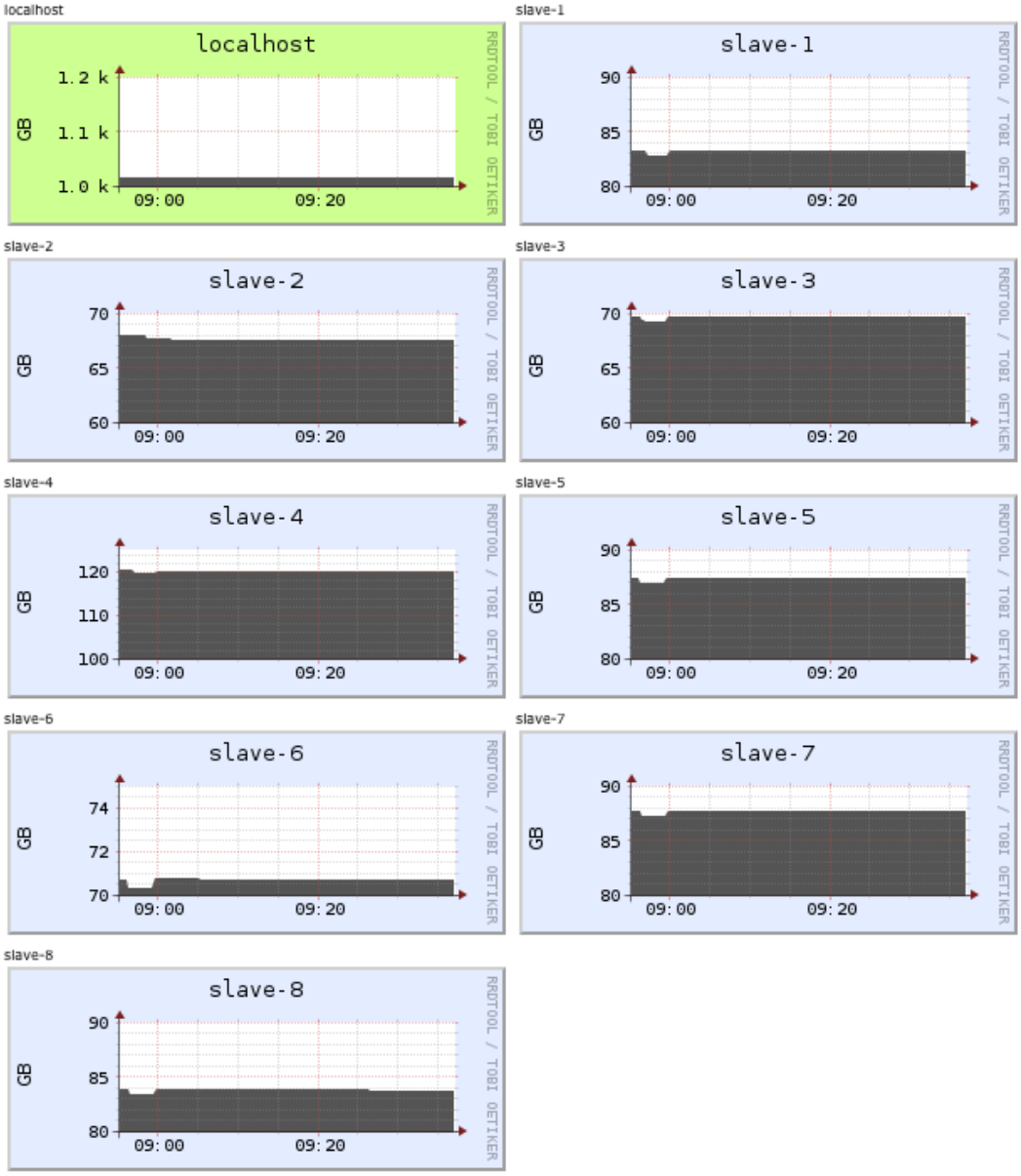


5.10. Memory usage nodes

5.5. Disk

The following are the charts regarding disk usage. These graphs are "inverse" in how much the zone delineated from the diagram represents the "free" memory not used and consequently that one used will be the white part (empty). You may notice some anomalies on some nodes: a greater use by some nodes depends on the excessive traffic on the network that slows down the transport of data from disk to memory.

View of all the nodes

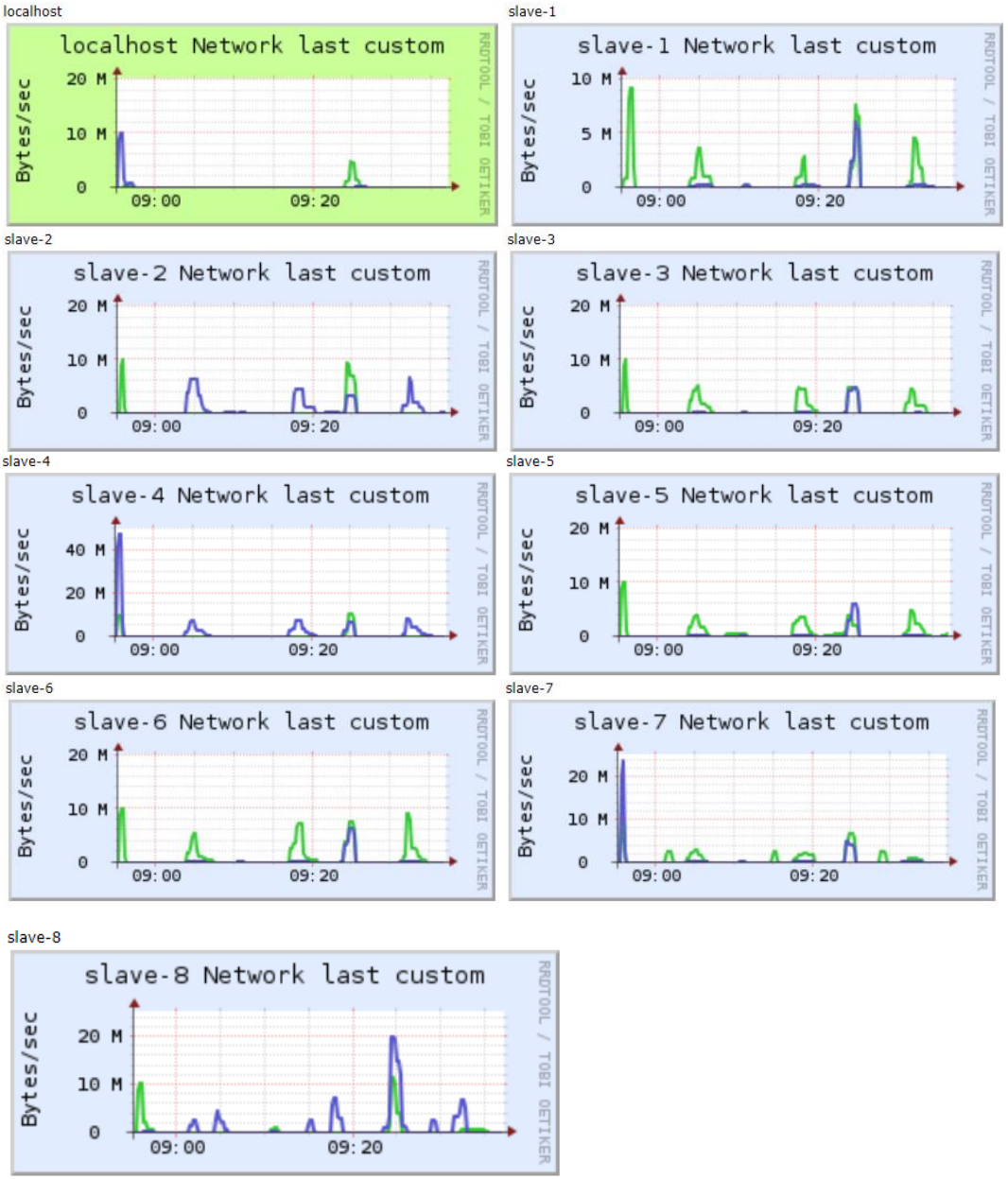


5.11. Disk usage nodes

5.6. Network

The following graphs show the network usage of each node. Communication between nodes rarely occurs while running a main job. The largest communication is at the end of a main job with the communication of the data related to the process just finished with the master node.

View of all the nodes



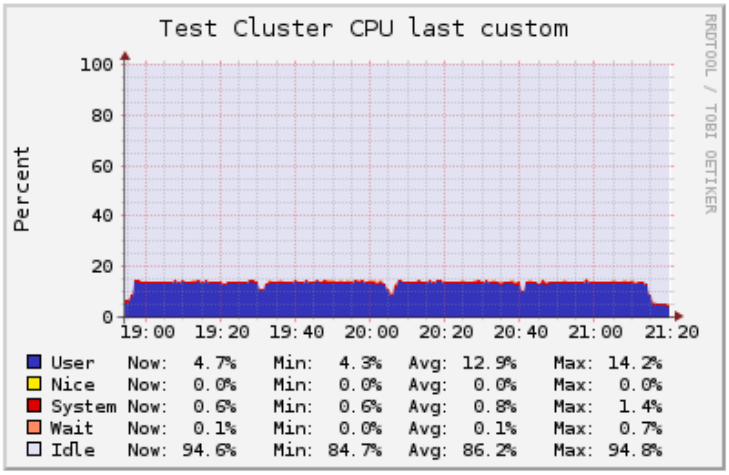
5.12. Network usage nodes

5.7. Different execution

To have an even wider view of the efficiency of our distributed implementation, we have tried other tests. For each further execution, we have modified the number of executors (starting from 1) while remaining fixed the input size (40 Gb) and 16 Gb of memory.

Executors	*Time*
1	2 hours 29 minutes

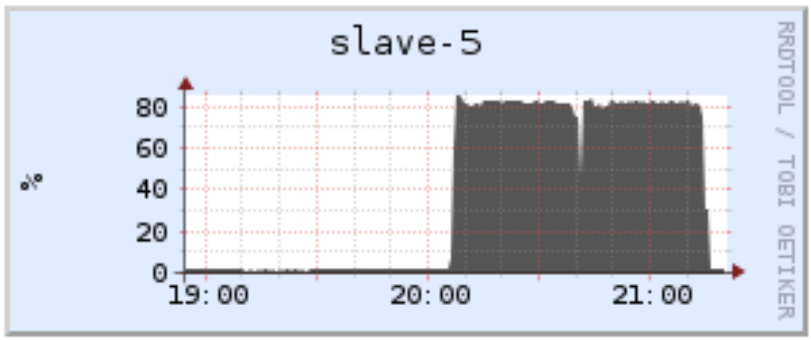
5.3. Executors table – 1



5.13. CPU usage cluster – 1 ex

slave-3

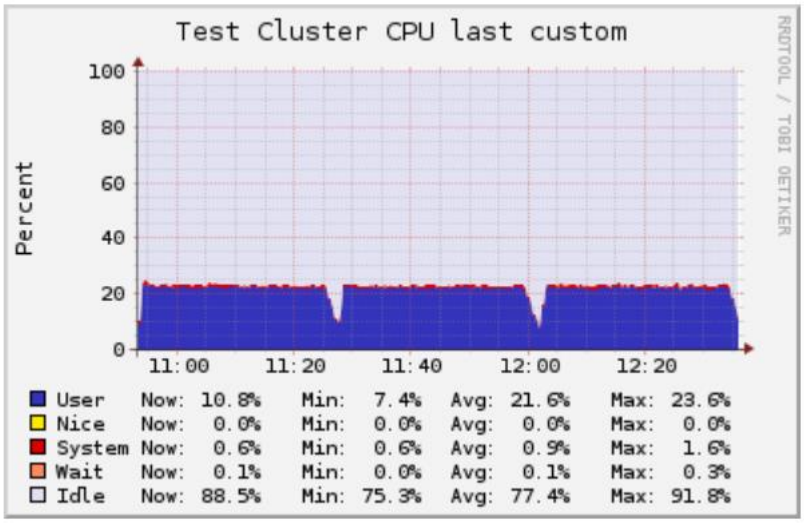
slave-5



5.14. CPU usage node – 1 ex

Executors	*Time*
2	1 hours 43 minutes

5.4. Executors table – 2



5.15. CPU usage cluster – 2 ex

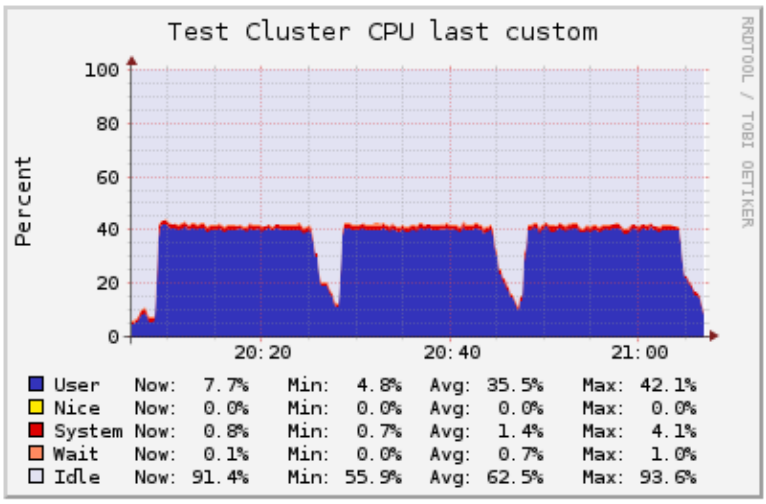
slave-1

slave-5

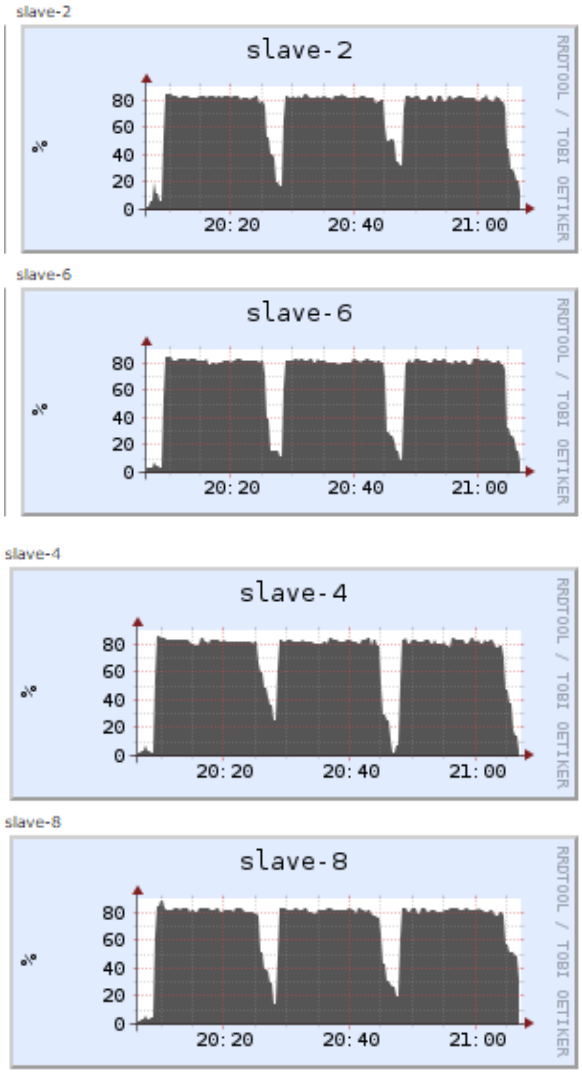
5.16. CPU usage node – 2 ex

Executors	*Time*
4	1 hours 1 minute

5.5. Executors table – 3



5.17. CPU usage cluster – 4 ex

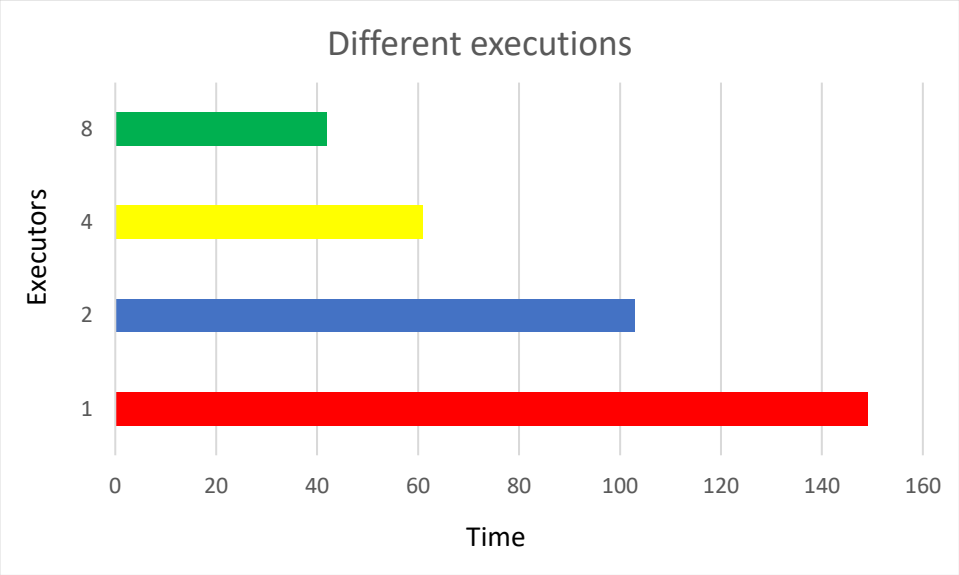


5.18. CPU usage node – 4 ex

5.7.1. Comments

After these executions we found that, depending on the dataset size, it's important to allocate the right amount of RAM so that the execution can be better parallelized. So, the right amount of RAM isn't the maximum possible, but the quantity that permits to spread the jobs on all the available machines.

We found that that the execution times decrease linearly as the slaves used are doubled (during the 1 to 4 slaves executions), while the 8-slaves execution decrease the time by half of the time gain of the other executions (from 40 to 20 minutes).



5.19. Executors graph

5.8. Comparison with sequential

Since we are dealing with two completely different implementations, it is difficult to make a comparison between sequential execution and parallel execution on SPARK. Observing the performances and the time obtained from the testing of the parallel execution, the advantages, in terms of execution time, regarding the sequential one appears obvious, as well as the facility with which the execution is carried out. The performances are not affected by the introduction of parallel computing and communication between the different nodes. In addition, due to the delay of the communication among the various nodes, the execution appears separated into three parts instead of being unique. This phenomenon obviously does not occur during sequential execution, as jobs are executed one after the other by the same node.

If we want to make a forced comparison, we could provide tables where, as our input varies (10, 20 and 40 GB) we can calculate the execution time of both programs to be able to make a forecast on a higher input (100/1000 Gb).

Input 10 Gb

Execution	***Time***
Sequential	2 hours 29 minutes
SPARK	12 minutes

5.6. Comparison table - 1

Input 20 Gb

Execution	***Time***
Sequential	4 hours 47 minutes
SPARK	28 minutes

5.7. Comparison table - 2

Input 40 Gb

Execution	***Time***
Sequential	9 hours
SPARK	42 minutes

5.8. Comparison table - 3

The executions with an input of little amount involve a partial use of the nodes that is reduced tendentially from 8 to 4.

6. Conclusion

The work has been very interesting. It gave us many ideas to learn and deepen skills in the use of parallel programming. In particular, the use of an instrument as new to us as SPARK, gave us the opportunity to get closer in some way, to the real problems that arise during the use of such platforms.

Like all applications, ours could be improved. Improvements could certainly be made to the refinement of the k-means phase and to foresee the use of an even larger data set.

This could probably significantly improve performance and consequently, get more accurate and meaningful features about clustering.

Source code

Entire source code developed for our applications is stored on GitHub.

- Code of sequential program is reachable at following link: https://github.com/co5m0/SOA-2020/tree/master/SequentialK-MeansPy
- Instead, at this link https://github.com/co5m0/SOA-2020/tree/master/SparkK-MeansScala is stored the SPARK implementation.

Bibliography

- Parallel K-Means Clustering Based on MapReduce. Weizhong Zhao, Huifang Ma, and Qing He. The Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences 2 Graduate University of Chinese Academy of Sciences.
- Automatic Keyphrase Extraction and Ontology Mining for Content-Based Tag Recommendation Nirmala Pudota, Antonina Dattolo,Andrea Baruzzo,Felice Ferrara,Carlo Tasso. Artificial Intelligence Laboratory, Department of Mathematics and Computer Science, University of Udine.
- Content-Based features extraction on Hadoop MapReduce Framework. Ciro Amati, Stefania Cardamone, Giovanni Grano, Advanced Operating Systems, University of Salerno.
- Comparing Apache Spark and Map Reduce with Performance Analysis using K-Means, Satish Gopalani, Rohan Arora. International Journal of Computer Applications, March 2015.
- Clustering e K-Means. https://spark.apache.org/docs/latest/ml-clustering.html
- Apache PDFBox. https://pdfbox.apache.org/
- Developing and Running a Spark WordCount Application. https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/spark_develop_run.html
- Term frequency-inverse document frequency. https://en.wikipedia.org/wiki/Tf%E2%80%93idf
- Downloading in bulk using wget. https://blog.archive.org/2012/04/26/downloading-in-bulk-using-wget/
- Archive.org About Search. https://archive.org/help/aboutsearch.htm
- Archive index. https://archive.org/details/arxiv-bulk?sort=-publicdate&and[]=collection%3A%22arxiv-bulk%22
- Advance search. https://archive.org/advancedsearch.php
- Pipelines in SparkML. https://spark.apache.org/docs/latest/ml-pipeline.html
- TFIDF su SparkML. https://spark.apache.org/docs/latest/ml-features.html#feature-extractors
- SparkNLP. https://nlp.johnsnowlabs.com/docs/en/concepts
- Scikit-learn per ML on python https://scikit-learn.org/stable/
- nltk for NLP on python https://www.nltk.org/