Data Clustering on the Parallel Hadoop MapReduce Model

Dimitrios Verraros

dimver@it.teithe.gr

Department of Informatics,

Alexander Technological Educational Institute of Thessaloniki

February 9, 2014

Abstract

Machine Learning is one of the best ways to process and analyse data. The industry created the tools to utilize the benefits of machine learning algorithms by executing them in a parallel way, on huge computer clusters where the information is stored. One of the most popular is Apache Hadoop, which provides the abstractions needed to perform those operations in a way that more businesses, organizations and individuals can use it, in order to achieve their goals. In this thesis, we examine the most popular machine learning algorithm, the K-means, and implement it on the MapReduce framework. We then execute it on a Hadoop cluster, to measure the performance gains offered by parallelizing the algorithm that analyzes data distributed on multiple machines. Alternative solutions and evolutions in the direction of parallel data processing are presented to conclude an overview of the possible directions that the Big Data term moves towards.

Contents

Chapter 1

Introduction

The web is evolving increasingly fast, leading to a chaotic amount of information that needs to be stored, accessed and processed efficiently. Thus, the importance of finding better ways to manipulate and analyse this volume of data is greater than ever.

Machine Learning is the branch of artificial intelligence, that managed to reach greater adoption in the industry, as the most effective way to train systems with data. While being around for more than 50 years, machine learning gained a lot of popularity in the past decade, with large scale companies using those algorithms for mainstream products, like web search, recommendation systems, photo tagging or spam filtering among others.

However, those techniques require a lot of computational power and time, and speed is a significant factor for companies, organizations and scientific research nowadays. Given the distributed nature of modern storage solutions that exceed the petabyte scale, is suitable to parallelize these algorithms and have them operating close to where the data resides, in order to achieve greater results that match the demands of the industry.

Apache Hadoop is a tool that captures those needs and simplifies the operation and management of large computer clusters. It offers a distributed storage layer for the data, a resource manager, and an implementation of the parallel computational framework MapReduce.

In the first section of this thesis, will focus on the clustering of data using machine learning algorithms and particularly the K-means. We will then introduce the Hadoop project, analysing its main components and offerings, while presenting a setup and utilization guide on a cluster. The MapReduce programming model, will be examined, in order to understand how it is possible to write and execute parallel programs on top of a Hadoop cluster.

We will then implement a parallel K-means algorithm using the MapReduce API, to present a scalable solution that makes possible to process vast amounts of data in a fair amount of time. We will see which parts of the algorithm can be parallelized in order to apply a clustering method on large scale data. Comprehensive experiments are conducted in order to demonstrate the possible attributes that can affect the performance of the algorithm and then scale up and out, showcasing the advantages and disadvantages of each scenario.

Finally the shortcomings of using the MapReduce framework for iterative algorithms, like K-means, is discussed, and alternative solutions that deal with this problems are briefly presented.

Chapter 2

Data Clustering and K-means algorithm

2.1 Introduction

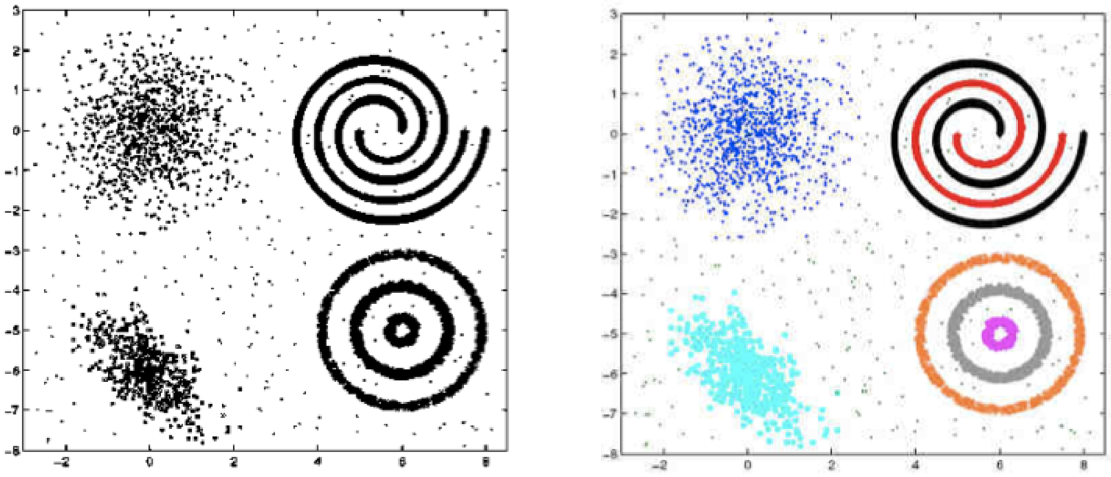
“Organizing data into sensible groupings is one of the most fundamental modes of understanding and learning”[13].

Learning is more than important nowadays, in the age of internet, surveillance, and social media. Recognising patterns amongst those sources provides significant value when it comes to understanding consumer habits, stock exchange, traffic flow, news and many other events which are now consistently stored and being available for accessing and analyzing them. Machine Learning is the science that investigates and tries to find out new ways of making useful assumptions and predictions out of the available knowledge. It is more abstractly split into two categories of supervised and unsupervised learning with their main difference being down to the ability of the respective algorithms to operate having labelled or unlabelled data in their disposal. Given the nature and the amount of data coming out of the aforementioned sources it gets clear that supervised learning is not applicable due to the fact there is no prior knowledge on the information we want to dig into. Cluster analysis is the study of algorithms and techniques for finding structure and similarities between objects that share the same characteristics. This is what distinguishes clustering, unsupervised, from classification analysis, unsupervised learning. The subject of Clustering is a long running topic in the Machine Learning science. Since the proposal of the K-means algorithm, more than 50 years ago, a vast amount clustering algorithms have been proposed, still though K-means remains one of the most popular ones, mostly due to its simplicity and effectiveness.

2.2 Data Clustering

Data clustering, or cluster analysis, is a tool for discovering the structure of data, without making assumptions used in other statistical methods. Objects are grouped together by their relationships instead of identifying them by preset tags or labels. Representing the entities of a dataset as points in a multi-dimensional space is the key to group them with their similar ones and distinguish them from the ones that differ. The biggest problem at this point is defining similarity. This can greatly vary because of differences in shape, size and density. Figure 2.2.1 shows how diversity in clusters can make the task of isolating the entities harder.

Figure 2.2.1: Cluster Diversity [13]



While it is pretty clear for a human to recognise the patterns in the previous example it is nearly impossible for the available algorithms to separate them. On the other hand, it is equally difficult for humans to perceive and understand a dataset which is presented in more than 3 dimensions. Data representation is thus essential for successfully identifying clusters and equally important with defining proximity and measuring it. Clustering algorithms are consistent, fast and reliable, enough to pose as great candidates for solving this king of problems. In the next sections we will describe the significance Data Representation and how perform clustering analysis on them using one if the most common and popular clustering algorithms, the K-means.

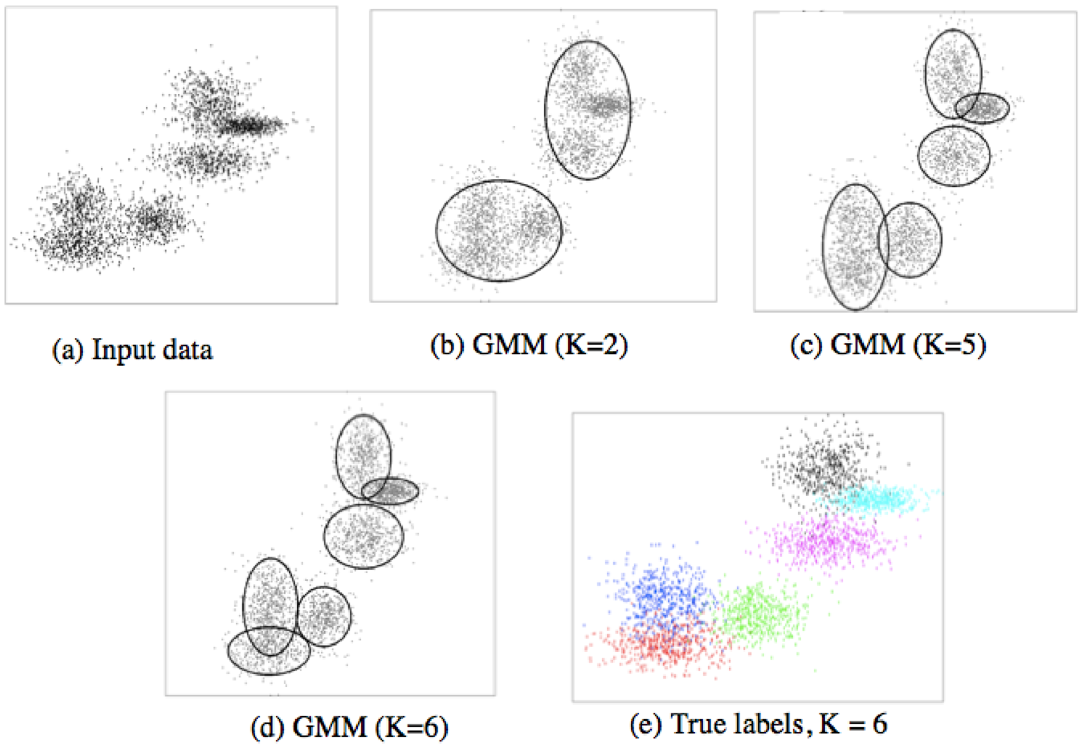
2.2.1 Data Representation

Representation of the data influences significantly the outcome of a clustering algorithm. With good representation clusters are going to be compact and isolated but this not always easy to achieve since prior domain knowledge is required. Careful selection of the purpose of grouping, the number of clusters but also using the appropriate data type and scale, are key points to have a successful clustering.

Grouping Deciding on the proper grouping decides the final outcome and conclusions to be made out of the clustering. In a movie dataset for example the movies could be grouped based on their country of origin, their genre or their budget. Those characteristics would give completely different clusters on the same dataset of movies.

Number of Clusters The number of clusters is not trivial to determine. Most of the times the number of clusters has to be manually defined but many automatic ways of deciding them are also proposed. Figure 2.2.2 shows how the number of clusters affects the result.

Figure 2.2.2: Number of Clusters [13]



Data Types Data Types refer to the “degree of quantization in the data”[14]. Each feature can be either binary, discrete or continuous. *Binary* data can be represented on a data matrix as 0/1s (or yes/no). *Discrete* can be a feature which is assigned a finite amount of values, like stars on an album rating. Likewise *continuous* can take as a value any real number in a given range. Data Types are an important factor for selecting the appropriate proximity function.

Data Scales Every object in a dataset is defined by a d-space vector. The values that make up each vector should be in a range that makes sense when compared to the rest of the objects so that the *ratio* and the *interval* between them is suitable for successfully separating the objects into clusters.

2.3 K-means

The selection of the most appropriate clustering algorithm depends on the factors mentioned above. The K-means algorithm proves to be quite effective across different domains and alongside its speed and efficiency. K-means is also quite easy to parallelize, making it scalable and eventually capable of analyzing very large data sets.

K-means is a partitioning algorithm that separates the objects of a dataset, represented in a *n*-dimensional data matrix. Given a set of n-dimensional points *X* = {x_i}, i=1,. . . ,n to be clustered in a set of *K* clusters, *C* = {c_k}, K=1,. . . ,K, K-means assigns each point to cluster in a way that the squared mean error of every cluster is minimized. The squared error of a cluster is the sum of the squared Euclidean distances between every point in a cluster C_k and its center $\mu^{(k)}$, which is also called within-cluster variation.

$$e_k^2 = \sum_{i=1} \left(x_i^{(k)} - \mu^{(k)}\right)^2 \tag{2.3.1}$$

Besides the Euclidean distance, other metrics can also be used, with the most popular being the Manhattan distance

$$Mht = \sum_{i=1} |x_i^{(k)} - \mu^{(k)}| \tag{2.3.2}$$

or the Mahalanobis distance. While the fisrt two are able to find only spherical clusters, using the Mahalanobis distance, ellipsoidal clusters can be detected, due to the fact that the Mahalanobis function takes into account the Cosine similarity, which however increases the computational cost. The goal of K-means is to minimize the square-error of all entire cluster, which is calculated by the next function:

$$E_K^2 = \sum_{k=1}^{K} e_k^2 \tag{2.3.3}$$

K-means is an iterative algorithm. It is implemented in two parts which are then repeated all over until a stopping condition is reached. This can be the number of iterations, the convergence or the combination of those two. The main part of the algorithm is implemented like that:

- Define the number of clusters K, and initialize them
- Repeat the following steps until the stopping condition is met
  - Assign every object to its closer cluster
  - Recalculate cluster centers

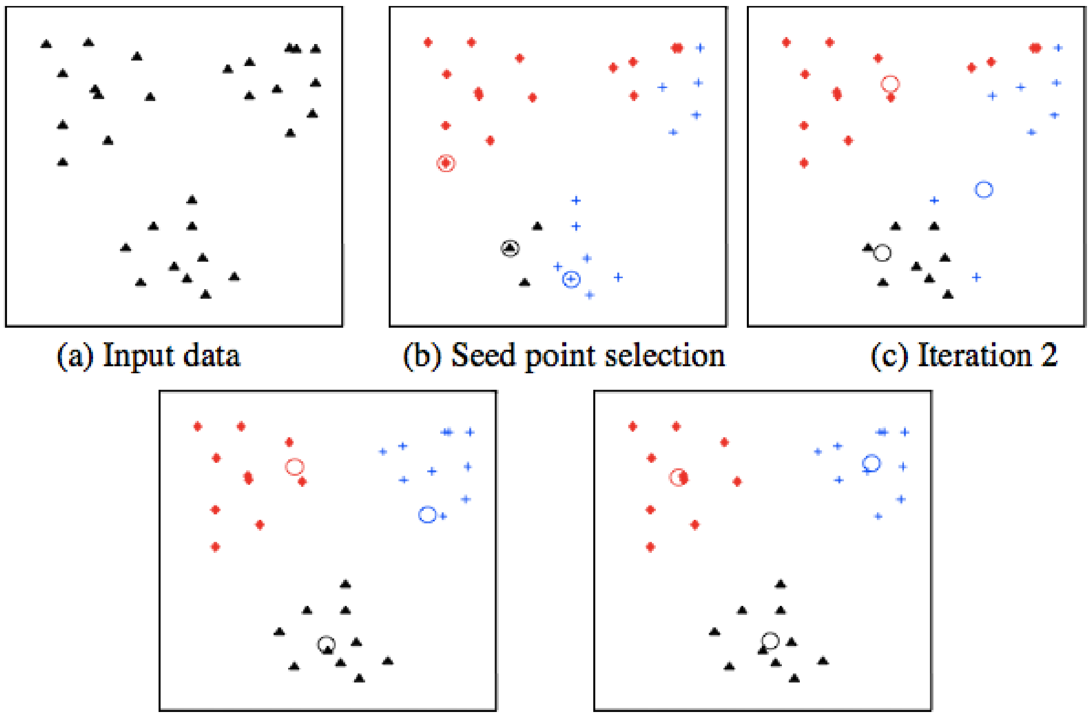
Figure 2.3.1 shows the steps during the execution of the K-means algorithm on a 2-dimensional dataset.

The are various implementations of the K-means algorithm, and picking one depends on the characteristics of the dataset. The most commonly used are the Forgy/Lloyd algorithm, the MacQueen algorithm and the Hartigan & Wong algorithm.

As mentioned earlier the selection of the number of clusters is the most critical choice. It is common that the algorithm can be run many times with different amount of clusters until the most meaningful result is picked. The way the data are represented also affects the clustering results. K-means converges to local minimums and running it many times with different combinations of the above can help finding out the best solution which is characterized by the smallest value of square error. It is always important to work on a small sample of the dataset for making the initial assumptions, and then move on to analyzing the whole dataset.

K-means has a complexity of O (x*n*K*I), where x is the number of points, n the attributes of each point, K the number of clusters and I the number of iterations. Due to the fact that it monotonically approaches a local minimum of the cost function, it is described as a *hard* Clustering algorithm.

Figure 2.3.1: (a) In the first step the data are represented on a 2-dimensional table (b) Three objects are selected as the initial cluster centers (c), (d) and (e) The algorithm iterations with the center recalculations until convergence is achieved [13]



(a) Input data (b) Seed point selection (c) Iteration 2

(d) Iteration 3 (e) Final clustering

Extensions of K-Means and other approaches on Data Clustering Many extensions have been proposed, in order to solve various problems related to it with the most notable ones being the fuzzy K-means [Dunn, 1973], X-Means [Pelleg & Moore, 2000], Kernel K-Means [Scholkopf et al., 1998]. All of those, improve the results of the K-means in many aspects, but hamper its performance by increasing the complexity of the algorithm. Bregman Divergences [Banerjee, Arindam; Merugu, Srujana; Dhillon, Inderjit S.; Ghosh, Joydeep (2005)], and Expectation Maximization (EM) [Dempster, Laird & Rubin (1977)] are also trying a similar approach to K-means by using probabilistic methods for measuring the similarity of the patterns. They are both characterized as *soft* Clustering algorithms since they try to avoid local optima by assigning each point to multiple cluster with a membership value (*soft assignment*).

Nevertheless, K-means remains one of the most prominent clustering algorithms. It is also quite easy to parallelize making it an excellent candidate for distributed computation on big clusters.

In the next chapter we will dig into Apache Hadoop, the one of the most popular distributed computational and storage solutions. We will the describe how to implement the K-means algorithm on the MapReduce framework and parallelize it in order to be able to analyze huge amounts of data stored across thousands of machines.

Chapter 3

Hadoop

3.1 Introduction

As of 2013 Hadoop tends to be a first class synonym when it comes to Big Data. Still it is a bit unclear what Hadoop is actually though.

Hadoop started as part of the Apache Nutch project, an open source web search engine, itself being part of Apache Lucene. One of the first issues its authors came upon was the problem of scalability and the ability of managing hundreds to thousands of machines that were meant to handle the analysis of the ever growing amount of data. Things became more clear with the publication of Google's MapReduce and Google File System (GFS) papers[4].

Hadoop is an open-source implementation of the aforementioned concepts of MapReduce and GFS. Being already applicable for a much broader realm than the one of Nutch, Hadoop became an independent subproject of Lucene in 2006. Early adopted and supported by Yahoo! Hadoop turned into a top-level project of the Apache Software Foundation in February of 2008. At the time of writing, Hadoop is used by a vast majority of technology giants including Facebook, Google, Microsoft, Twitter, Last.fm, Netflix and Spotify to name a few.

3.1.1 Big Data

The exponentially increasing amount of data raised the need for an efficient solution to collect, store and analyze them. The term Big Data is quite spread lately with its definition being quite loose. In general Big Data can be described as a *very large, loosely structured data set that defies traditional storage*[18].

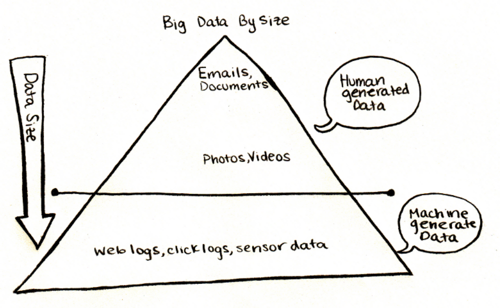
Those data come out from two sources: Human Generated Data and Machine Generated Data.

Human Generated Data consist from emails, articles, facebook statuses, photos, tweets etc. Considering the amount of time that each one of us spends on a computer every day you can imagine the rate of data generation in a world scale.

Machine Generated Data comes out of activity logs, traffic cameras, stock exchange, flight's telemetries and more.

Prior to Hadoop, machine generated data were largely ignored, mostly down to the inability and inefficiency to collect those. However things changed and we know now have machine generated data outnumbering the human generated data.

Figure 3.1.1: Big Data Pyramid [18]



And some numbers:

- Facebook: has 40 PB of data and captures 100 TB / day
- Yahoo!: 60 PB of data
- Twitter: 8 TB / day
- eBay: 40 PB of data, captures 50 TB / day

The above information can help us understand the scale of the amount of storage and computational power which is required in order to deal with the above requirements. Moreover considering the nature of that data which tends to be unstructured or semi-structured traditional methods of storing and accessing them, like relational databases, prove to be not such a good fit.

3.2 The Hadoop Approach

Having in mind those demands, Hadoop can solve those problems by providing a common compute and storage cluster. Merging those two eliminates the problem of having separate structures for processes that are about to exploit both of them. It skips the communication barriers raised and makes managing those tasks easier offering the right tools. Most importantly Hadoop clusters can run and scale horizontally on commodity hardware that can be added and removed with ease. While it can run on a single node it can scale out to thousands of nodes with different specification and efficiently distribute jobs across them. As stated by Yahoo! *hundreds of gigabytes of data constitute the low end of Hadoop-scale*[12], thus earning the tag "web-scale".

Hadoop is a big ecosystem under which a lot of projects lie, including HBase, ZooKeeper, Hive and Pig amongst others, all of them running on top of Hadoop. Itself though consists of the following core components.

The *Hadoop Common* package is a collection of Operating System tools and utilities, that help managing the Hadoop instance, the needed Java Archives (JAR) files and scripts in order to run it, the source code, the documentation and the libraries needed to use it.

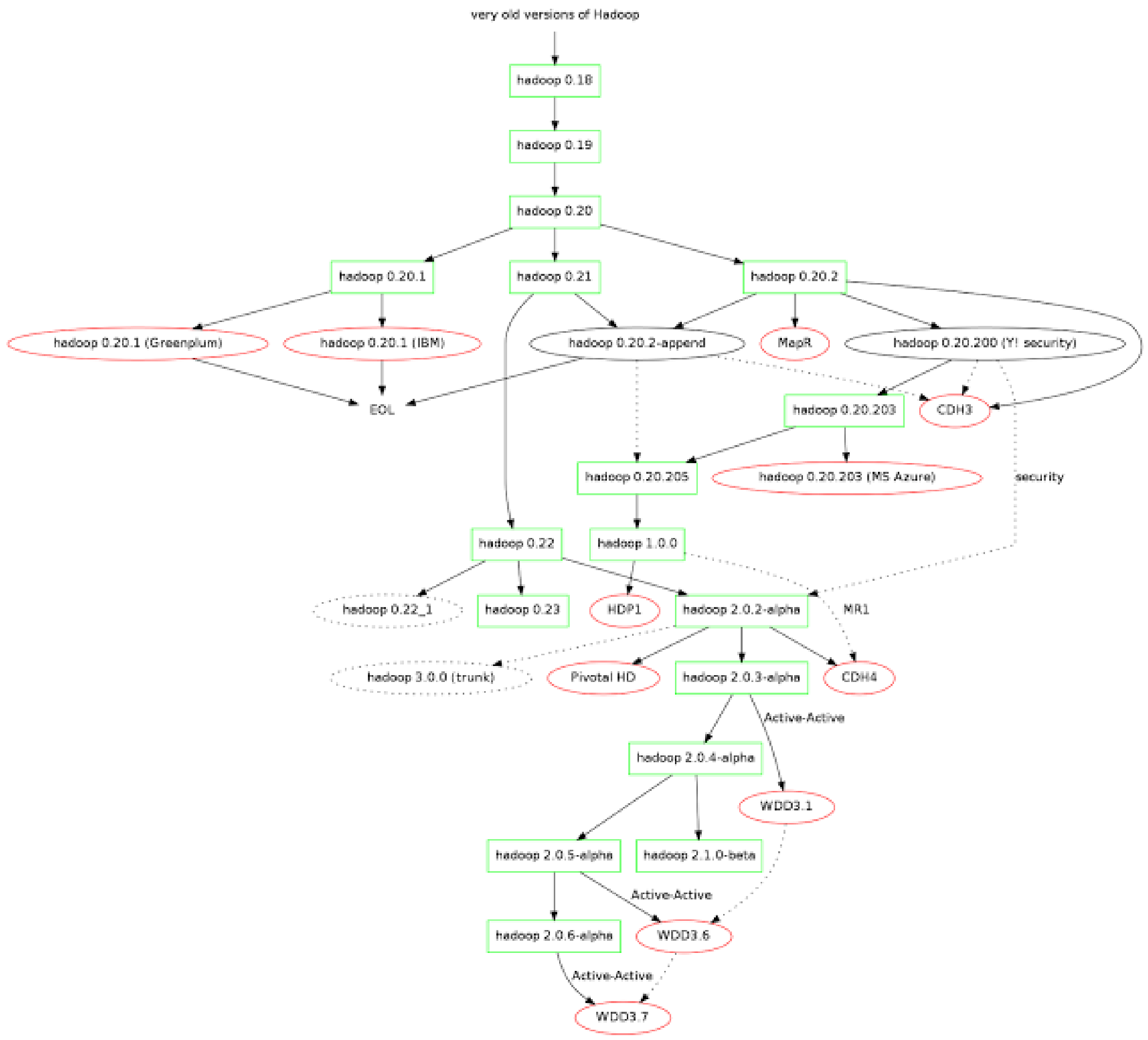
Next comes MapReduce and HDFS. It is suitable here to distinguish MapReduce and Hadoop though. MapReduce is a programming model *for processing large data sets with a parallel, distributed algorithm on a cluster*[17]. It is inspired by the *map* and *reduce* functions commonly used in functional programming. The model was initially developed by Google and it was published in 2004. Its target is to distribute tasks in multiple computers and make sure that they are successfully executed regardless network or machine failures. *Hadoop MapReduce* is the open-source implementation of that computational model which goes hand by hand with the next module which is the Hadoop Distributed File System (HDFS).

HDFS is also modeled after Google's GFS. It stands for the "storage layer" of a Hadoop cluster. Considering the amount of data meant to be stored on a Hadoop cluster, going for expensive hardware and complex structures to achieve the needed scale and speed proves quite inefficient. Another fact is that this data is going to be stored on many computers, since the filesystem is distributed, which are commodity hardware increasing the possibility of failures. Thus Hadoop has to take care of the possible failures by having replication and keeping copies of that data on many machines.

3.2.1 Versions and Development Timeline

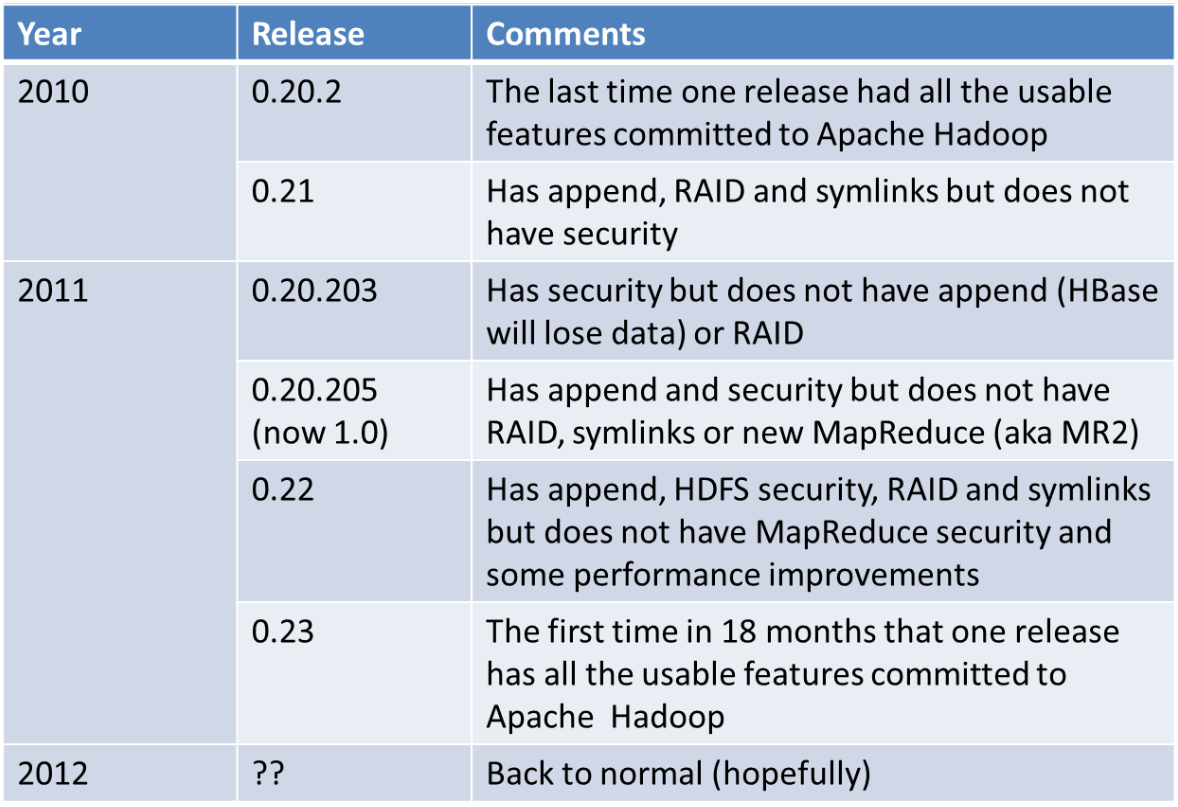
A very common image that someone can come across while looking for the right Hadoop version is seen in Figure 3.2.1.

Figure 3.2.1: Hadoop Versions [2]



Since Hadoop came in life a lot of companies showed a great interest in it and actively supported its development. In most Apache projects new features are developed on a main codeline named “trunk”. Every large feature is developed on its own branch which is supposed to be merged in the future back into trunk. Any community member can create a branch and name it as they like. Big players like Cloudera (CDH), HortonWorks, MapR and IBM got involved among others. In most of the cases it ended up each one of them working on their own branch, bringing their features which finally reached a situation that there has been an 18 month period where there has been no one Apache release including all the committed features[24]. Table 3.2.2 shows the features each version has, as of the beginning of 2012.

Figure 3.2.2: Hadoop Version Diagram [24]



Year	Release	Comments
2010	0.20.2	The last time one release had all the usable features committed to Apache Hadoop
	0.21	Has append, RAID and symlinks but does not have security
2011	0.20.203	Has security but does not have append (HBase will lose data) or RAID
	0.20.205 (now 1.0)	Has append and security but does not have RAID, symlinks or new MapReduce (aka MR2)
	0.22	Has append, HDFS security, RAID and symlinks but does not have MapReduce security and some performance improvements
	0.23	The first time in 18 months that one release has all the usable features committed to Apache Hadoop
2012	??	Back to normal (hopefully)

Since then the situation didn't evolve in a much better direction, however it is a bit clearer where is the current stable version and which one is the main future branch. An important thing to have in mind is that the version number does not indicate the chronological order in which each one of it started. For example Hadoop 0.22 was released one month after 0.23.

Hadoop 1.x.x, a descendant of branch 0.20.x, is now the main stable version, containing most of the features and being officially supported by most Hadoop vendors. The current version of this branch is 1.2.1. The second main, and stable version, branch is 2.x.x, which comes out of branch 0.23.x. It is still a big question which version of Hadoop is someone supposed to use considering its features, stability and support. He should also make sure that the version he picks is compatible with the subproject of the Hadoop family he is indenting to use with. The following table sums up the features supported by each version:

Some conclusions can be made having a look on this table, which confirm the aforementioned version history. Versions 0.23 and 2.0 are almost identical. Versions 0.20 and 1.0 alongside the Cloudera versions are production ready. It is also important to notice which versions support YARN (MRv2) and which ones MRv1,

Feature	0.20	0.21	0.22	0.23	1.0	2.0	CDH3	CDH4
Production quality	X				X		X	X
HDFS append		X	X	X	X	X	X	X
Kerberos security		X[a]	X	X	X	X	X	X
HDFS symlinks		X	X	X		X		X
YARN (MRv2)				X		X		X
MRv1 daemons[b]	X	X	X		X		X	X
Namenode federation				X		X		X
Namenode HA				X		X		X

Table 3.2.1: [a] Support for Kerberos-enabled HDFS only. [b] All versions include support for the MRv1 APIs.

with them being mutually exclusive.

Versions 0.23.x and subsequently 2.x see the introduction of YARN. This is one of the major overhauls brought in those versions. In the chapter dedicated to the MapReduce part of Hadoop we will explain the differences between MRv2 and MRv1 and the significance of the new features in terms of performance and efficiency that the new implementation carries.

Considering the improvements this branch brings and that is now the second stable version of Hadoop, this project will go through the details and architecture of Hadoop 2.2.0.

3.2.2 Architecture Overview

As mentioned earlier, Hadoop consists from Hadoop Common, the Hadoop Distributed File System (HDFS) and MapReduce/YARN. In the following sections an overview of the design and features, alongside with a setup and configuration of those components, will be presented.

3.2.2.1 Hadoop Distributed File System (HDFS)

Like every Hadoop component HDFS is written in Java. Given the fact that Java is highly portable and supported by most machines, HDFS can be deployed on a wide range of them. HDFS is a distributed file system with its key differences when compared to other distributed file systems is its ability to run on low cost, commodity hardware. This fact triggers a series of assumptions and solutions that HDFS is supposed to provide.

Hardware Failure Given the status of the used hardware, HDFS takes failure as a common situation rather than a worst case scenario. Therefore fault-tolerance is very important and detection of failures and fast recovery from them is the main goal. This is achieved by replicating (duplicating) data in many different nodes.

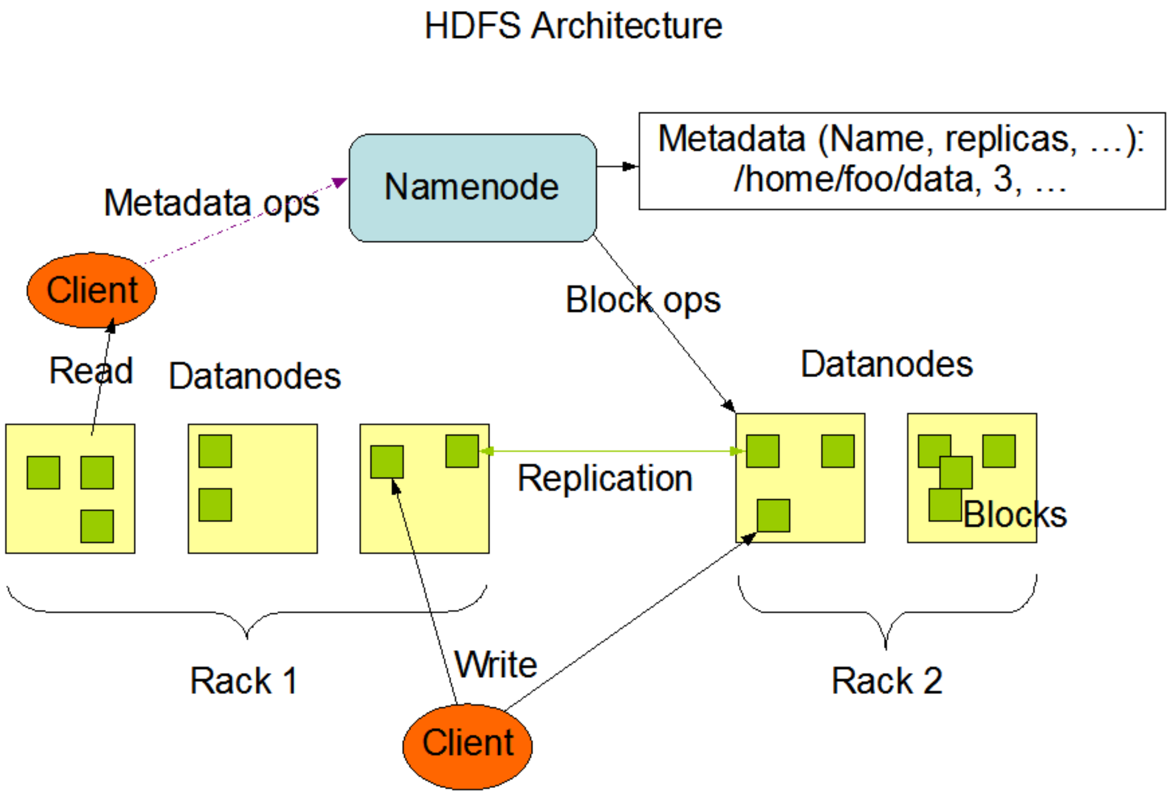
Streaming Data Access and Large Files HDFS is quite different from a common file system where a user is accessing frequently many small files, but actually fast throughput is needed for use with large data sets. HDFS favours that instead of low latency. Likewise HDFS trades some key features of the POSIX specification in order to adapt to the demands of applications targeted to for HDFS.

Data Sets used on Hadoop clusters tend to take gigabytes to terabytes of space. HDFS is tuned to match those requirements having a typical block size of 64 to 256 MBs. Another key point of HDFS is that it does not support overwriting files but only creating and appending to them favouring write-once-read-many approach enabling high throughput data access fitting perfectly a Map/Reduce application or a web-crawler.

Master/Slave Architecture Since data is spread across many nodes there must be a way to coordinate them. Every machine has a 'daemon' running which is talking to Master node that manages the whole cluster, hence ending up with a master/slave architecture. This brings us to the conclusion of having a single point of failure in our cluster, it being the master node. While many could consider that unreasonable this approach greatly simplifies the system's architecture, design and implementation. This single node, named *NameNode*, manages the file system namespace and regulates file access amongst clients. In other words it controls all the file system operations like file creation, access, permissions, etc. The *DataNodes* run on every node and represent the slaves of the cluster and are responsible for managing the storage of every machine they run on.

Files are stored in many blockes which are distributed and stored in a set of DataNodes. The NameNode performs the file system namespace operations and

Figure 3.2.3: HDFS Architecture [6]



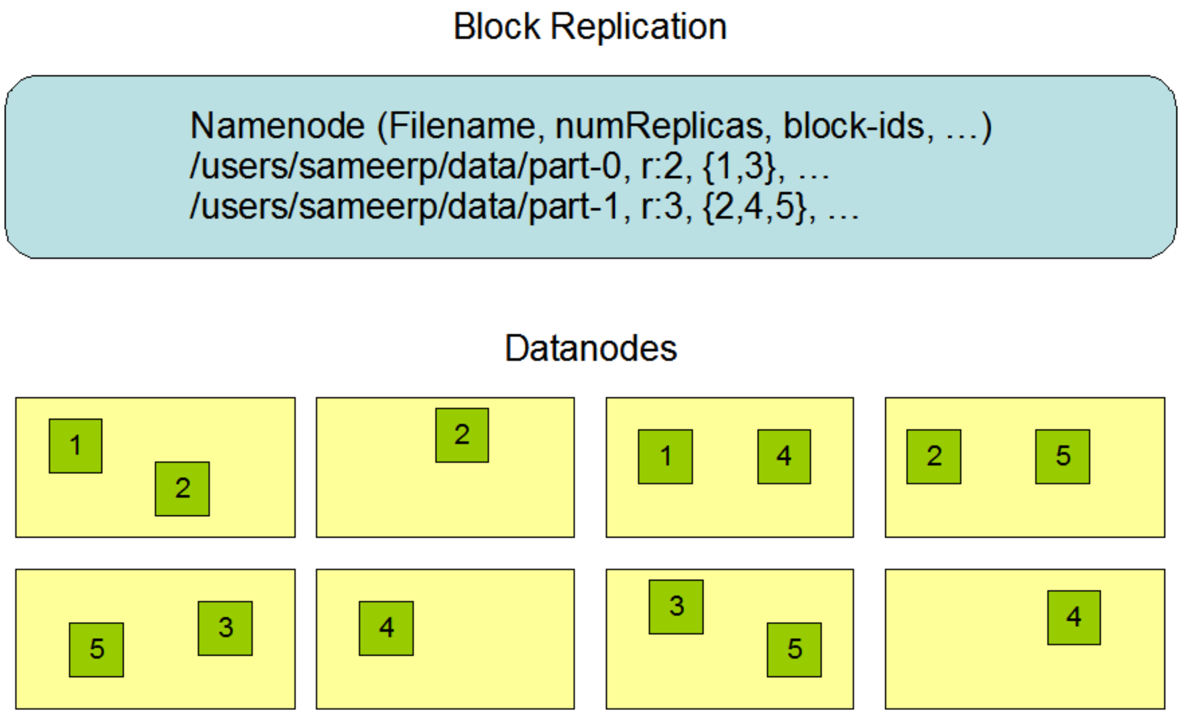
determines the mapping of blocks across the DataNodes, which on their turn execute the commands received from the NameNode.

Data Replication Like in most file systems, files stored on HDFS are split into sequences of blocks, each one of them, except the last, having the same file size. To achieve fault-tolerance those blocks are replicated in many nodes, which can be configured per file. The NameNode determines where each block shall be stored, and is aware of the status of every DataNode by receiving a Heartbeat and a Blockreport from them periodically. The replication of data across the DataNodes can be seen in Figure 3.2.4.

The way those blocks are distributed depends on the cluster configuration. In most cases they are distributed evenly across the DataNodes with every block running on different nodes or even different racks. Tuning the cluster for increased reliability and performance is not a trivial task and requires a lot of experience. There are many policies that can be implemented having in mind the network topology and bandwidth, the available hardware and the nature of the data being stored.

Despite having only one master node on which the NameNode runs there can be a secondary NameNode. Because of the fact that the NameNode can also fail,

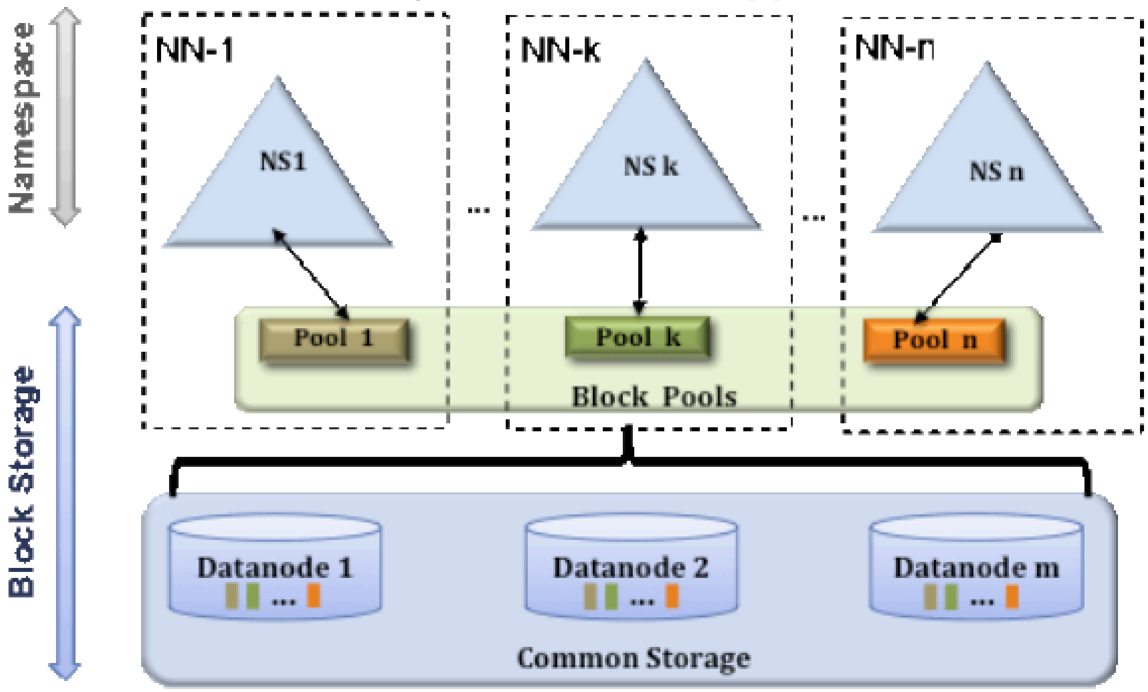
Figure 3.2.4: Datanodes [6]


Block Replication
Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

the secondary NameNode acts as a fail-over node merging the namespace image and logs and keeping a backup of those. It normally runs on a separate physical machine from the master node and can replace the master node in case this one fails. It should be noted that the secondary NameNode is not an exact copy of the NameNode, and there must be a recovery process from the NameNode in on order to replace it. The 0.23.x and 2.x sees the introduction of two new significant features in HDFS, the HDFS Federation and HDFS High-Availability which handle those scenarios with ease and effect.

HDFS Federation When the cluster grows up the need to scale the name service horizontally comes up. With the introduction of HDFS Federation it is possible to run many NameNodes which run independent and do not coordinate with each other. This way the namespace can be split across many nodes having each one managing a diffent rooted directory. All DataNodes are used as storage blocks by every NameNode simultaneously. Under federation each NameNode manages a self-contained unit, made up of the namespace and a block pool named *Namespace Volume*, with every block from the those block pools being stored on the DataNodes. Figure 3.2.5 illustrates the concept of block pools.

Figure 3.2.5: Block Pools [6]



HDFS High-Availability Having multiple NameNodes and secondary NameNodes the possibility of data loss is very limited, however every NameNode is a single point of failure. With HDFS HA NameNodes can be configured in active-standby pairs that share all the log entries and block mapping, making it possible for the standby node to quickly take over in case a NameNode fails.

HDFS Interfaces HDFS can be accessed in many ways. Those include a command line interface, Java and C APIs, and through HTTP using the WebHDFS and HttpFS clients.

The command line interface can be accessed from a node using the `bin/hadoop dfs` command from Hadoop's installation root directory, passing as parameters common unix commands like `ls` or `cat`, among others.

HDFS can be used from an application using the provided libraries for different platforms. Java APIs are available with the Hadoop distributables while C programs can use libhdfs. There are also packages for python like Pydoop or Snakebite.

Finally HDFS can be accessed remotely through HTTP. WebHDFS exposes all the available HDFS operations with a REST API, while HttpFS is a REST HTTP gateway which is interoperable with webhdfs. An important difference is that HttpFS supports communication with clusters running on different versions of Hadoop overcoming RCP versioning issues that occur in webhdfs.

3.2.3 Hadoop MapReduce

Hadoop 2.x sees the introduction of YARN, a revision of the initial MapReduce implementation, in order to catch up with the increasing demands of the industry and overcome the limitations of the first version. At first we will go through the objectives and targets of the MapReduce concept and then describe the architecture of YARN.

As mentioned earlier MapReduce is a framework for writing applications which run on a large number or machines, that make up a cluster, and process huge amounts of data in a parallel way. It is designed to run where the data resides, taking advantage of low latency. MapReduce takes place in two steps, *map* and *reduce*, hence the name. Each step is split in multiple jobs which operate on key/value pairs, that are coming as an input, get processed and returned as a new output in key/value form.

On the *Map Step* the master node takes the input divides it into smaller jobs which are independent and run on parallel on the worker nodes. Every job processes chunks of data which reside on the node it runs on and after they are finished they return the answer to the master node.

On the *Reduce Step* all the outputs collected from the executed jobs are combined and the reduce task returns in turn the output of the problem.

The number of mappers and reducers is a very important factor for the performance of Hadoop. Many tasks increase the framework overhead but improve the load balancing and lower the cost of failures. The normal amount of maps is around 10–100 per node. This number is determined by Hadoop, but depends on the configuration. Task setup can take time so it makes sense for every mapper to have an significant amount of work to do. A decisive factor for the number of mappers is the size of the input and the block size, since the amount of splits equals the division of these values and one mapper is assigned per split. The proposed rule for the number of reducers ranges from 0.95 to 1.75 multiplied by the number of nodes and the maximum number or simultaneous reduces per task tracker.

Compute nodes tends to be the same as the storage node and with efficient configuration the tasks are scheduled to the appropriate nodes thus increasing the bandwidth across the cluster.

To have a better understanding of the map/reduce process we will got through the commonly used example of finding the maximum temperature from a dataset containing recorded temperatures from many cities many times per year. We assume that the data are semi-structured which means data may be missing for some cities over the years or they are formed in slightly different way.

Our dataflow starts by reading the file, parsing it and formatting the data to `<key, value>` pairs that can be represented like this:

```
<Athens, 25>
<London, 17>
<Berlin, 15>
<Paris, 24>
<London, 23>
<London, 19>
<Paris, 28>
```

This is the *input reader* which divides the data and assigns it to each *Map function* that will process it and give a result like this:

```
<Athens, [25]>
<Berlin, [15]>
<London, [17, 19, 23]>
<Paris, [24, 28]>
```

The amount of maps is normally decided by the number of the inputs which in turn is the total number of blocks of the input files. While the map functions are completed, their output is collected and merged. During this process, the *partition function*, the results are sorted and shuffled simultaneously and they are allocated to individual *reducers*. The partition function generates a hash key for every task and by its module the respective reducer is selected. On the *reducer function* the aggregated values are reduced in smaller sets which are decided by the key. In our case the output of the reduce function will look like this:

```
<Athens, 25>
<Berlin, 15>
<London, 23>
<Paris, 28>
```

Note that the results are sorted by their key which is the city name. Final step is formatting the results and writing them to a file on HDFS which is handled by the *Output Writer*.

As part of the process there is a *Reporter* which tracks the job execution and reports the progress to the master node, sets the application-level status and holds the *Counters*.

3.2.3.1 MapReduce 1.0 and YARN (MRv2)

With the second version of Hadoop, MapReduce was was revised in order to overcome increased maintenance costs and addressing the significant drawbacks like lack of support for non-MapReduce job execution. YARN is actually a separation between the MapReduce model implementation and the task execution system.

The initial implementation of Hadoop MapReduce the project was split in the those parts:

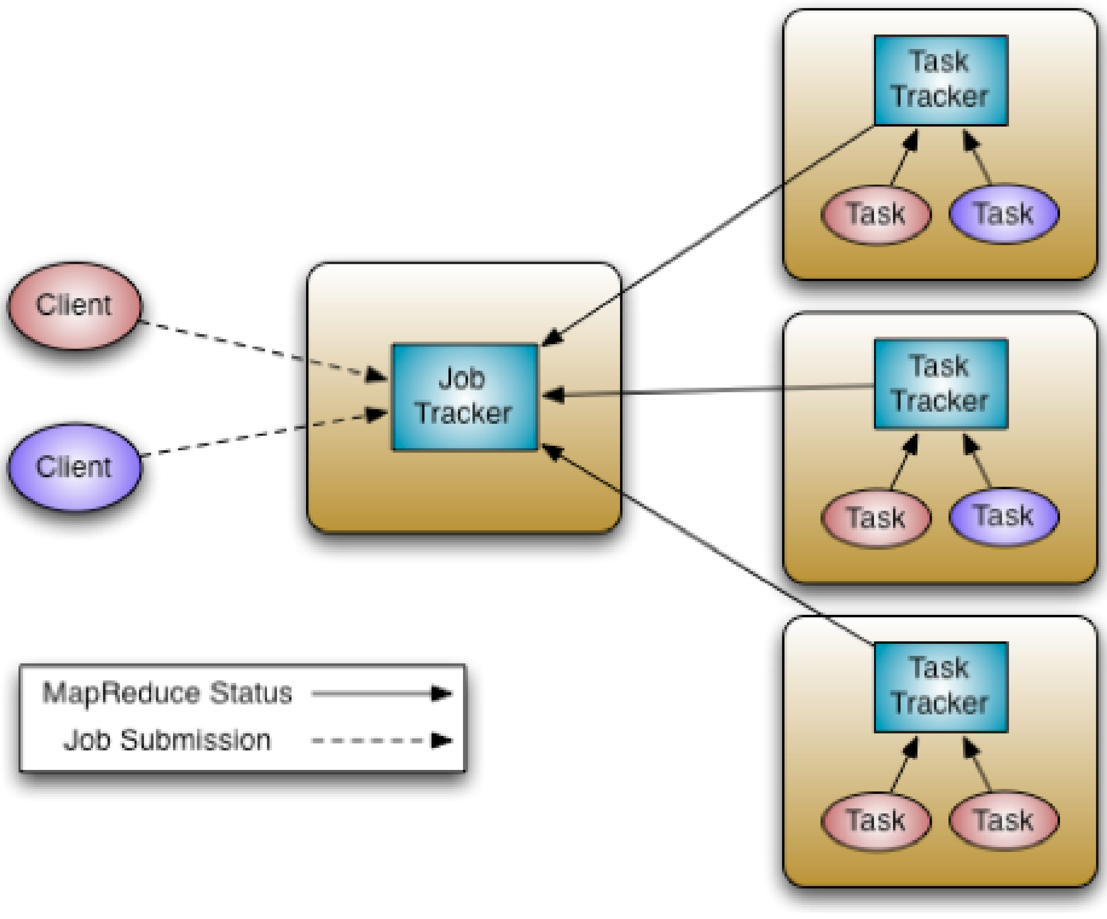
- The MapReduce API for writting MapReduce programms
- The MapReduce framework, which is the implementation of the mapreduce steps
- The MapReduce system, which are the services running on the nodes taking care of the job monitoring and execution

The MapReduce system is composed from the JobTracker and the TaskTracker. The JobTracker runs on the master node, while the TaskTracker runs on the slaves. JobTracker is assigning tasks for the submitted jobs to the TaskTrackers that are responsible for executing them. Figure 3.2.6 gives an overview of the first Hadoop MapReduce implementation.

Moving to YARN Support for more programming models is one of the main reasons to separate MapReduce from the task management system, and this is down to the fact that not every application can be modelled efficiently on it. The difference between Hadoop version 1.x and 2.x on the rerource management level, can be seen in Figure 3.2.7. Since a massive amount of data is already residing on HDFS the need to be able to run more ways of processing them on the cluster beyond MapReduce, it would be appropriate to be able utilize them through many paths.

Demands for bigger clusters were also there and the existing implementation had limitations on its scalability. The biggest step to achieve that came by splitting

Figure 3.2.6: MapReduce 1.0 [6]



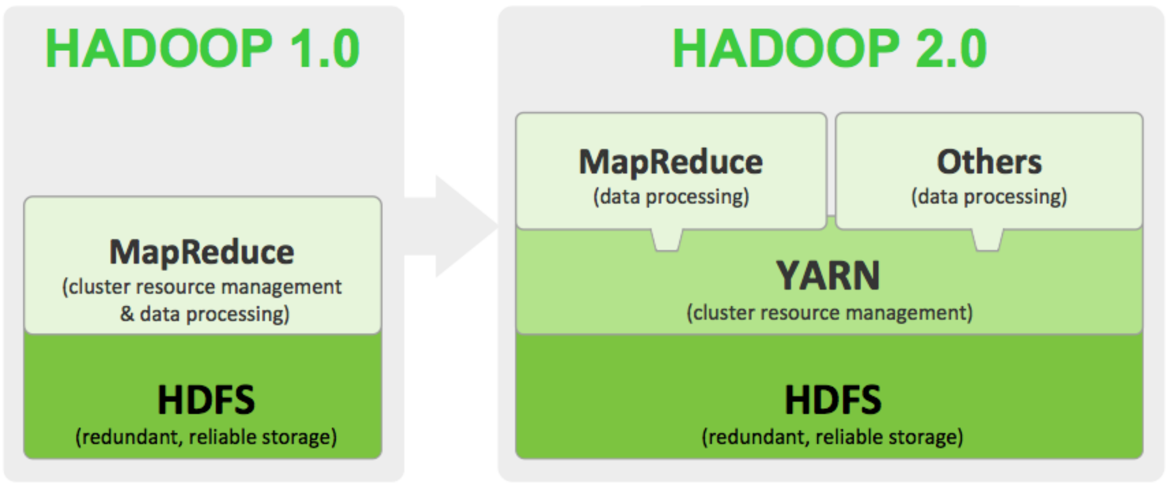
JobTracker's main responsibilities of resource management and job scheduling in two daemons, the *ResourceManager (RM)* and *ApplicationMaster (AM)*. The ResourceManager is a global entity while the ApplicationMaster runs per-application into the cluster tracking the execution of every job. A *NodeManager (NM)* runs on every slave node arranging the resources and the processes which run on it and reporting back to the ResourceManager. The YARN architecture is shown in Figure 3.2.8.

The ResourceManager and the NodeManager form the data-computation framework.

ResourceManager The ResourceManager organizes the job execution on the cluster. It is made up by two components, the *Scheduler* and the ApplicationsManager.

The Scheduler allocates resources in *Containers* across the cluster for pending jobs given a number of constraints like available memory or CPU and application requirements. It's functionality is decided by policies which are pluggable onto it and

Figure 3.2.7: From MapReduce to YARN [11]


HADOOP 1.0
MapReduce
(cluster resource management & data processing)
HDFS
(redundant, reliable storage)
HADOOP 2.0
MapReduce
(data processing)
Others
(data processing)
YARN
(cluster resource management)
HDFS
(redundant, reliable storage)

define the way resources should be used.

The ApplicationsManager accepts the job submissions and initiates the ApplicationMaster on the Container where the application is going to be executed. ApplicationMasters are the main negotiators of resources from the Scheduler and track applications status and progress.

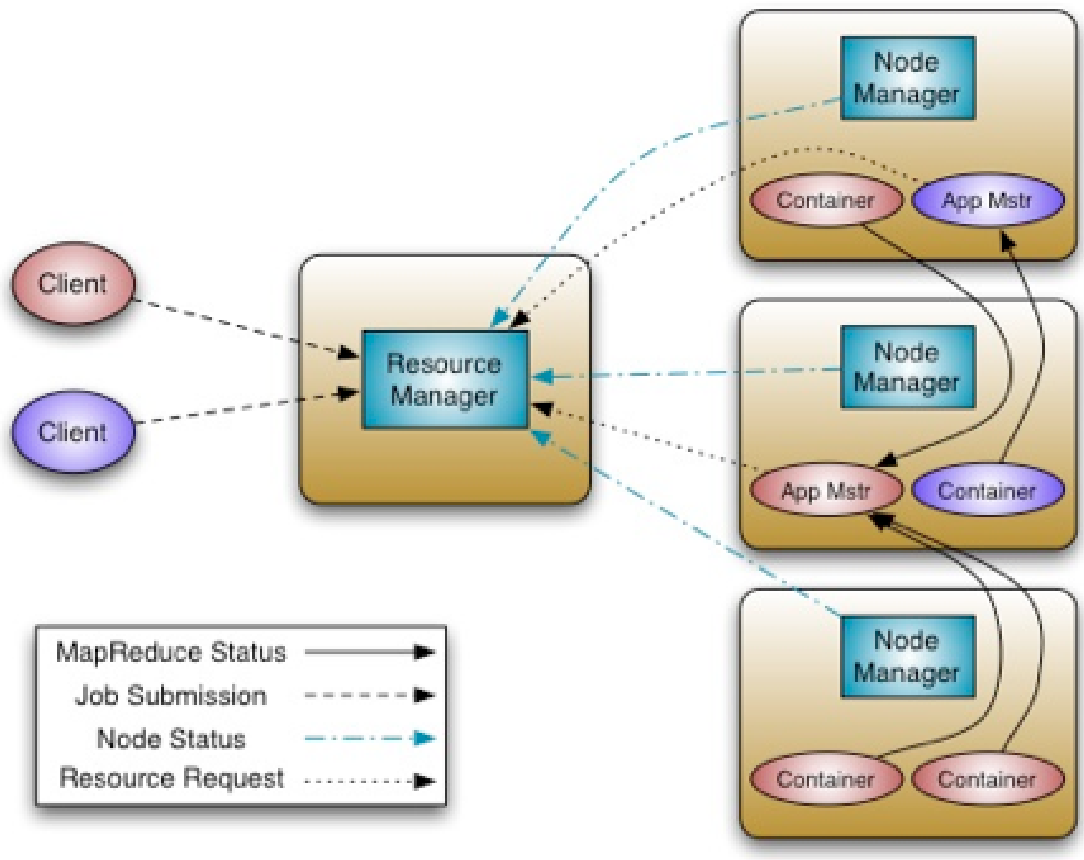
NodeManager The NodeManagers run per-node and they are responsible for the containers running on them, reporting their resource consumption back to the Scheduler. They keep track of the status of the node they run on, log events and monitor the jobs which are executed on it.

3.3 Cluster Setup

In order to go through the setup of a Hadoop cluster it is needed to setup and configure every single machine which is going to be part of it. Since Hadoop is written on Java, it is required that is is installed in all of the nodes. While the machines' hardware specifications may vary it highly recommended that they are all using the same version of Java. In addition, they all need to have an SSH server installed and running because this is how Hadoop communicates with every node on the cluster.

On the first a single node setup needs to take place, followed up by the multi-node configuration.

Figure 3.2.8: YARN [6]



3.3.1 Single Node Setup

The proposed setup includes the following:

- Java Oracle 1.6
- Ubuntu 12.04 Server
- Hadoop 2.2.0

User and group In favour of security, user management and permissions it is quite useful to add a user account and dedicated to running Hadoop. We will create a hadoop group and a user hduser and add him to the hadoop and sudo groups. To achieve that execute the following:

```
$ sudo addgroup hadoop
$ sudo adduser --ingroup hadoop sudo hduser
$ su - hduser
```

SSH configuration After going through the process of creating a user and group, SSH access needs to be configured. Initially an RSA private/public key needs to be created for enabling public key authentication. Creating a new pair goes like that:

```
$ ssh-keygen -t rsa -P ""
```

Now that the key is generated access can be granted to this node by adding this key to every slave's authorized keys list. Since the master node, except from NameNode, acts as a DataNode too, the node needs to have ssh access on itself through localhost, so its public key had to be into its own authorized key list. We append our private key on this list like that:

```
$ cat $HOME/.ssh/id_rsa.pub > $HOME/.ssh/authorized_keys
```

Finally we have to test ssh access on the node and save the node's host key fingerprint on the `known_hosts` file.

```
$ ssh localhost
```

Network configuration Since our node is going to run both as master and slave, given that we are on a single node setup, we have to set our hosts file to refer to our machine's local IP address as master. In linux this file is `/etc/hosts` and we edit it as follows:

```
# 127.0.0.1 localhost // we disable the localhost reference

192.168.0.1 master // we set our local network IP to master which may vary
...
```

Downloading Hadoop Now the right version of Hadoop had to be downloaded and extracted into the hduser's home folder. From the Hadoop website we select the desired version and download the precompiled package. It can be downloaded and extracted executing:

```
wget \
http://apache.mirror.digionline.de/hadoop/common/hadoop-2.2.0/hadoop-2.2.0.tar.gz
tar -xzvf hadoop-2.2.0.tar.gz
```

Hadoop configuration After we are done with the previous steps it is now time to write our configuration files. First we have to define the JAVA PATH on the `hadoop-env.sh` file which is in the `hadoop-2.0.0/etc/hadoop` folder alongside the rest of the configuration files. We add this line in the file:

```
export JAVA_HOME=/usr/lib/jvm/java-6
```

The above path depends on every platform and care must be taken in order to point to the right directory.

The main configuration files of Hadoop are XML documents and follow the `*-site.xml` naming convention. These contain a configuration root, under which various properties are added assigning the desired values to each parameter. There are 4 main files which refer to each one of Hadoop's main components. These are the following:

- `core-site.xml` – contains the values for the core Hadoop properties
- `hdfs-site.xml` – contains configuration for HDFS
- `mapred-site.xml` – sets the default MapReduce Jobs behaviour
- `yarn-site.xml` – contains the configuration for the ResourceManager

Before adding the configuration to those files, it is appropriate to create the `tmp` and `hdfs` directories. The first one is needed for running Hadoop, and the latter is where HDFS allocates its space. They can be placed in a directory named `hadoop`:

```
$ mkdir -p $HOME/hadoop/tmp
$ mkdir -p $HOME/hadoop/hdfs
```

First we edit the `core-site.xml` file defining the HDFS port and the tmp.dir directory. It should look like this:

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://master:9000</value>
  </property>
  <property>
```

```
    <name>hadoop.tmp.dir</name>
    <value>/home/hduser/hadoop/tmp</value>
  </property>
</configuration>
```

Second comes the `hdfs-site.xml` where we define the desired replication, the HDFS directory and disabling folder permissions:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>2</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>/home/hduser/hadoop/hdfs</value>
  </property>
  <property>
     <name>dfs.permissions</name>
     <value>false</value>
   </property>
</configuration>
```

Next comes the `mapred-site.xml` where we add the following configuration:

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
      </property>
    <property>
        <name>mapreduce.job.tracker</name>
        <value>hdfs://master:9001</value>
        <final>true</final>
    </property>
</configuration>
```

Finally the `yarn-site.xml` should contain the ResourceManager ports and a couple more properties:

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>user.name</name>
        <value>hduser</value>
    </property>
    <property>
        <name>yarn.resourcemanager.address</name>
        <value>master:54311</value>
    </property>

    <property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value>master:54312</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>master:54314</value>
    </property>
    <property>
        <name>mapred.job.tracker</name>
        <value>master</value>
    </property>
</configuration>
```

This is an example set of configuration to get your node up and running. Hadoop configuration has many parameters which can be tweaked depending on the available hardware, the desired behaviour, network topology and many more. There a great space for optimization and experimentation which is out of our scope.

Formatting the NameNode After our configuration is in place, we can format the NameNode executing the format command from our Hadoop's installation root directory:

```
$ bin/hdfs namenode -format
```

Please note that this command will erase all the data in the HDFS.

Starting the Single Node Cluster Given the successful completion of the previous steps, it is now possible to start the single node cluster. First we start the HDFS:

```
$ sbin/start-dfs.sh
```

This command will initiate the NameNode the DataNode and the SecondaryNameNode. The output should look like this:

```
Starting namenodes on [master]
master: starting namenode, logging to /home/hduser/hadoop-2.2.0/...
master: starting datanode, logging to /home/hduser/hadoop-2.2.0/...
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /home/hduser/hadoop-2.2.0/...
```

Executing the command `jps` will return us the following running Java processes:

```
615 DataNode
954 SecondaryNameNode
364 NameNode
```

In case that something is not this way something is wrong. In order to find out the cause of the problem we can have a look at the logs directory.

After that we can start YARN, which initiates the ResourceManager and the NodeManager:

```
$ sbin/start-yarn.sh
starting yarn daemons
starting resourcemanager, logging to /home/hduser/hadoop-2.2.0/logs/...
master: starting nodemanager, logging to /home/hduser/hadoop-2.2.0/logs/...
```

Last, but not least, the `proxyserver` and the `historyserver` need to be started with the following commands:

```
$ ./sbin/yarn-daemon.sh start proxyserver
$ ./sbin/mr-jobhistory-daemon.sh start historyserver
```

Now all of the required services must be running and `jps` should return them:

```
### JobHistoryServer
### ResourceManager
### DataNode
### SecondaryNameNode
### NameNode
### NodeManager
```

Hadoop also provides some web interfaces to monitor the status of the cluster. The default interface can be accesed at port 8088. On a local machine it can be accessed at localhost:8088. Through that someone can get the status of his nodes, applications and logs among others. On port 8042 node related info can be found, while ports 50070 and 50090 show information related to the NameNode and the Secondary NameNode respectively.

Running a MapReduce Job We are now ready to run out first MapReduce job. We will utilize one of the provided MapReduce examples, the wordcount application. This job reads a file from HDFS, puts all the words in key value pairs, where key is the word and value is the number of occurrences. In the map step all words are assigned a value of `one`, while in the reduce step all the values per word are summed up to give the final output.

For our demonstration we will download J. Verne's `A Journey to the Center of the World`, copy it into HDFS and then run the job.

```
$ wget http://www.textfiles.com/etext/FICTION/center_earth

# Copy the downloaded file to HDFS
$ $HADOOP_HOME/bin/hdfs dfs -copyFromLocal ./center_earth /
$ $HADOOP_HOME/bin/hdfs dfs -ls /
# Output
Found 1 items
-rw-r--r--   2 hduser supergroup     489319 2013-06-27 14:44 /center_earth
```

We can now run the job by pointing to the available MapReduce example jar and providing as arguments the desired app the file path on HDFS and the output path:

```
$ $HADOOP_HOME/bin/hadoop jar \
    ./share/hadoop/mapreduce/hadoop-mapreduce-examples-2.2.0.jar \
    wordcount /center_earth /wordcount_output
```

After the job is successfully completed we can see the results executing:

```
$ $HADOOP_HOME/bin/hdfs dfs -cat /wordcount_output/part-r-00000
# output
...
you?"   4
young   4
young,  2
your    66
yourself    2
yourself,   5
yourself-   1
yourself.   2
...
```

To stop our cluster we can the equivalent ones for starting:

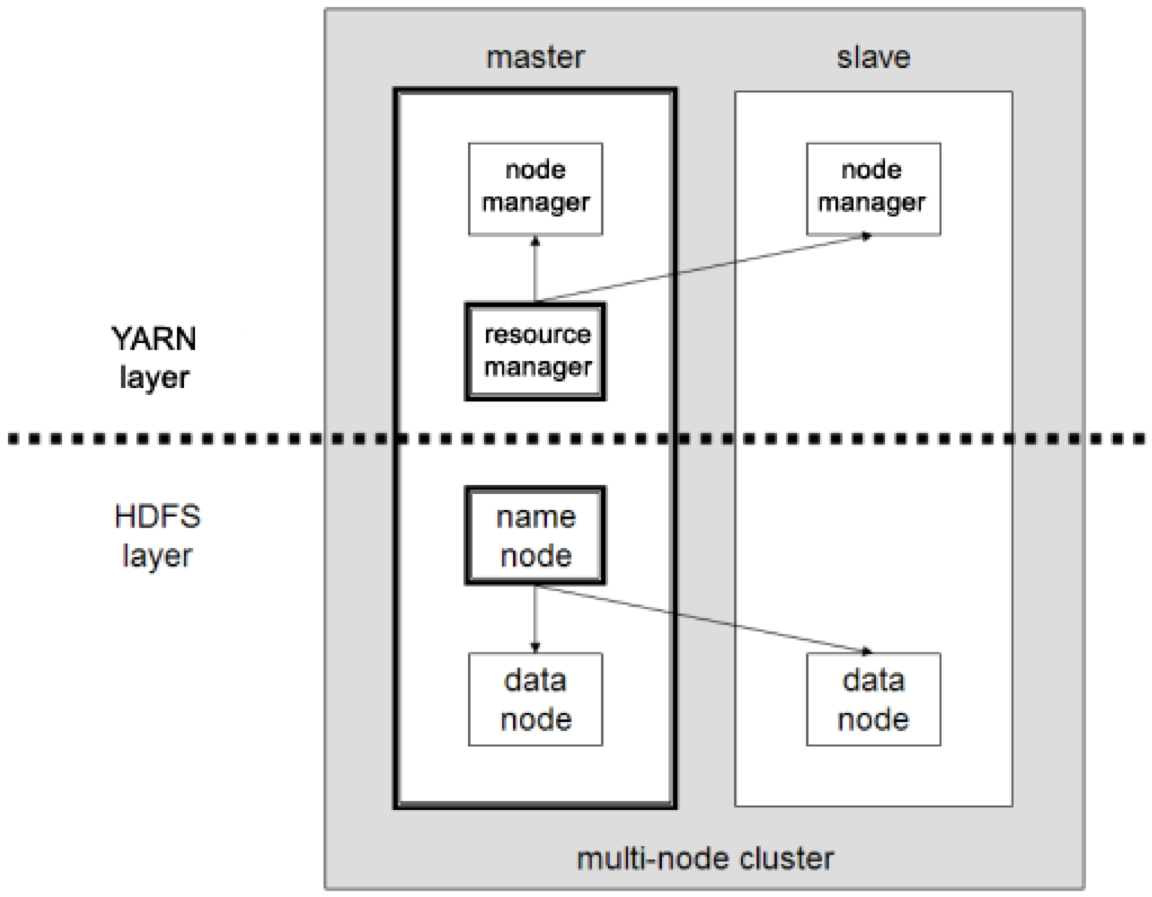
```
$ $HADOOP_HOME/sbin/stop-dfs.sh
$ $HADOOP_HOME/sbin/stop-yarn.sh
```

This concludes the steps needed to setup a single node cluster. Of course it poses only for demonstration purposes since it does not have any value running Hadoop on one machine, but it is the first step to test a single node and then add it on a cluster. On the next section we will describe how to set a real multi node cluster.

3.3.2 Multi Node Cluster Setup

For setting up a cluster it is important to have multiple single node machines set up and running. Following the instructions of the previous sections and setting up each individual machine it is now quite easy to merge the changes and start our cluster. Our target is having a setup which is illustrated in Figure 3.3.1.

Figure 3.3.1: Multi Node Cluster Overview [21]



In our case we suppose that all nodes are identical thus sharing the same configuration. It is important that our nodes run on the same local network and have a simplified network structure. For starting we first have to stop all the nodes, in case they are running, in the way described in the end of the previous section.

Network configuration As in the previous section we have to define the IP address of our master and slave in the `/etc/hosts` file. We append the IP and the role of each node on every machine's configuration:

```
192.168.0.1     master
```

```
192.168.0.2    slave0
192.168.0.3    slave1
...
192.168.0.N    slaveN
```

In the single node setup, hduser needs to have ssh access to its own user account on the same machine. On a multi node cluster it is needed that hduser of the master node can access the hduser account on each single slave node with password-less login. To do that you have to copy the master's public key on each slaves `authorized_keys` file:

```
$ ssh-copy-id -i $HOME/.ssh/id_rsa.pub hduser@slaveX
```

This will prompt the user to enter the password on each machine. After the public key resides in every slaves `authorized_keys` file, the it should be able to connect from the master's hduser account to every slave:

```
$ ssh hduser@slaveX
# enter "yes" to verify that you want to add every slave's RSA key fingerprint
to the known_hosts file
```

Cluster Configuration The only change we need to make on our cluster now is defining the list of slaves in the master's slave file, which is into the default hadoop's configuration folder under `$HADOOP_HOME/etc/hadoop/slaves`. In the there we define the list of our slaves by their hostname:

```
master
slave0
slave1
...
slaveN
```

Remember that the above action needs to be performed only on the master node.

After everything is set we can start our cluster from the master node executing the same commands as in the single node setup. The difference observed is that the daemons now start not only in the master node but on the slaves too. After starting the cluster, the DataNode and the NodeManager should run in every slave. This can always be verifies with the `jps` command.

Slave nodes can always be added on the fly on a running cluster without further configuration by manually starting on them the DataNode and the NodeManager:

```
$ $HADOOP_HOME/sbin/hadoop-daemon.sh start datanode
$ $HADOOP_HOME/sbin/yarn-daemon.sh start nodemanager
```

They can be removed in the same way replacing `start` with `stop` in the aforementioned commands.

Managing a big number of nodes is not a trivial task, from setting them up to maintaining them. For this purpose there are many configuration management tools that automate those procedures and reduce the repetition overhead. One can follow the above steps and execute them on many machines using tools like `pssh`. The best solution is using tools like chef or puppet which distribute configuration and run automation scripts across many servers.

Chapter 4

Parallel K-Means on MapReduce

4.1 Introduction

Following the introduction on the K-means algorithm and the MapReduce model, we now have an overview of our main ingredients and can go forward with combining them in order to parallelize the heavyweight task of clustering big amounts of data. Implementing the Parallel K-Means algorithm using MapReduce model is quite straightforward since its two main steps of assigning a vector to the closest center and then recalculating the new centers, can be implemented on the Map and Reduce step respectively.

The proposed implementation is built using the `mapreduce` APIs which are bundled with the Hadoop distribution. This is the latest version of the MapReduce library which deprecates the old `mapred` APIs. The `mapred` APIs are still compatible with the newer versions of Hadoop, however, while still being mostly compatible with older versions of Hadoop, the new `mapreduce` APIs are not guaranteeing backwards compatibility. It is based on the implementation of Thomas Jungblut[16].

4.2 Implementation Overview

As discussed, the K-means algorithm initially reads the input data from an external source, creates N random cluster centers, and then assigns the closest center to each vector, generating a cluster. Afterwards a new center for each cluster is calculated and is written back on the disk. Reading the data and initiating the centers consists the *Job Level* of the algorithm, the assignment to the closest centers is implemented in the *Map* step, and the computation of the new centers is done in the *Reduce* step. The Map and Reduce steps are repeated until the end condition is reached.

4.2.1 Job Level

The Job Level is responsible for initializing the application, from reading the input data and creating the required resources, to handling the job execution by setting the configuration and iterating over them. It starts by parsing the given arguments, like the path of the input data, the amount of lines to be processed, the desired number of clusters and the maximum amount of iterations. Those are passed from the main function to the run method, which in turn, will parse the input file and create two `Sequence files`.

Sequence Files are "flat files consisting of binary key/value pairs." [SequenceFile], which are extensively used as input/output files in MapReduce in the user level, but also internally for the temporary outputs of maps. These files are usually compressed and stored in HDFS, making them ideal for serializing large files and storing them in Hadoop's filesystem. The first file contains the cluster centers and the second one the vectors of our data set. Both share the same key/value structure, where the key is a `Text` object holding a unique name for each cluster, and value is a `Vector` type representing our vectors. These two files is were, and how, our data are stored and used by the Map and Reduce Steps. In the beginning, the file containing the vectors of our data set has an empty string as the name of the cluster as it has none assigned to it yet. These files are created, after a cleanup of any files created on previous runs is done.

It is suitable here to describe the two classes used as our key/value pairs, since they play a major role for the whole implementation. As mentioned before, Map and Reduce tasks iterate over key/value pairs. Any type which is going to be used as a `key` in the Hadoop MapReduce framework should implement the `WritableComparable` interface, while the `values` should at least implement the `Writeable` interface. The `Text` type, is distributed with Hadoop and it stores, as its name suggests, text. The `Vector` type is a custom implementation of the Writeable interface, and stores vectors as tables of `double` values.

After the initial step is done, the main algorithm, which is repeated until the maximum number of iterations is reached, takes over. In every iteration the following actions are performed:

- Configuration initialization – reads the default Hadoop configuration and overwrites the desired parameters or sets new ones.
- Setting the input file – since we are iterating over the same data the input of every job is the output of the previous one.
- Initializing the Job – set our Map and Reduce classes, the input and output format, and the output key/value pair types.
- Logging – printing helpful information about our jobs' progress, which is also a quite useful tool for debugging.

4.2.2 Map Step

The Map step is our K-means *assignment* step. Our `KMeansMapper` class extends the `Mapper` class of the mapreduce API and it is done in two steps. Initially the file containing the cluster centers is read, and those are stored in a list of `ClusterCenter` objects. The ClusterCenter class, is a helper class for representing our cluster center objects, holding the name of the cluster and its center, and providing the distance measurement function.

Reading the cluster centers is done in the `setup` method which is called once at the beginning of the task. After this is done we go forward to the `map` method, which is the one that will actually loop over our vectors and emit them to the closest cluster using the aforementioned distance function for calculating their proximity based on the Euclidean distance formula. Since the map step iterates over each data point, at this step our algorithm is parallelized over them.

The output of our map step are, as it should be clear by now, key/value pairs, were key is the name of the assigned cluster and value is the actual vector. The code of the mapper can be found in the Listing A.1.

4.2.3 Reduce Step

In the Reduce step we are going to get the output of the mapper shuffled, sorted and grouped by the key. The `KMeansReducer` extends the `Reducer` class and performs the reduce step and the final cleanup. Since we iterate over our keys, which are our clusters, the algorithm now is parallelized over them and we are going to have a list of the vectors for every cluster, that we will average to get our new cluster centers. After the new centers are calculated they are added in a ClusterCenter list.

The output of the reduce function is stored in a SequenceFile in the file system, containing all of our vectors with their assigned cluster name as their key. Furthermore, when the reduce step is finished the final `cleanup` function is called, which loops over the ClusterCenter list created in the reduce step, and stores the new cluster centers with their name and vector in our center SequenceFile.

The output of the this job alongside the updated center file are going to be our new input files for the next iteration, repeating this process until the end condition. The reducer code is in Listing A.2.

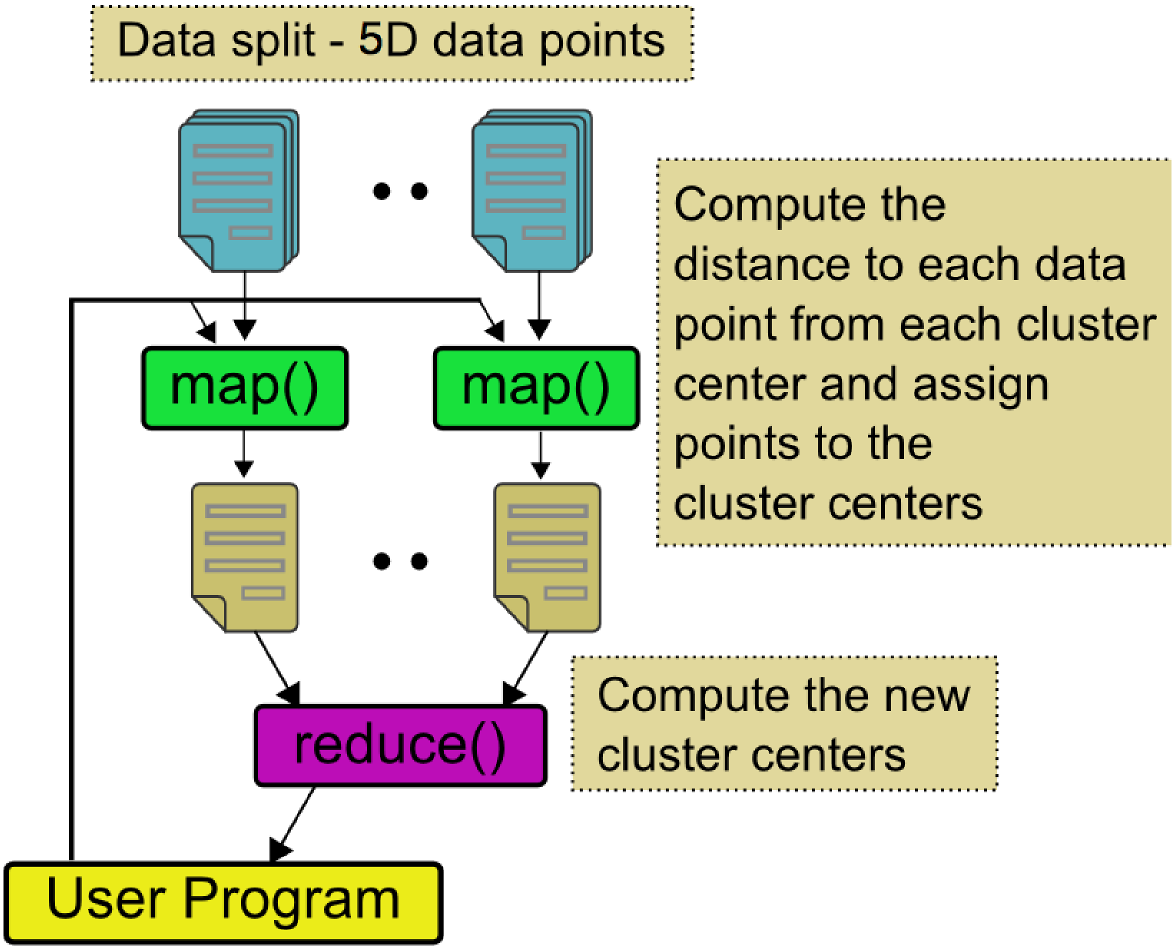
4.3 Job Execution

Since we implemented the map and reduce functions and set the configuration for executing the job, we can run our application. YARN will take care of the task assignment, by splitting it to mapppers, combining their output and then sending it to the reducer. The processes overview can be seen in Figure 4.3.1.

There are some considerations regarding the implementation of K-means on MapReduce. First of all, iterating mapreduce jobs is not a standard formulation and not the best way to work with repetitive tasks. For each iteration the data needs to be written and then read back from the filesystem, which adds a significant overhead, especially in cases of large data sets. On the other hand big data sets are the case were Hadoop shines, and mappers need to get as many data points in order to perform in way that is beneficial over a non parallel K-means implementation.

In the next chapter we are going to examine the behaviour and performance of the our Parallel K-means algorithm, using a showcase setup and a provided amazon data set, and discuss the proposed solution, the problems that came up, and what would be the ideal test cases and scenarios.

Figure 4.3.1: K-means on MapReduce


Data split - 5D data points
map()
map()
Compute the distance to each data point from each cluster center and assign points to the cluster centers
reduce()
Compute the new cluster centers
User Program

Chapter 5

MLSP Amazon Access Data Clustering

For demonstrating the use and function of our Parallel K-means on Hadoop we are use the Amazon provided data set for the Machine Learning for Signal Processing (MLSP) competition of 2012[19]. The infrastructure on which Hadoop will run, are virtual instances from *okeanos*, GRNET's cloud service, available for all the members of the Greek research and academic community.

We are going to utilize the given data and resources on different scenarios and compare the results. We will first present the available data and how we will process them in order to initialize our data points for our K-means algorithm.

5.1 Amazon Access Data

The Amazon Access data set, was made available by Amazon's Information Security organization for use in academic models against real industry data. This gives as a taste of the nature and the size of the information described as Big Data and the challenges that are posed when it comes to processing them.

The data set consists from 4 files containing historical data collected from 2010/2011. These files hold different kind of data, which are samples from the access history of Amazon employees on the company's resources, stored in `CSV` (comma separated values) format.

The first file is `amzn-anon-access-samples-user-profile-history.csv` holding a set of attributes describing the employee profiles and the history of changes made on them. Those are a timestamp of the change on an employee's profile, his unique ID, his manager's ID, and ID's related to his role, department name, title, role family and role code. This file has 8116040 rows and it's size is approximately 600 mb. The next three files are relatively smaller in size and contain data like transaction history, and snapshots of changes on specific dates, as well as a validation data file.

These files were used as input data for the MLSP competition of 2012, for applications of collaborative filtering and data clustering. As it was mentioned on the first chapter for extracting useful information from a data set and running a clustering algorithm that will eventually produce valuable results the data needs to be carefully studied and preprocessed for normalizing the input for the algorithm. This is out of the scope of this project were the main focus is the implementation and examination of a Parallel K-means algorithm on MapReduce.

For this purpose, only the first file will be utilized, of whose values are going to be used as random data for our vectors that will constitute our test data set. While completely random values could be generated and used, we opted in using some real data for demonstrating the storage and processing capabilities of Hadoop. It is preferred over the rest of the files because it is more rich in information and bigger in size. However we will see that its size is still quite small for the scale that Hadoop is supposed to process.

5.1.1 Processing the Amazon Data

The Amazon access file that we will use as seed for our data points has the following structure:

```
DATE,EMPLOYEE_ID,MGR_ID,ROLE_ROLLUP_1,ROLE_ROLLUP_2,ROLE_DEPTNAME,
                        ROLE_TITLE,ROLE_FAMILY_DESC,ROLE_FAMILY,ROLE_CODE
2011-01-03,63936,37250,119611,118350,117895,118194,117913,117887,118196
2010-12-13,63936,37250,119611,118350,117895,118194,117913,117887,118196
2011-01-17,63936,37250,119611,118350,117895,118194,117913,117887,118196
2010-11-29,119612,547,117926,117927,117941,117885,117913,117887,117888
2011-06-06,95028,21184,118095,118096,118008,117885,117913,117887,117888
2011-05-09,95028,21184,118095,118096,118008,117885,117913,117887,117888
```

The fields used to form our vectors are from `ROLE_ROLLUP_1` to `ROLE_FAMILY_DESC`, producing vectors of size 5. Since the file holds regular snapshots of each employee in different dates, there is a high possibility of the values being repeated over and over if no change happens on a users profile. Using them the way they are would hamper our data set and produce poor results, since a very high percentage of vectors would have been repeated over and over again. For that reason the following modifications are performed in order to normalize the data.

- Use the last 3 characters of the parsed fields to create integers in the range 100–999
- Multiply them with a random number to differentiate the values producing vertices in the range 0–999

Each vector will be stored in a new created SequenceFile which will be used as the data point input of first MapReduce job. The number of the size of the data set is required as a parameter to limit the amount of vectors to be processed.

5.1.2 Cluster Center Initialization

The amount of clusters is set to a default value of 10. However this value is quite small for the size of our data set. For this reason the desired amount of clusters can be defined passed a command line argument. Given the size of the desired data set, *N* random clusters will be picked from it, serve as the initial cluster centers.

5.2 Experimental Setup

The Hadoop cluster is initially set on 2 virtual machines, running on GRNET's cloud service okeanos. The available resources are 3 machines that share a total of 16 CPUs, 22 GBs of memory and 260 GB of hard drive. The machines are running Ubuntu Server 12.04.3 LTS and have installed the Java version 1.6.0_45 and Hadoop 2.2.0. In the beginning, the first machine used as the master node has 4 GBs of RAM and 60 GB of hard drive, while the slave has the remaining 2 GBs of RAM and 40 GBs of HDD. Later we scale them up to 8 cores and 8GBs of memory each to compare the performance gain on bigger machines.

These virtual machines are set as in the guide at Section 3.3, forming a multi-node Hadoop cluster. Is should be noted that the storage and computational capacity of these machines, and the size of this cluster, is extremely small compared to real industry use cases that are utilizing hundreds to thousands of bare metal servers with much more memory, CPUs and HDDs. The purpose here is to showcase the parallelization of the K-means algorithm and investigate the potential of Hadoop and the MapReduce framework.

5.3 Results

In the this section the results of a set of different parameters will be presented and discussed. The proposed implementation will be run on the above setup using the amazon data set as input and tested against the following settings:

- Sample size – the amount of records used for the test
- Block size – the HDFS block size used for storing the data defaults to 32mb
- Number of clusters – default 10
- Number of iterations – default 5

The block size is a very important factor for the performance of the application since it is the decisive parameter, alongside the size of the input file, for the number of splits and thus the amount of mappers per job. A compressed sequence file containing 1 million vectors is approximately 60 MBs. Having a block size of 64 MBs, which is the default size, will produce one map per job which eventually means that our algorithm is not parallelized.

Running the application is done with the following command:

```
$ bin/hadoop jar kmmr-0.0.1-SNAPSHOT.jar KMeansClusteringJob \
    /amazon/amzn-anon-access-samples-user-profile-history.csv \
    5000000 64 500 5
```

The command `bin/hadoop` is ask to load a `jar` file, where we pass our algorithm archived snapshot, and we execute the `KMeansClusteringJob`. We then pass the path of our input file on HDFS, the amount of samples we want to process, the block size in megabytes, the number of cluster centers and the amount of iterations.

In the output we can many info related to the job initialization, out log messages, a progress report of every job and some results for each one of them. In the end, all the cluster centers are printed, and a text file containing all the vectors with their assigned cluster is stored in HDFS. The cluster centers result looks can be seen here:

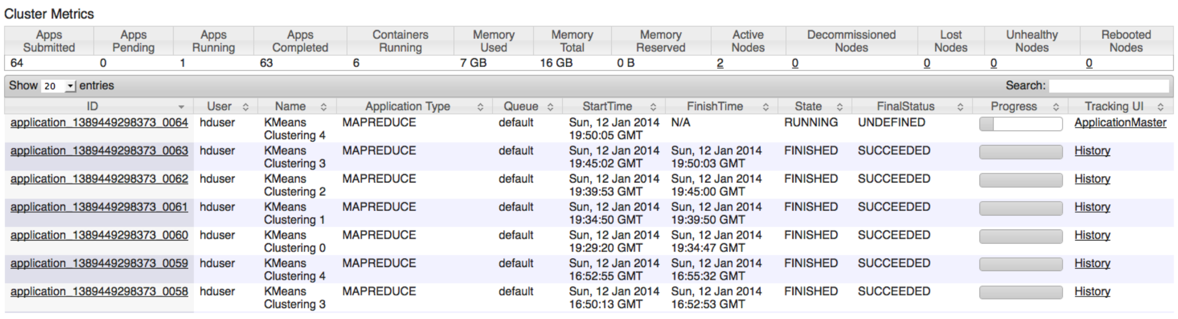
```
14/01/12 21:55:24 INFO KMeansClusteringJob: cluster133 /
                    [494.12, 442.42, 510.78, 525.50, 506.75]
14/01/12 21:55:24 INFO KMeansClusteringJob: cluster134 /
                    [566.46, 526.00, 557.36, 563.19, 549.39]
14/01/12 21:55:24 INFO KMeansClusteringJob: cluster135 /
                    [122.40, 112.70, 199.38, 208.80, 194.09]
```

The progress report looks like this:

```
14/01/12 18:18:27 INFO mapreduce.Job:  map 75% reduce 8%
14/01/12 18:18:35 INFO mapreduce.Job:  map 79% reduce 8%
14/01/12 18:18:39 INFO mapreduce.Job:  map 79% reduce 13%
14/01/12 18:18:45 INFO mapreduce.Job:  map 83% reduce 13%
14/01/12 18:18:46 INFO mapreduce.Job:  map 100% reduce 13%
14/01/12 18:18:49 INFO mapreduce.Job:  map 100% reduce 17%
14/01/12 18:18:52 INFO mapreduce.Job:  map 100% reduce 25%
```

It is clear here that after some mappers are done the results are already sent to the reducer which progresses as the rest of the mappers still work. The progress of the jobs can be also seen from the web interface which is available on HTTP port 8088 and looks like in Figure 5.3.1.

Figure 5.3.1: Hadoop Web Interface



Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
64	0	1	63	6	7 GB	16 GB	0 B	2	0	0	0	0

Show 20 entries

Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1389449298373_0064	hduser	KMeans Clustering 4	MAPREDUCE	default	Sun, 12 Jan 2014 19:50:05 GMT	N/A	RUNNING	UNDEFINED		ApplicationMaster
application_1389449298373_0063	hduser	KMeans Clustering 3	MAPREDUCE	default	Sun, 12 Jan 2014 19:45:02 GMT	Sun, 12 Jan 2014 19:50:03 GMT	FINISHED	SUCCEEDED		History
application_1389449298373_0062	hduser	KMeans Clustering 2	MAPREDUCE	default	Sun, 12 Jan 2014 19:39:53 GMT	Sun, 12 Jan 2014 19:45:00 GMT	FINISHED	SUCCEEDED		History
application_1389449298373_0061	hduser	KMeans Clustering 1	MAPREDUCE	default	Sun, 12 Jan 2014 19:34:50 GMT	Sun, 12 Jan 2014 19:39:50 GMT	FINISHED	SUCCEEDED		History
application_1389449298373_0060	hduser	KMeans Clustering 0	MAPREDUCE	default	Sun, 12 Jan 2014 19:29:20 GMT	Sun, 12 Jan 2014 19:34:47 GMT	FINISHED	SUCCEEDED		History
application_1389449298373_0059	hduser	KMeans Clustering 4	MAPREDUCE	default	Sun, 12 Jan 2014 16:52:55 GMT	Sun, 12 Jan 2014 16:55:32 GMT	FINISHED	SUCCEEDED		History
application_1389449298373_0058	hduser	KMeans Clustering 3	MAPREDUCE	default	Sun, 12 Jan 2014 16:50:13 GMT	Sun, 12 Jan 2014 16:52:53 GMT	FINISHED	SUCCEEDED		History

The results of executing the algorithm in the initial setup with two small nodes, are presented for some cases depending on the aforementioned parameters on Table 5.3.1. Significant factors affecting the overall time, is that in every run CSV file containing the amazon data should be parsed and written as the sequence file with

Table 5.3.1: Results – Cluster with Master node with 1 core and 4 GBs of memory and Slave node with 1 core and 2 GBs of memory

	Sample Size	Block Size	Clusters	Iterations	Mappers	Avg. Job Time	Total Time
Run 1	1 million	2 MB	100	5	30	5 min 2 sec	27 min 25 sec
Run 2	1 million	16 MB	100	5	4	2 min 36 sec	14 min 27 sec
Run 3	1 million	32 MB	100	5	2	2 min 9 sec	12 min 33 sec
Run 4	1 million	64 MB	100	5	1	1 min 56 sec	11 min 6 sec
Run 5	5 million	32 MB	500	5	10	4 min 34 sec	28 min 45 sec
Run 6	5 million	64 MB	500	5	10	5 min 13 sec	30 min 1 sec
Run 7	8 million	32 MB	500	5	16	6 min 23 sec	39 min 28 sec
Run 8	8 million	64 MB	500	5	8	6 min 7 sec	37 min 32 sec

our vectors. In the end of the run the application writes out in a text file the output of the algorithm, which is also time consuming, depending on the sample size. These can add approximately minutes to the overall time. Additionally, every job needs approximately 1 minute to be initialized. It should be noted that every job has one reducer.

Having in mind these factors we can make some conclusions about our results. First, it is quite clear that the block size is inversely proportional to the number of mappers. As it can be seen, a quite small block size of 2 MBs – which is an unusual and not recommended configuration – increases the number of mappers that process small chunks of our data, having a quite dramatic effect on the performance of the algorithm. Interestingly enough for the same amount of vectors and block size of 64 MBs, we only have one mapper – meaning no parallelization – and the performance is slightly better than with 2 splits of the 32 MBs block size run and parallelization. This is due to the overhead of managing the job and splitting it over too machines that are anyway quite slow, plus that the amount of data per mapper is still lower than the recommended 64 MB block size. The same thing can be observed when comparing run 7 and 8, where completing a job with 8 mappers is slightly faster than processing with the same amount of data but half the block size and twice the mappers.

Scaling up our cluster, which translates to increasing the size of the machines in terms of computational and power, is fairly easy on cloud services like *okeanos*. Bringing the nodes from 1 to 8 cores and the memory to 8 GBs shows significants performance gains as expected, that are shown in table 5.3.2

Table 5.3.2: Results – Cluster with 1 Master and 1 Slave nodes with 8 cores and 8 GBs of memory

Sample Size	Block Size	Clusters	Iterations	Mappers	Avg. Job Time	Total Time
8 million	64 MB	500	5	8	3 min 23 sec	19 min 42 sec

Depending on the size of our data we have to decide whether to scale vertically, by growing up our machines, or horizontally, by adding new nodes on our cluster. Since the proposed amount of mappers per node ranges from ten to hundred it makes to sense to parallelize the algorithm to more machines if the number of expected mappers does not exceed this range.

Once again it should be noted that the results are not the most representative but are mostly showcasing the setup of a Hadoop cluster and parallelizing the K-means algorithm. Hadoop runs on multi-core machines with high memory and hundreds of machines, operating on terabytes of data.

Another serious consideration, is if MapReduce is the ideal way of implementing iterative algorithms which is the case for most Machine Learning and Artificial Intelligence applications. This is mostly due to the fact that each mapreduce job reads and writes the data back to the hard drive, instead of being able to keep them in-memory and every reducers feeding directly the next mapper.

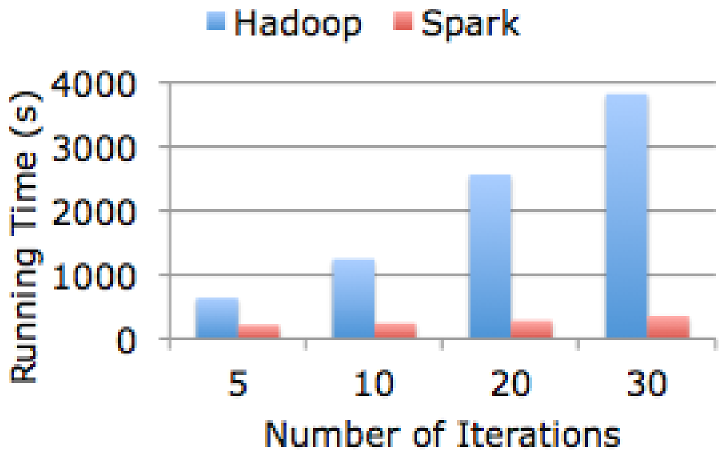
5.4 Alternatives

In this section the most prominent implementations of K-means and more machine learning algorithms, will be presented.

Apache Mahout [8] An official Apache Hadoop subproject offering distributed, or so called scalable, implementations of machine learning algorithms, covering a wide range of applications in the fields of clustering, collaborative filtering and classification. It is the most well known scalable machine learning library. As it is claimed on the official home page of the project, it is “scalable to reasonably large data sets”.

Apache Spark [9] Spark is an Apache incubator project, initiated by the Berkeley AMPLab. Spark has its own runtime environment, but can run on top of other cluster managers including Hadoop. In Figure 5.4.1 a comparison against mapreduce can be seen, as the number of iterations increases. Spark applications can be written in Java, Scala or Python. It has some ready examples including parallel K-means implementations for all the three aforementioned programming languages.

Figure 5.4.1: Hadoop MapReduce / Spark comparison [9]

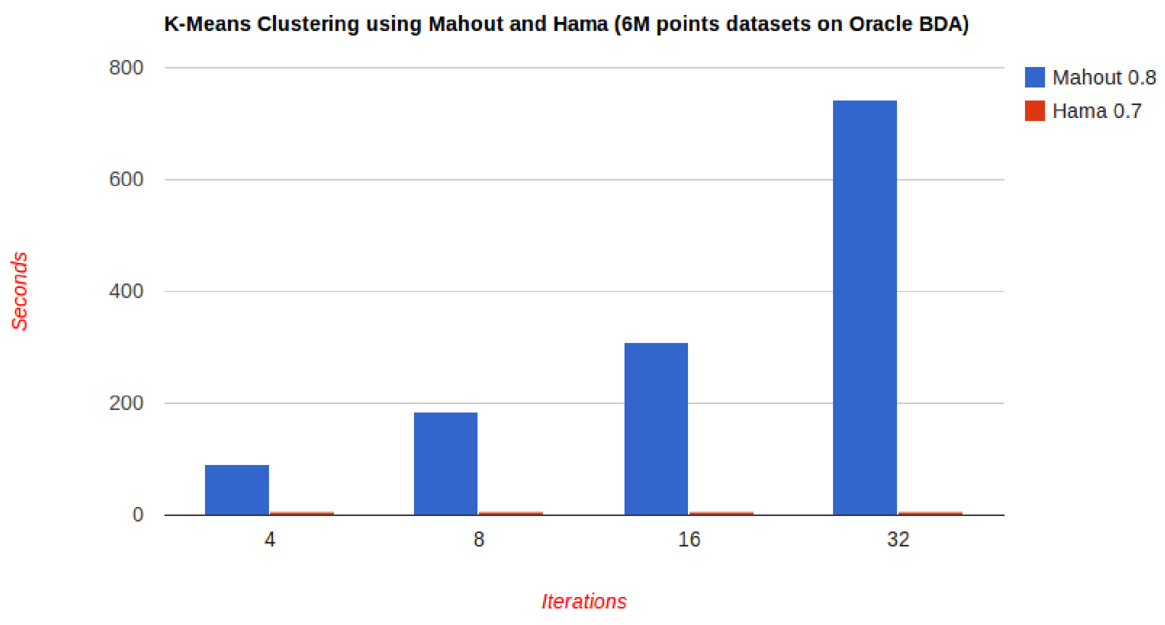


A more comprehensive Machine Learning library running on Spark is the MLbase library, which is also developed by the Berkeley AMPLab.

Apache Hama [7] Yet another Apache project, Hama is an analytics tool running on top Hadoop but is not implemented on the MapReduce model but uses the Bulk Synchronous Parallel (BSP) model which is more effective considering the iterative nature of machine learning and graph algorithms. Figure 5.4.2 shows a comparison against Mahout.

Stratosphere [22] A European Union project developed in the TU Berlin, is Stratosphere, offering *next-generation Big Data Analytics*. Stratosphere extends the MapReduce model adding more operators besides, map, reduce and combine, including joins, unions and iterations among others, enabling advance data flow graphs beyond the classic map -> reduce process. In-memory data transfers increase performance by significantly reducing disk and network I/O. Stratosphere offers both Java and Scala APIs and can run on top of YARN. K-means is offered as one of its example applications, suggesting significantly better performance compared to standard MapReduce implementations.

Figure 5.4.2: Hama / Mahout comparison [7]



Chapter 6

Conclusions

We presented an implementation of Parallel K-means on the MapReduce model. K-means is one the most widely used data clustering algorithms and while, quite simplistic it gives consistent and valuable results. The current volume of data that need to be processed in the real industry, makes impossible processing them on single nodes. Hadoop offers a great platform for storage of terabytes, to petabytes of data on a distributed environment, and gives the ability of processing them in a parallel way on the machines they reside on. The second version of Hadoop introduced YARN, that extends the capabilities of Hadoop by scaling it on more than 4000 nodes and enabling the freedom of using other programming models beyond MapReduce.

We examined a fair amount of the aspects of the Hadoop framework, covering its Filesystem, the implementation of the MapReduce model and the YARN resource managers. We also investigated the parameters and the settings needed to setup a single Hadoop node and eventually a cluster.

While the K-means matches the map and reduce steps, lack of support of native iterations, adds big overheads on disk and network I/O and job initialization. It is also clear that K-means implementations on Hadoop are meaningful given a significant amount of data. Still our experimental setup was able to perform well enough considering the data size and available resources. A significant advantage is that the solution is portable and can run out of the box on any existing Hadoop cluster, and can scale as the number of data increases.

K-means shares the main principles of most machine learning algorithms, and in the effort of parallelizing it, we came across the common problems met in these scenarios. The need for scaling machine learning algorithms is extremely important nowadays, and this is clear by the effort and involvement put in developing more libraries and solutions, by many companies and organizations.

Hadoop is the most established tool for managing large clusters, and is under constant development. While many applications can be directly developed and run on top of Hadoop, it is obvious that Hadoop serves as the platform for running higher level applications that are offered for storing, processing and analyzing data on industry scale environments.

Bibliography

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 0387310738.

[2] Konstantin I. Boudnik. *Hadoop Genealogy: continued*. Sept. 2013. URL: http://drcos.boudnik.org/2013/09/hadoop-genealogy-continued.html.

[3] Abhinandan S. Das et al. "Google News Personalization: Scalable Online Collaborative Filtering". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. Banff, Alberta, Canada: ACM, 2007, pp. 271–280. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242610. URL: http://doi.acm.org/10.1145/1242572.1242610.

[4] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[5] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. "MapReduce for Data Intensive Scientific Analyses". In: *Proceedings of the 2008 Fourth IEEE International Conference on eScience*. ESCIENCE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 277–284. ISBN: 978-0-7695-3535-7. DOI: 10.1109/eScience.2008.59. URL: http://dx.doi.org/10.1109/eScience.2008.59.

[6] Apache Software Foundation. *Apache Hadoop 2.2.0*. July 2013. URL: https://hadoop.apache.org/docs/current2/.

[7] Apache Software Foundation. *Apache Hama*. Feb. 2014. URL: http://hama.apache.org/.

[8] Apache Software Foundation. *Apache Mahout*. Feb. 2014. URL: http://mahout.apache.org/.

[9] Apache Software Foundation. *Apache Spark*. Feb. 2014. URL: http://spark.incubator.apache.org/.

[10] Thilina Gunarathne et al. “Scalable Parallel Computing on Clouds Using Twister4Azure Iterative MapReduce”. In: *Future Gener. Comput. Syst.* 29.4 (June 2013), pp. 1035–1048. ISSN: 0167-739X. DOI: 10.1016/j.future.2012.05.027. URL: http://dx.doi.org/10.1016/j.future.2012.05.027.

[11] Hortonworks Inc. *Hadoop YARN*. Jan. 2014. URL: http://hortonworks.com/hadoop/yarn/.

[12] Yahoo! Inc. *Yahoo! Hadoop Tutorial*. Jan. 2014. URL: http://developer.yahoo.com/hadoop/tutorial/module1.html.

[13] Anil K. Jain. “Data clustering: 50 years beyond K-means”. In: *Pattern Recognition Letters* 31.8 (2010), pp. 651–666.

[14] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0-13-022278-X.

[15] Manasi N. Joshi. *Parallel K- Means Algorithm on Distributed Memory Multiprocessors Abstract*. 2003.

[16] Thomas Jungblut. *k-Means Clustering with MapReduce*. May 2011. URL: http://codingwiththomas.blogspot.de/2011/05/k-means-clustering-with-mapreduce.html.

[17] *MapReduce*. URL: http://en.wikipedia.org/wiki/MapReduce.

[18] Sujee Maniyam Mark Kerzner. *Hadoop Illuminated*. Hadoop illuminated LLC, 2013.

[19] *MLSP 2012 Competition: Amazon Data Science Competition*. Jan. 2012. URL: http://mlsp2012.conwiz.dk/index.php?id=43.

[20] Makho Ngazimbi. “Data Clustering using MapReduce”. MA thesis. Boise State University, 2009.

[21] Michael G. Noll. *Running Hadoop on Ubuntu Linux (Multi-Node Cluster)*. July 2011. URL: http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/.

[22] *Stratosphere*. Feb. 2014. URL: http://stratosphere.eu/.

[23] Tom White. *Hadoop: The Definitive Guide*. 3rd. O’Reilly Media, Inc., 2012. ISBN: 978-1-4493-1152-0.

[24] Charles Zedlewski. *An update on Apache Hadoop 1.0*. Jan. 2012. URL: http://blog.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/.

[25] Weizhong Zhao, Huifang Ma, and Qing He. “Parallel K-Means Clustering Based on MapReduce”. In: *Cloud Computing*. Ed. by MartinGilje Jaatun, Gansen Zhao, and Chunming Rong. Vol. 5931. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 674–679. ISBN: 978-3-642-10664-4. DOI: 10.1007/978-3-642-10665-1_71. URL: http://dx.doi.org/10.1007/978-3-642-10665-1_71.

Appendix A

Code Listings

Listing A.1: MapperClass

```
import ...

public class KMeansMapper extends
                        Mapper<Text, Vector, Text, Vector> {

    private final List<ClusterCenter> clusterCenters =
                             new ArrayList<ClusterCenter>();

    @Override
    protected void setup(Context context) throws IOException,
                                    InterruptedException {
        super.setup(context);
        Configuration conf = context.getConfiguration();
        Path centroids = new Path(conf.get("centroid.path"));

        SequenceFile.Reader reader =
        new SequenceFile.Reader(conf, Reader.file(centroids));

        Text key = new Text();
        Vector value = new Vector();
        while (reader.next(key, value)) {
            ClusterCenter clusterCenter = new ClusterCenter(
                                            new Text(key),
                                            new Vector(value)
```

```
                                                    );
            clusterCenters.add(clusterCenter);
        }
        reader.close();
    }

    @Override
    protected void map(Text key, Vector value, Context context)
                    throws IOException, InterruptedException {
        ClusterCenter nearest = clusterCenters.get(0);
        double nearestDistance = Double.MAX_VALUE;
        for (ClusterCenter clusterCenter : clusterCenters) {
            double dist =
                clusterCenter.measureDistance(value.getVector());
            if (nearestDistance > dist) {
                    nearest = clusterCenter;
                    nearestDistance = dist;
            }
        }

        context.write(nearest.getName(), value);
    }
}
```

Listing A.2: Reducer Class

```
public class KMeansReducer extends
                        Reducer<Text, Vector, Text, Vector> {

    private final List<ClusterCenter> clusterCenters =
                                    new ArrayList<ClusterCenter>();

    @Override
    protected void reduce(Text key, Iterable<Vector> values,
                                            Context context)
                throws IOException, InterruptedException {
        Vector newCenter = new Vector();
        List<Vector> vectorList = new ArrayList<Vector>();
        newCenter.setVector(new double[5]);
        for (Vector value : values) {
            vectorList.add(new Vector(value));
            for (int i = 0; i < value.getVector().length; i++) {
                newCenter.getVector()[i] += value.getVector()[i];
            }
        }

        for (int i = 0; i < newCenter.getVector().length; i++) {
            newCenter.getVector()[i] =
                    newCenter.getVector()[i] / vectorList.size();
        }

        ClusterCenter center =
                new ClusterCenter(new Text(key), newCenter);
        clusterCenters.add(center);
        for (Vector vector : vectorList) {
            context.write(key, vector);
        }
    }

    protected void cleanup(Context context) throws IOException,
                                        InterruptedException {
        super.cleanup(context);
        Configuration conf = context.getConfiguration();

        Path outPath = new Path(conf.get("centroid.path"));
        FileSystem fs = FileSystem.get(conf);
        fs.delete(outPath, true);

        SequenceFile.Writer out =
            SequenceFile.createWriter(conf, Writer.file(outPath),
                                      Writer.keyClass(Text.class),
                                      Writer.valueClass(Vector.class));
        for (ClusterCenter clusterCenter : clusterCenters) {
            out.append(clusterCenter.getName(),
                                        clusterCenter.getCenter());
        }
        out.close();
    }
}
```