Open University of Catalonia

What about Big Data?

BS Final Project

Sergio Cruz Gonzalez

Computer Science and Engineering student
by The Open University of Catalonia

2015

Advanced Computer Architecture Department

Open University of Catalonia

What about Big Data?

BS Final Project

Author: Sergio Cruz Gonzalez
Computer Science and Engineering student by The Open University of Catalonia

Advisor: Dr. Francesc Guim Bernat
Dr. Computer Science

2015

Title:
What about Big Data?

Author:
Sergio Cruz Gonzalez

BS Final Project Committee:

President:	PRESIDENT
Speaker:	VOCAL
Secretary:	SECRETARY

Agree to mark with the qualification:

Barcelona, 29 December 2015

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

1. Introduction

In today's ICT era, data is more voluminous and multifarious and it is being transferred at high speed. Some reasons for these trends are: scientific organizations are solving big problems related to high-performance computing workloads, different types of public services are emerging and being digitized as well as new types of resources are being used. Mobile devices, global positioning systems, financial transaction logs, social media, sensors, monitoring systems, earth observation or medical imaging are all sources of Big Data and therefore they are generating large sets of complex data.

Figure 1.1 shows the results of a Big Data recent survey exhibited by Talend [1]. The survey revealed that many common real-world applications deal with Big Data:

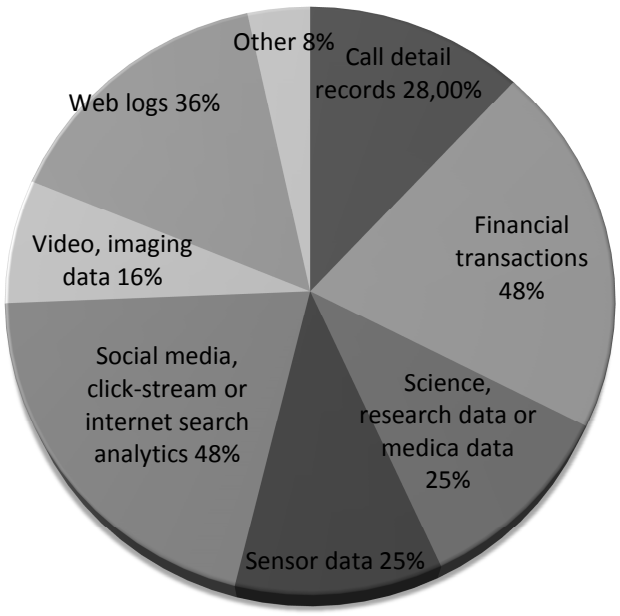


Figure 1.1 Big Data recent survey

Managing and mining such data to discover useful information is a significant challenge. All of these questions lead us to the Big Data concept, which tries to answer and solve all these issues. We can define Big data as huge and complex structured or unstructured data (among others types) that is difficult to manage using traditional technologies such as database management system (DBMS [2]) and software techniques. From my point of view and according to ICT experts, Big Data is the latest important trend along with Virtualization [3] and Cloud computing [4].

1.1 Project motivation

Such as we described above, Big Data is a broad term for data sets so large or complex that traditional data processing applications are inadequate; that is, it usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process data within a tolerable elapsed time.

Big Data characteristics are often described using a multi-V model. Gartner [5] proposed a 3V model of Big Data (Volume, Velocity and Variety), but an additional dimension, Veracity, is also important to achieve data reliability and accuracy:

- Volume: Volume is a major dimension of Big Data. Currently, the volume of data is increasing exponentially, from terabytes to petabytes and beyond.

- Velocity: Velocity includes the speed of data creation, capturing, aggregation, processing, and streaming. Different types of Big Data may need to be processed at different speeds. In this point, we have multiple types of solutions such as Batch Processing [6], Real-time Processing [7] and Streaming Processing [8] .

- Variety: Variety is one of the most important characteristics of Big Data. Many sources of Big Data generate many different forms of data. For example, if a new application is developed, a new type of data format may be introduced. It means that data mining and analysis techniques become more challenging. Thus, structured data, unstructured data, semi-structured data or mixed data are a few examples of Big Data forms.

- Veracity: The veracity of Big Data is the reliability, accuracy and understandability of the data. In some Big Data applications, controlling data quality and data precision has demonstrated to be a big challenge.

Our motivation in this BS Final Project will be analyzing the different Big Data processing approaches and techniques. In other words, we will focus on the Velocity concept, specifically in the different solutions based on Batch Processing techniques. The study will focus on the two major current proposed solutions: Apache Hadoop [9] and Apache Spark [10]. As we will see, both are open source Big Data processing frameworks, but each one uses the Batch Processing solution in a different manner.

1.2 Goals and objectives

The main goal of this BS Final Project is to compare both Apache Hadoop and Apache Spark solutions through different perspectives such as:

- Analyzing and understanding both Apache Hadoop and Apache Spark Big Data processing frameworks.
- Designing and building Big Data applications using the standard programming models (paradigms) proposed for both solutions (MapReduce [11] and Resilient Distributed DataSets [12] respectively).
- Using benchmarking and stress testing, discuss and conclude the strengths and weaknesses of both solutions.

An important point that we need to keep in mind and deserves further consideration is the use of Cloudera (Cloudera's open-source Apache Hadoop distribution, CDH) [13] as infrastructure solution. Cloudera, also known as *"The Platform for Big Data"*, it will allow us to have a free Hadoop architecture/environment (including Apache Spark) in a few steps thanks to one of its key components: Cloudera Manager [14].

1.3 Requirements

To achieve the goals and objectives described above, we will need a capable environment to support the Cloudera's solution. Due to Cloudera requirements are very high, we will dispose of a full environment in the cloud, which is completely reachable through Internet (only one node must be reachable). A brief description of the hardware environment is shown below:

- Four physical nodes interconnected through private LAN.
- Private network addressing (excluding one node, which will dispose of both private and public network addressing).
- Default Operating System: GNU/Linux CentOS 6.6 [15].
- CPU Model: Intel® Xeon® Processor L5410.
- CPU (MHz): 2333.
- CPU Cores: 8.
- Memory (MB): 19987.
- Disk (GB): 48.

1.4 Work breakdown

The work breakdown is shown below:

- Phase 0: Work breadkdown (16/09/2015–22/10/2015, 37 days)
  - Information collection
  - Information analysis
  - Creating a work breakdown structure (BS Final Project – PAC 1)

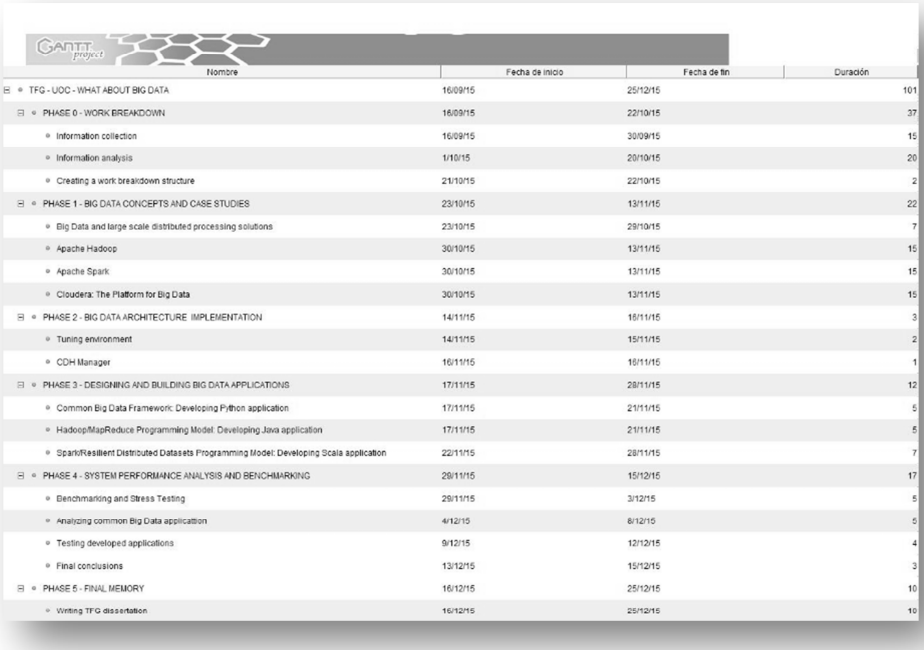
- Phase 1: Big Data concepts and case studies (23/10/2015–13/11/2015, 22 days)
  - Big Data and large scale distributed processing solutions
  - Apache Hadoop
  - Apache Spark
  - Cloudera: The Platform for Big Data

- Phase 2: Big Data architecture implementation (14/11/2015–16/11/2015, 3 days)
  - Tuning environment
  - CDH Manager

- Phase 3: Designing and building Big Data applications (17/11/2015–28/11/2015, 12 days)
  - Common Big Data Framework: Developing Python application
  - Hadoop/MapReduce Programming Model: Developing Java application
  - Spark/Resilient Distributed Datasets Programming Model: Developing Scala application

- Phase 4: System performance analysis and benchmarking (29/11/2015–15/12/2015, 17 days)
  - Benchmarking and Stress Testing
  - Analyzing common Big Data application
  - Testing developed applications
  - Final conclusions

- Phase 5: Final memory (16/12/2015–25/12/2015, 10 days)
  - Writing TFG dissertation



Nombre	Fecha de inicio	Fecha de fin	Duración
TFG - UOC - WHAT ABOUT BIG DATA	16/09/15	25/12/15	101
PHASE 0 - WORK BREAKDOWN	16/09/15	22/10/15	37
Information collection	16/09/15	30/09/15	15
Information analysis	1/10/15	20/10/15	20
Creating a work breakdown structure	21/10/15	22/10/15	2
PHASE 1 - BIG DATA CONCEPTS AND CASE STUDIES	23/10/15	13/11/15	22
Big Data and large scale distributed processing solutions	23/10/15	29/10/15	7
Apache Hadoop	30/10/15	13/11/15	15
Apache Spark	30/10/15	13/11/15	15
Cloudera: The Platform for Big Data	30/10/15	13/11/15	15
PHASE 2 - BIG DATA ARCHITECTURE IMPLEMENTATION	14/11/15	16/11/15	3
Tuning environment	14/11/15	15/11/15	2
CDH Manager	16/11/15	16/11/15	1
PHASE 3 - DESIGNING AND BUILDING BIG DATA APPLICATIONS	17/11/15	28/11/15	12
Common Big Data Framework: Developing Python application	17/11/15	21/11/15	5
Hadoop/MapReduce Programming Model: Developing Java application	17/11/15	21/11/15	5
Spark/Resilient Distributed Datasets Programming Model: Developing Scala application	22/11/15	28/11/15	7
PHASE 4 - SYSTEM PERFORMANCE ANALYSIS AND BENCHMARKING	29/11/15	15/12/15	17
Benchmarking and Stress Testing	29/11/15	3/12/15	5
Analyzing common Big Data application	4/12/15	8/12/15	5
Testing developed applications	9/12/15	12/12/15	4
Final conclusions	13/12/15	15/12/15	3
PHASE 5 - FINAL MEMORY	16/12/15	25/12/15	10
Writing TFG dissertation	16/12/15	25/12/15	10

Figure 1.2 Gantt chart

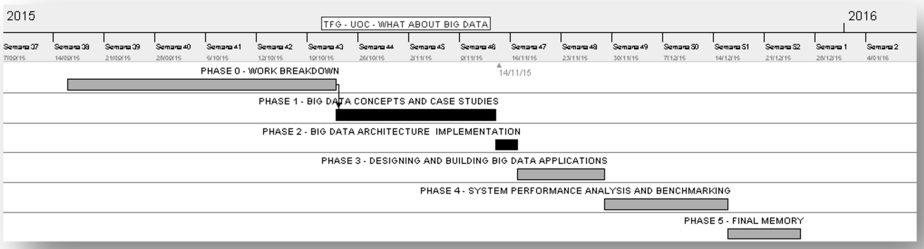


Figure 1.3 Simplified Gantt chart

1.5 Organization of the BS Final Project

The rest of this BS Final Project is organized as follows:

- Chapter 2 presents key Big Data concepts. We describe challenges and opportunities, lifecycle management as well as characteristics (Gartner's model). Furthermore, we aim to provide a general overview of the current Big Data processing models (Batch, Real-time and Stream Processing).

- Chapter 3 describes our case studies. We discuss about the two major current proposed solutions: Apache Hadoop and Apache Spark. The main goal is to dissect both solutions and try to understand their behaviors through different perspectives, such as their architectonic solutions or their default logical roles at high level.

- Chapter 4 presents our Big Data architecture implementation. In the first part, we show both low-level logical architecture (hardware tier, specifications) and high-level logical architecture (software tier, associated roles) as an architecture solution. In the second part, we perform a full installation of a platform for Big Data; the procedure includes installing and tuning a GNU/Linux environment as well as the use of Cloudera (Cloudera's open-source Apache Hadoop distribution, CDH) as infrastructure solution.

- Chapter 5 describes the methodology that we have followed in order to perform system performance analysis and benchmarking. The main goal is to perform through some benchmarks available, system performance analysis and benchmarking of both large scale distributed processing solutions (Haddop and Spark). In this point, we present three different applications, where each one of them perform different tasks and follow a different programming paradigm. Additionally, we show different performance metrics as well as statistical data.

- Chapter 6 presents our final conclusions and possible future studies

Chapter 2

2 Key Big Data concepts

In this chapter we present key Big Data concepts. At first, we describe challenges and opportunities arising from Big Data term. Next, we dissect the Big Data lifecycle management from two different perspectives and describe the Big Data characteristics (Gartner's model). The last subchapter aims to provide a general overview of the current Big Data processing models.

2.1 Challenges and opportunities

Big Data is a huge structured or unstructured data set that is difficult to compute using traditional technologies such as database management system (DBMS) and software techniques. An increasing number of organizations are producing huge data sets, the size of which start at a few terabytes. Some examples and evidences are:

- Sloan Digital Sky Survey (SDSD) [16]: SDSS is a major multi-filter imaging and spectroscopic redshift survey using a dedicated 2.5-m wide-angle optical telescope at Apache Point Observatory in New Mexico. When the Sloan Digital Sky Survey started work in 2000, its telescope in collected more data in its first few weeks than had been amassed in the entire history of astronomy. Now, a decade later, its archive contains 140 terabytes of information. A successor, the Large Synoptic Survey Telescope [17], due to come on stream in Chile in 2016, will acquire that quantity of data every five days.

- WaltMart [18]: WaltMart is an U.S. retail giant that handles more than 1m customer transactions every hour, feeding databases estimated at more than 2.5 petabytes of information (the equivalent of 167 times the books in America's Library of Congress).

- Facebook [19]: Facebook, the most famous online social networking service, it is capable to store more than 40 million photos.

- Genome Research [20]: Decoding the human genome involves analyzing around 3 billion base pairs. The procedure took ten years the first time that it was done (2003), but it can now be achieved in one week.

All these examples and evidences lead us to the same idea: the real world contains an unimaginably huge amount of digital information which is increasing exponentially, from terabytes to petabytes and beyond.

This fact brings us new benefits and opportunities, but it also produces a great number of problems and challenges. In other words, well managed data set can be very useful; they can be used to unlock new information, obtain new sources of value and provide new ideas into science for example. However, managing and mining such data to discover useful information is a significant challenge. For example, all these newly generated data exceed the current available storage space; moreover, ensuring data security and protecting privacy is becoming harder due to the information is growing rapidly and is being shared widely around the world.

All of these questions lead us to a new term: Big Data.

2.2 Life cycle management

To better understand Big Data characteristics (see *The 4 V's of Big* Data), it is necessary to describe the Big Data life cycle management. Generally, Big Data systems use the following life cycle to manage its Big Data:

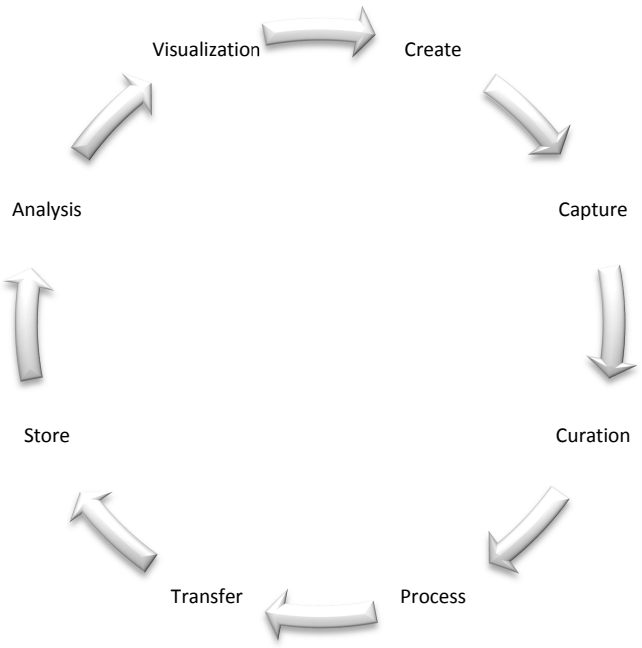


Figure 2.1 Genuine Big Data Life-Cycle Management

However, we can still simplify the above illustration and divide it into four major phases:

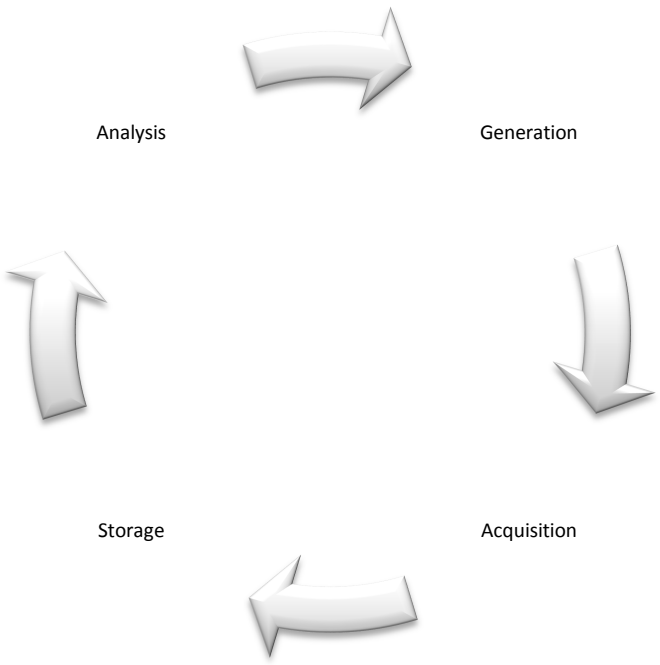


Figure 2.2 Simplified Big Data Life-Cycle Management

In this way, we can analyze the above phases as follows:

1. **Big Data Generation**: The first phase of the Big Data life cycle involves creating of Big Data. Data sources generate a huge amount of data. These types of data sources are anything like social networking sites, mobile devices or sensors and they can be categorized as enterprise data, healthcare or Internet of Things (IoT) [21] among many other things.

2. **Big Data Acquisition**: The second phase of the Big Data life cycle involves capturing, curating, processing and transmission of Big Data. In this phase, raw data generated by different data sources is captured, processed and transmitted to the next stage of the Big Data life cycle. Some examples of techniques for acquiring Big Data could be log files, sensors or sensing devices. Processing and transmission are mechanisms that deserve further consideration. Both of them aim to solve some issues in this critical phase like redundant and useless data. In this way, new useful data can save storage space and improve overall computing efficiency for Big Data processing.

3. **Big Data Storage**: The third phase of the Big Data life cycle involves the storage procedure. Specifically, this phase is responsible of data availability and reliability for Big Data analytics. Distributed file system (DFS) [22] is commonly used to store Big Data originating from large scale sources, distributed systems or data-intensive applications for example.

   Some current examples are GFS [23], HDFS [24], TFS [25] and NoSQL [26] databases (they are commonly used for Big Data storage and management).

4. **Big Data Analysis**: Big-data analysis is the last stage of the Big Data life cycle and includes Big Data analysis approaches and techniques as well as visualizing final data. Big-data analysis is similar to traditional data analysis in that potentially useful data is extracted and analyzed to maximize the value of the data. Approaches to Big Data analysis include mathematical approaches (which are used in many fields like engineering, healthcare or biology) and data mining approaches (which are used in many fields like regression analysis, clustering analysis or machine learning) among others.

2.3 The 4 V's of Big Data

Big Data characteristics are often described using a multi-V model. Gartner proposed a 3V model of Big Data (volume, velocity and variety), but an additional dimension, veracity, is also important to achieve data reliability and accuracy:

Figure 2.3 The 4 V's of Big Data

The 4 V's of Big Data can be described as follows:

- **Volume**: Volume is a major dimension of Big Data. Currently, the volume of data is increasing exponentially, from terabytes to petabytes and beyond.

- **Velocity**: Velocity includes the speed of data creation, capturing, aggregation, processing, and streaming. Different types of Big Data may need to be processed at different speeds. Velocity can be categorized as:

    - Batch Processing: Data arrives and is processed at certain intervals. Many Big Data applications process data in batches and have batch velocity.

    - Near-time Processing: The time between data arrival and its processing is very small, close to real time.

    - Real-time Processing: Data arrives and is processed in a continuous manner, which enables real time analysis.

    - Streaming Processing: Similar to real-time, data arrives and is processed upon incoming data flows.

- **Variety**: Variety is one of the most important characteristics of Big Data. Many sources of Big Data generate many different forms of data. For example, if a new application is developed, a new type of data format may be introduced. It means that data mining and analysis techniques become more challenging. Variety can be categorized as:

    - Structured data: In this form, data is very easy to input and analyze because there are many database management system (DBMS) tools that can store, query, and manage the data efficiently. A few examples of this type of data are characters, numbers or floating points.

    - Unstructured data: In this form, data cannot be stored and managed using database management system (DBMS) tools because of data is not in a table (according to a relational model). A few examples of this type of data are location information, sensors data or biological data. According to latest surveys and studies, social media websites and sensors are major sources of this type of data and eighty to ninety percent of today's data in the world is unstructured social media data. An additional important point: HP Labs has estimated that by 2030 approximately 1 trillion sensors will be in use, monitoring phenomena such as energy consumption, cyberspace, and weather.

    - Semi-structured data: In this form, data cannot be stored and managed using database management system (DBMS). This type of data is a type of structured data that is not organized in a table (according to a relational model).

- Mixed data: In this form, data may be a mixture of the above types of data. Mixed data requires complex data capture and processing.

- **Veracity**: The veracity of Big Data is the reliability, accuracy and understandability of the data. In some Big Data applications, controlling data quality and data precision has demonstrated to be a big challenge.

2.4 Processing models

As we described above (see *The 4 V's of Big Data*), due to Big Data may need to be processed at different speeds, in general terms velocity can be categorized as Batch Processing, Near-time Processing, Real-time Processing and Streaming Processing.

However, if we focus on processing models, that is, large scale distributed processing solutions, we can group processing technologies into three major categories: Batch Processing, Real-time Processing and Stream Processing.

2.4.1 Batch processing

Batch processing is used to process data in batches (jobs can take from minutes to hours lag). It means that data input is read, it is processed and it is written to the output. Apache Hadoop is the most well-known and popular open source implementation of batch processing as well as the most well-known and popular open source implementation of MapReduce programming model.

MapReduce is designed for batch processing of large volumes of data, and it is not suitable for recent demands like real-time processing or stream processing. In other words, it is appropriate for batch processing of Big Data that may take several hours or even days, and inadequate for jobs and queries that should finish in seconds or at most, minutes.

Basically, MapReduce defines computation as two functions: map and reduce. The input is a set of key/value pairs, and the output is a list of key/value pairs. The map function takes an input pair and extracts a set of intermediate key/value pairs. The reduce function takes an intermediate key and a list of intermediate values associated to that key as its input, and results a set of final key/value pairs as the output. Execution of a MapReduce program involves two phases. In the first phase each input pair is given to a map function and a set of input pairs is produced. Then, in the second phase, all of the intermediate values that have the same key are aggregated into a list, and each intermediate key and its associated intermediate value list is given to a reduce function.

The execution of a MapReduce program follows the same two-phase procedure. Usually, distributed MapReduce is implemented using master/slave architecture. The

master machine is responsible of assignment of tasks and manages the slave machines. The input is stored over a shared storage (like a distributed file system), and is split into chunks. First, a copy of map and reduce functions code is sent to all workers. Then, master assigns map and reduce tasks to workers. Each worker assigned a map task, reads the corresponding input split and passes all of its pairs to map function and writes the results of the map function into intermediate files. After the map phase is finished, the reducer workers read intermediate files and pass the intermediate pairs to reduce function and finally the pairs resulted by reduce tasks are written to final output files. The overall execution of a MapReduce program is given in Figure 2.4:

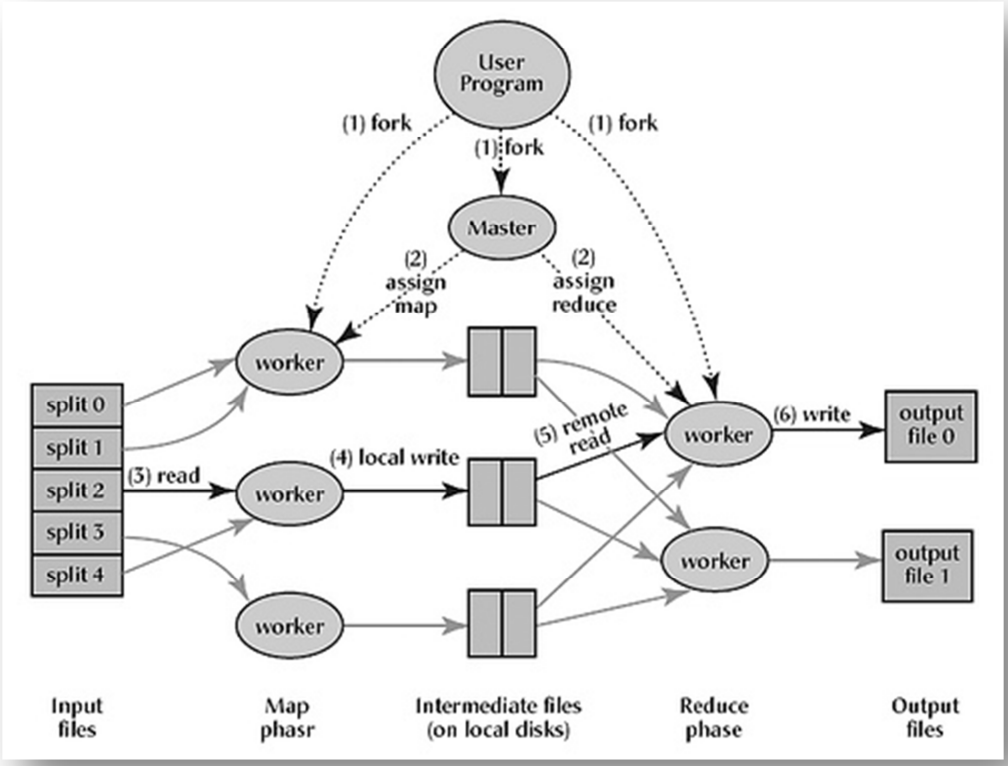


Figure 2.4 Batch processing key concept

The advantages of the MapReduce programming model include easy programing model, near-linear speed up, good scalability as well as fault tolerance. The major disadvantage of this processing model is that it is unable to execute iterative or recursive jobs. Besides, the standard batch processing behavior is that all inputs must be ready by map before the reduce job starts, which makes this processing model unsuitable for real-time processing or stream processing.

2.4.2 Real-time processing

Real-time processing is used to process data and get the results almost immediately (within seconds lag). Solutions in this category can be classified into two major groups: solutions that try to reduce overhead of MapReduce programming model and make it faster to enable execution of jobs in less than seconds (they are also known as improved or fast Batch processing) or solutions that focus on providing means for real-time queries over structured and unstructured Big Data using new optimized approaches and techniques.

Here, we describe two different approaches of these solutions respectively:

- **In-memory computing**: In-memory computing is based on using a distributed main memory system to store and process Big data in real-time. Main memory provides higher bandwidth, more than 10 gigabytes per second compared to hard disk's 200 megabytes per second. Access latency is also much better, nanoseconds versus milliseconds for hard disks. Price of RAM is also affordable. Currently, 1 TB of RAM can be bought with less than 20,000$. These performance superiority combined with dropping price of RAM makes in-memory computing a great alternative to disk-based Big Data processing. Apache Spark is the most well-known and popular open source implementation of this type of solution. We will discuss main characteristics and capabilities of Apache Spark in next chapters.

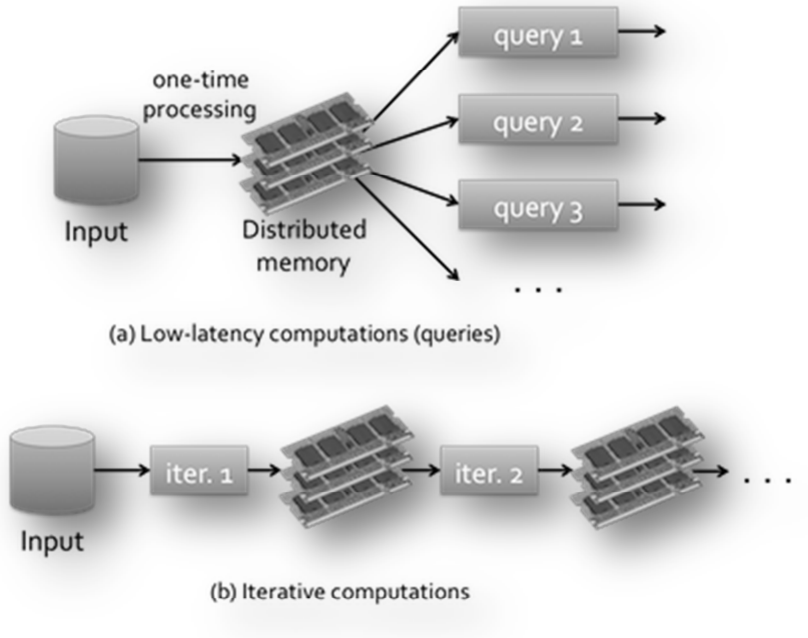


Figure 2.5 In-memory computing key concept

- **Real-time queries**: Real-time queries over Big Data was first implemented in Dremel by Google [27]. It uses a novel columnar storage format for nested structures with fast index and scalable aggregation algorithms for computing query results in parallel instead of batch sequences. These two techniques enable Dremel to process complex queries in real-time. There are several implementations in this field such as Cloudera Impala (an open source implementation of Dremel) [28], Apache Drill [29], Shark (SQL on Spark) [30] or the Stinger project by Hortonworks [31], which is an effort to make 100x performance improvement and add SQL semantics to future versions of Apache Hive [32].

2.4.3 Stream processing

Also known as event-stream processing, it is used to continuously process and handle on the live stream data to get a result (a stream of data is processed at real time in parallel). Log streams, message streams or even event streams are good examples of data streams. The two major implementations of this processing model are Apache Storm [33] from Twitter and Apache S4 [34] from Yahoo.

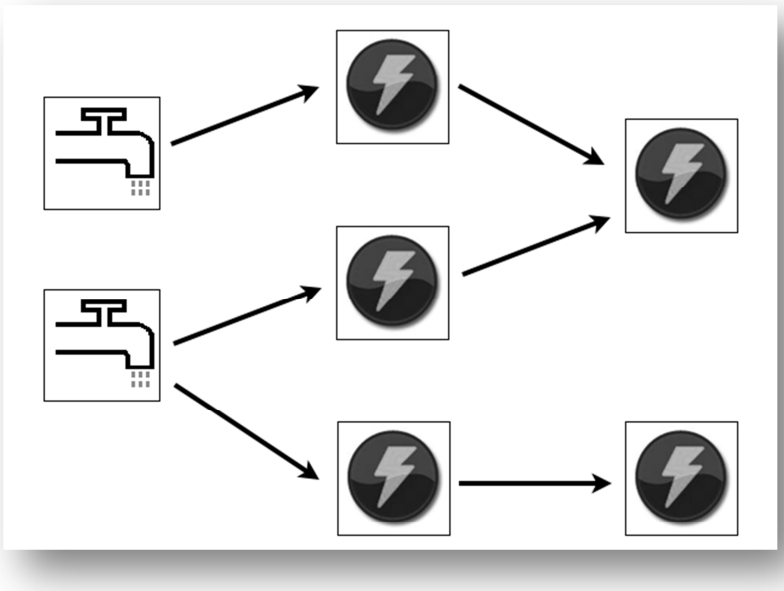


Figure 2.6 Apache Storm topology: Spout & Bolt

Furthermore, we can talk about hybrid computation model (a technique known as micro-batching [35]). Micro-batching is a mix between batch processing and stream processing (a special case of batch processing where batch sizes are very small). The main idea is to treat the stream as a sequence of small batch chunks of data. On small intervals, the incoming stream is packed to a chunk of data and is delivered to batch system to be processed. Despite of its originality, this technique incurs a cost of latency; that is to say, it might not be suitable for certain applications. Spark Streaming [36] is the most well-known solution.

Due to current demands and applications requirements, we need to keep in mind that stream processing model will significantly grow in the future.

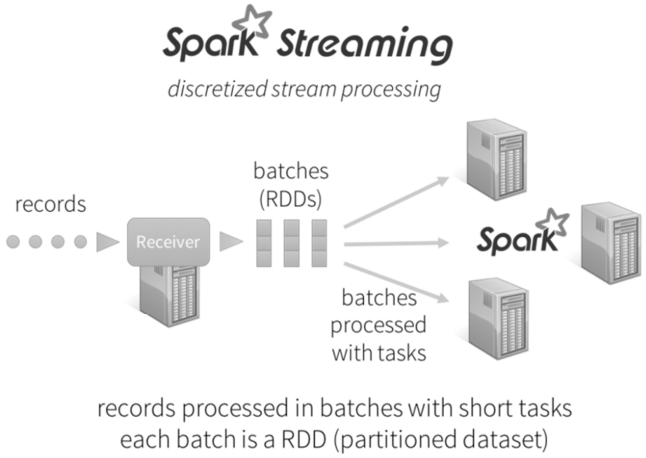


Figure 2.7 Spark Streaming Architecture

3 Case studies

In this chapter we present our case studies. We discuss about the two major current proposed solutions: Apache Hadoop and Apache Spark. Both solutions are based on the Batch processing model (although Apache Spark is really in-memory computation solution, it is a Batch processing system at heart too). The main goal is to dissect both solutions and try to understand their behaviors through different perspectives, such as their architectonic solutions or their default logical roles at high level.

3.1 Apache Hadoop

Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be susceptible to failures.

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

- **Hadoop Common**: The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN** [37]: A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce:** A YARN-based system for parallel processing of large data sets.

The term "Hadoop" is not just the main project (the base modules shown previously), rather it is an ecosystem or collection of additional software packages that can be installed on top of or alongside Hadoop, such as Ambari [38], Avro [39], Cassandra [40], Chukwa [41], HBase [42], Hive [32], Mahout [43], Pig [44], Spark [10], Tez [45] and Zookeeper [46] among others.

The Hadoop framework is implemented in Java programming language (with some native code in C and command line utilities written as Shell script). For end-users, although Java is the standard option to implement MapReduce applications, any programming language can be used with "Hadoop Streaming" [47] or “Hadoop Pipes” [48] to implement the "map" and "reduce" functions of the user's program.

It is important to keep in mind that Apache Hadoop's MapReduce and HDFS core components were inspired by Google papers on their MapReduce [49] and Google File System [23].

3.1.1 Hadoop architecture

To better understand the core behavior of Apache Hadoop, it is essential to understand its base architecture (tiered architecture), which has suffered a few changes over the past years. Thus, we going to describe the last two architectures: Hadoop MapReduce 1.0 (MRv1) and Hadoop MapReduce 2.0 (MRv2 or YARN) respectively. In this point, it is important to highlight that last implemented architecture (Hadoop 2.0) is the base architecture for our case studies (Apache Hadoop and Apache Spark).

As shown below, basically in Hadoop 2.0 a new layer has been introduced between HDFS and MapReduce. This is YARN (**Y**et **A**nother **R**esource **N**egotiator) framework which is responsible for doing Cluster Resource Management (by resources we mean CPU, memory, etc.). The main purpose of this division is to release cluster management from MapReduce engine (in other words, MapReduce will only perform data processing), so YARN can take over the task of cluster management.

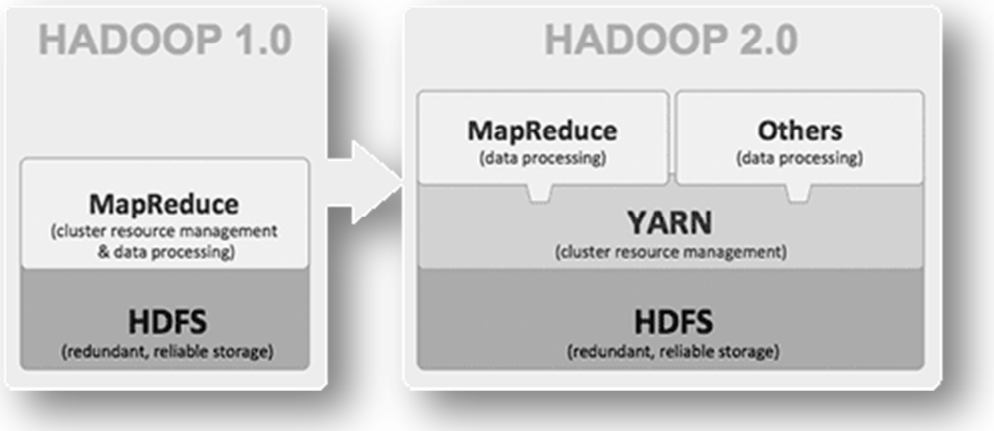


Figure 3.1 Hadoop 1.0 to Hadoop 2.0 architecture

In Hadoop 1.0, there is a strong coupling between cluster resource management and MapReduce engine. If we focus on cluster resource management, it is composed of the JobTracker process, which is the master node, and the per-node slaves called TaskTrackers processes. The cluster resource management architecture in Hadoop 1.0 is illustrated below:

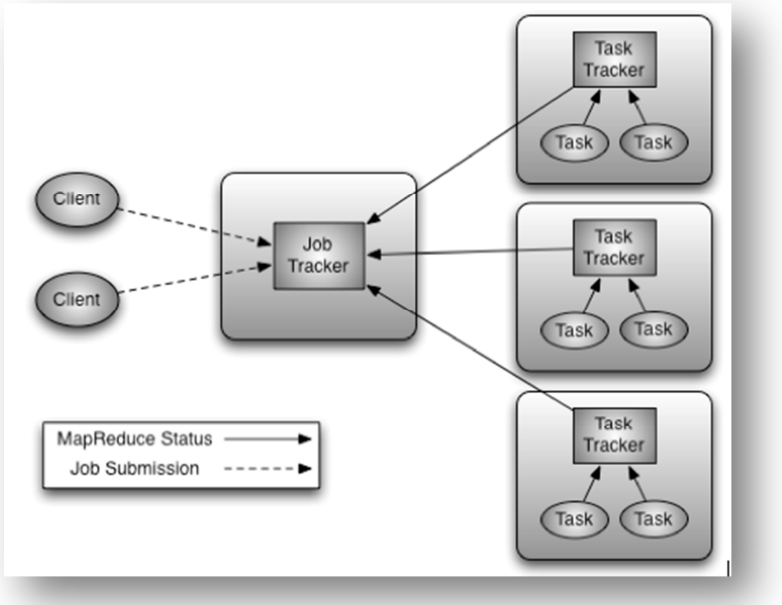


Figure 3.2 Hadoop 1.0 architecture

The JobTracker master process is the central scheduler for all MapReduce jobs in the cluster. Similar to most resource managers, the JobTracker process has two pluggable scheduler modules: Capacity and Fair. The JobTracker process is responsible for managing the TaskTrackers on worker server nodes, tracking resource consumption and availability, scheduling individual job tasks, tracking progress, and providing fault tolerance for tasks.

The TaskTracker process manages tasks on the individual nodes. The TaskTracker process communicates with JobTracker process; that is, it is controlled and directed by the JobTracker process. Besides, it is responsible to launch and remove jobs as well as providing task status information to the JobTracker process. The TaskTracker process also communicates through heartbeats to the JobTracker process; if the JobTracker process does not receive a heartbeat from a TaskTracker process, it assumes that it has failed and takes appropriate action (for example restarts jobs).

Hadoop 1.0 architecture represents many issues like:

- Scalability: JobTracker runs on single machine doing several tasks (resource management, scheduling, monitoring, etc.).
- Availability: JobTracker is a single point of failure (if JobTracker fails, all jobs must restart). Besides, there is only one JobTacker per cluster (limit of about 4000-5000 nodes per cluster).

- Resource utilization: inflexible "slots" configured on nodes (map or reduce, not both).

- Limitation with running MapReduce applications: force everything needs to look like MapReduce (lack support for alternate paradigms).

As we will see below (see *Hadoop YARN (Yet Another Resource Negotiator)*) Hadoop 2.0 architecture solves all these problems with YARN.

3.1.2 Hadoop HDFS (Hadoop Distributed File System)

The Hadoop Distributed File System (HDFS) is the primary distributed storage used by Hadoop applications. HDFS is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. For example, HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets.

Some key features of HDFS are:

- **Highly fault-tolerant**: HDFS may consist of hundreds or thousands of server machines. HDFS assumes nodes will fail, so it achieves reliability by replicating data across multiple nodes (it divides files into blocks of 128 MB by default and distributes 3 copies randomly across the cluster).

- **High throughput**: In HDFS, when we want to perform a task, the work is divided and shared among different nodes; it means that all the nodes will be executing the tasks assigned to them independently and in parallel. So the work will be completed in a very short period of time. For example, by reading data in parallel, we decrease the actual time to read data. In this way, HDFS gives high throughput.

- **Suitable for applications with large data sets**: Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size (HDFS is designed to support large files). This fact should provide high data bandwidth and scale to hundreds of nodes in a single cluster; besides, it should support tens of millions of files in a single instance.

- **Streaming access to file system data**: Applications that run on HDFS need streaming access to their data sets. HDFS is designed for Batch processing rather than interactive use by users.

- **Built out of commodity hardware**: Commodity hardware is generally a non-expensive system (which is not of high quality or high-availability for example). Thus, HDFS can be installed in any average commodity hardware without any problem.

Typically, HDFS has master/slave cluster architecture and consists of a NameNode (which manages the file system metadata and regulates access to files by clients) and DataNodes (which store actual data, handle read and write requests and perform internal block operations). Currently, is typical to find a Backup Node (Secondary Namenode). By its name, it gives a sense that it is a backup for the Namenode, but in reality it is not. Secondary Namenode takes responsibility of merging editlogs (sequence of changes made to the filesystem after Namenode started) with fsimage (snapshot of the filesystem when namenode started) from the namenode. In other words, its whole purpose is to have an HDFS checkpoint.

The HDFS architecture is illustrated below:

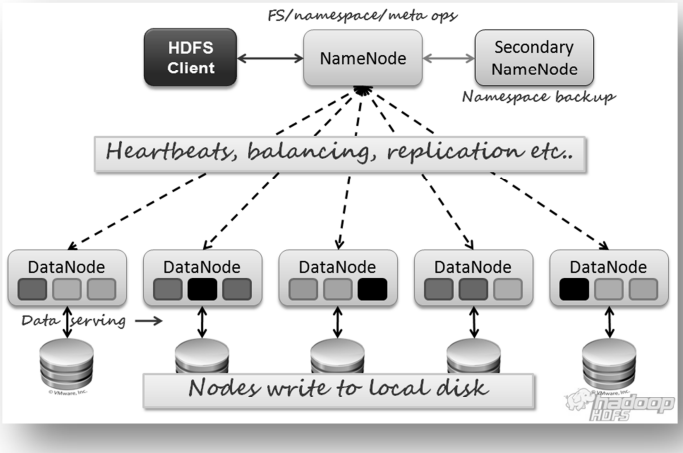


Figure 3.3 HDFS architecture

Keeping in mind above information, it is also interesting to understand about how data flow happens between clients and HDFS system.

In the figure below, general steps are described in a reading operation from HDFS. We suppose that a client (HDFS client) wants to read a file from HDFS:

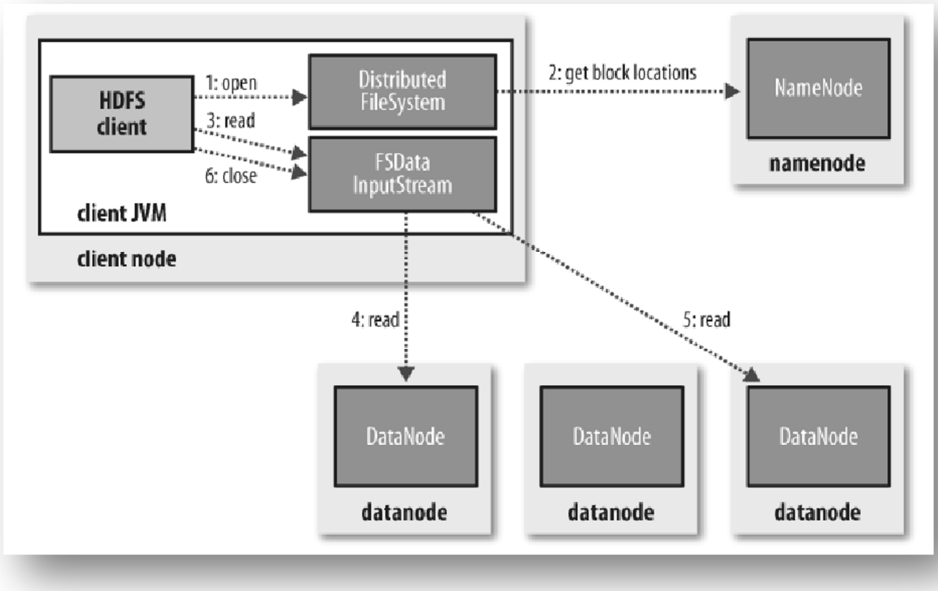


Figure 3.4 HDFS read operation

- **Step 1**: The client opens the file that it wishes to read by calling "open()" operation on the FileSystem object (which for HDFS is an instance of DistributedFileSystem class).

- **Step 2**: DistributedFileSystem calls the Namenode using RPC (Remote Procedure Call protocol) to determine the locations of the blocks for the first few blocks in the file. Besides:

    - For each block, the Namenode returns the addresses of the Datanodes that have a copy of that block.
    - The Datanodes are sorted according to their proximity to the client.
    - The DistributedFileSystem returns a FSDataInputStream to the client for it to read data from.

- **Step 3**: The client calls “read()” operation on the stream. Also, DFSInputStream connects to the first (closest) Datanode for the first block in the file.

- **Step 4**: Data is transmitted from the Datanode to the client.

- **Step 5**: When the end of the block is reached, DFSInputStream will close the connection to the Datanode, then it will find the best Datanode for the next block.

- **Step 6**: When the client has finished all read tasks, it calls “close()” operation on the FSDataInputStream.

Other things to consider are as follows:

- During reading operation, if the client encounters an error while communicating with a Datanode, then it will try the next closest Datanode for that block.

- Client remembers Datanodes that have failed, so it avoids unnecessary retries over them for later blocks.

- Client also verifies checksums for the data transferred to it from the Datanode, so if a corrupted block is found, it is reported to the Namenode.

In the same way, we show general steps involved in a writing operation from HDFS. We suppose that a client (HDFS client) wants to write a file from HDFS.

Figure below represents writing operation:

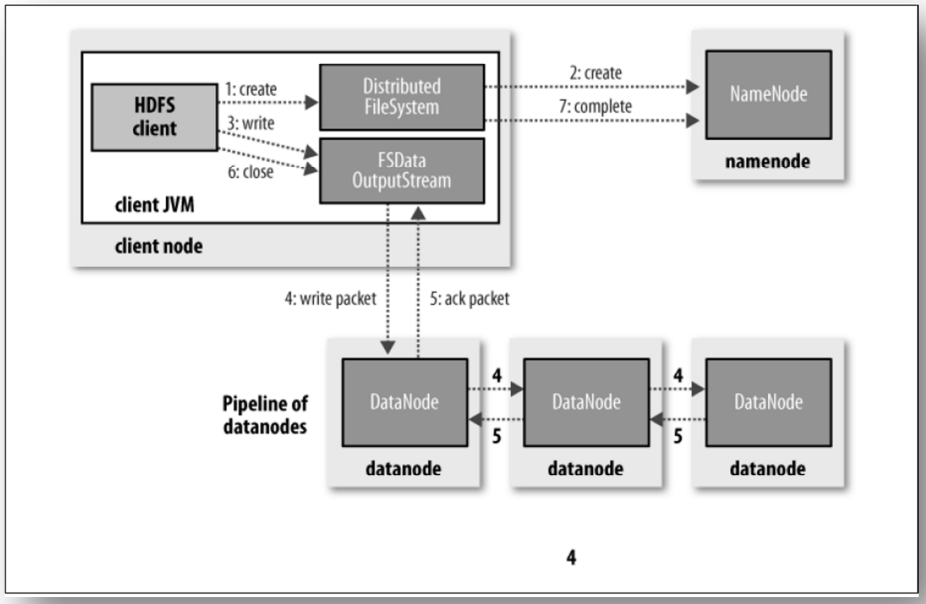


Figure 3.5 HDFS write operation

- **Step 1**: The client creates the file by calling "create()" operation (which is a method on DistributedFileSystem class).

- **Step 2**: DistributedFileSystem makes an RPC call to the Namenode to create a new file in the filesystem's namespace, with no blocks associated with it. Here, The Namenode performs various checks to make sure that the file doesn't already exist and that the client has the right permissions to create the file. If checks pass, the Namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException. Finally, the DistributedFileSystem returns a FSDataOutputStream for the client to start writing data to.

- **Step 3**: As the client writes data, DFSOutputStream splits it into packets, which it writes to an internal queue, called the "data queue". The "data queue" is consumed by the DataStreamer, whose responsibility is to ask the Namenode to allocate new blocks by choosing a list of suitable Datanodes to store the replicas. The list of Datanodes forms a pipeline.

- **Step 4**: The DataStreamer streams the packets to the first Datanode in the pipeline, which stores the packet and forwards it to the second Datanode in the pipeline. Similarly, the second Datanode stores the packet and forwards it to the third (and last) Datanode in the pipe line.

In this point, we assume that the replication level is three, so there are only three nodes in the pipeline.

- **Step 5:** DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by Datanodes (the “ack queue”). A packet is removed from the ack queue only when it has been acknowledged by all the Datanodes in the pipeline.

- **Step 6:** When the client has finished writing data it calls “close()” operation on the stream.

- **Step 7:** Above step flushes all the remaining packets to the Datanode pipeline and waits for acknowledgments before contacting the Namenode to indicate that the file is complete.

Other things to consider are as follows:

- If a Datanode fails while data is being written to it, then:

    - First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue.

    - The current block on the “good” Datanodes is given a new “identity” by the Namenode, so that the partial block on the failed Datanode will be deleted if the failed Datanode recovers later.

    - The failed Datanode is removed from the pipeline and the rest of the block’s data is written to the two “good” Datanodes in the pipeline.

    - The Namenode notices that the block is under-replicated, and it arranges for an additional replica to be created on another node.

3.1.3 Hadoop YARN (Yet Another Resource Negotiator)

As we discussed above (see *Hadoop architecture*) Hadoop 1.0 architecture suffers many problems. Thanks to YARN (**Y**et **A**nother **R**esource **N**egotiator), MapReduce engine only performs data processing, so YARN can take over the task of cluster management.

Next figure illustrates the main idea of Hadoop 2.0 architecture:

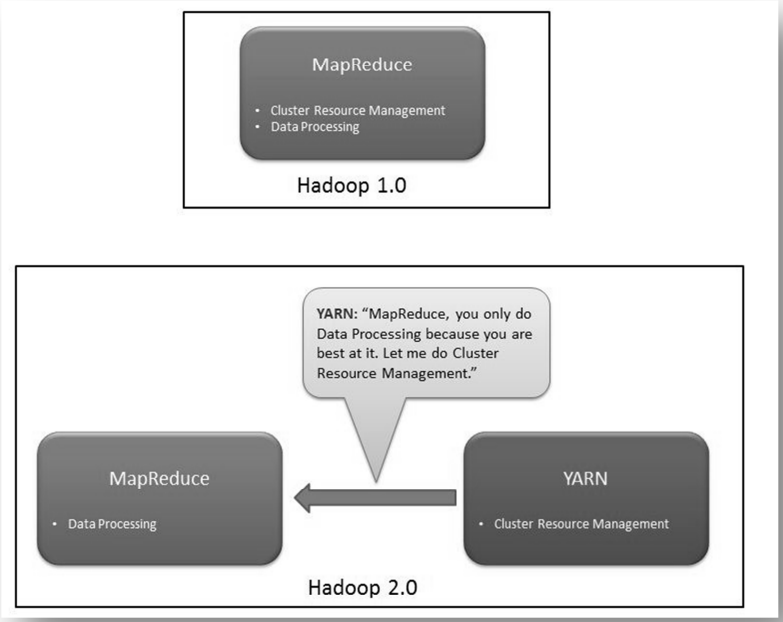


Figure 3.6 Hadoop 2.0 key concept

YARN has central resource manager component which manages resources and allocates the resources to the application. Multiple applications can run on Hadoop via YARN and all application could share common resource management.

The cluster resource management architecture in Hadoop 2.0 is illustrated below:

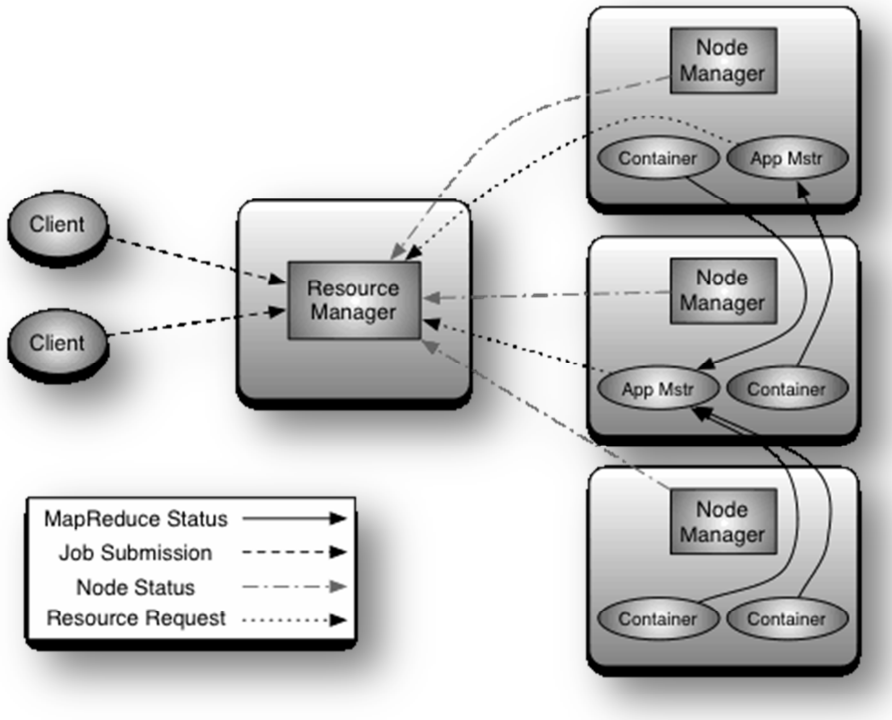


Figure 3.7 YARN architecture

In YARN based architecture, the JobTracker is split into two different daemons called ResourceManager (which runs on master node) and NodeManager (which runs on slaves nodes) respectively. Also there are two new components: an ApplicationMaster (per-application specific framework) and Containers (amount of resources such as CPU or memory).

The ResourceManager is a pure scheduler; that is, its sole purpose is to manage available resources among multiple applications on the cluster. As with Hadoop 1.0 architecture, both Fair and Capacity scheduling options are available.

The NodeManager is the per-machine framework agent that is responsible for Containers, monitoring their resource usage (CPU, memory, disk, network, etc.) and reporting back to the ResourceManager.

The ApplicationMaster is responsible for accepting job submissions, negotiating resource Containers from the ResourceManager and tracking progress of jobs. ApplicationMasters are specific to and written for each type of application. The ApplicationMaster also provides the service for restarting the ApplicationMaster Container on failure. Besides, ApplicationMasters request and manage Containers, which grant rights to an application to use a specific amount of resources (CPU, Memory, etc.) on a specific host.

Once given resources by the ResourceManager, ApplicationMaster contacts with the NodeManager to start individual tasks. For example, using the MapReduce framework, these tasks would be mapper and reducer processes.

On previous figure, we have two ApplicationMasters running within the cluster, one of which has three Containers (the red client) and one that has one Container (the blue client). Note that the ApplicationMasters run on cluster nodes and not as part of the ResourceManager, thus reducing the pressure on a central scheduler. Also, because ApplicationMasters have dynamic control of Containers, cluster utilization can be improved.

Next, we are going to describe application execution flow on YARN. Figure below shows how YARN allocates resources and runs an application:

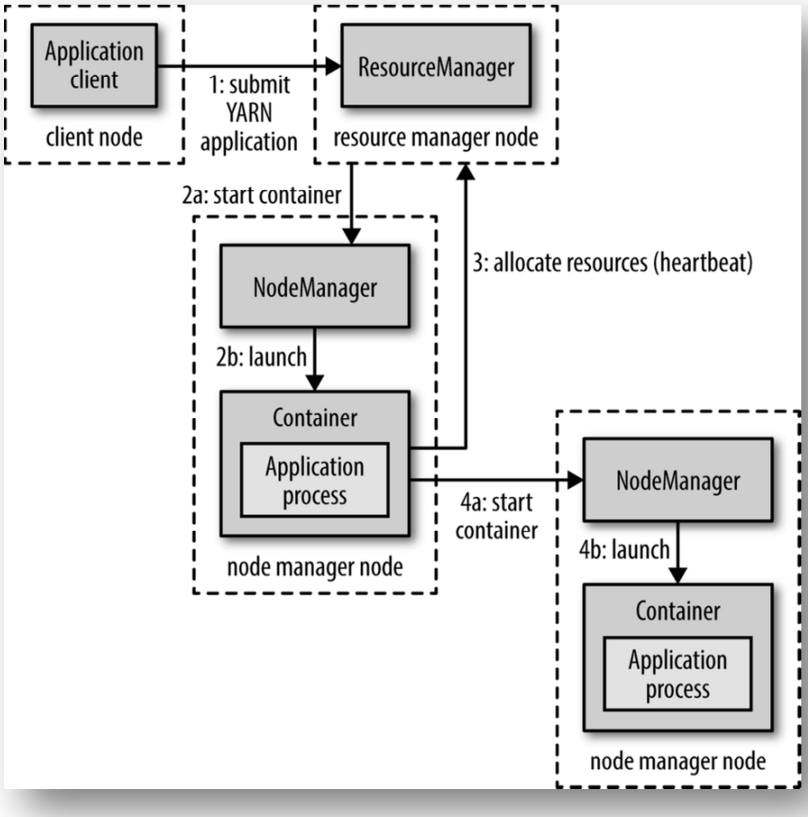


Figure 3.8 YARN Cluster: Running an application

- **Step 1**: To run an application on YARN, a client contacts with the ResourceManager and asks it to run an ApplicationMaster process.

- **Step 2**: The ResourceManager then finds a NodeManager that can launch the ApplicationMaster process in a container (steps 2a and 2b).

- **Step 3**: In this point, the ApplicationMaster process has two options; it could run a computation in its own Container, or it could request more Containers from the ResourceManager. Usually, the last option is the most common for distributed applications.

- **Step 4**: Once given appropriate resources, the ApplicationMaster process uses them to run a distributed computation (steps 4a and 4b).

We have seen how YARN improves general performance and provides multiple benefits. The main differences between Hadoop 1.0 and Hadoop 2.0 are as below:

Hadoop 1.0	Hadoop 2.0
Only MapReduce application can be processed	Non MapReduce application can also be processed
Cluster Resource Management is handled by JobTracker process (which is component of MapReduce engine)	Cluster Resource Management is handled by YARN
Cluster Resource are not fully utilized due to concept of fixed number of map and reduce slots	Optimization of resources is better due to Central Resource Management. No more fixed map and reduce slots
JobTracker and TaskTracker processes are responsible for the execution of application	YARN (Resource Manager and NodeManager) replaces JobTracker and TaskTracker processes. No more JobTracker and TaskTracker processes in Hadoop 2.0

Table 3.1 Comparison table: Hadoop 1.0 vs Hadoop 2.0

To sum up, we can say that Hadoop 2.0 architecture is more isolated and scalable as compared to the earlier Hadoop 1.0 architecture. Besides, other tools can also perform data processing via YARN (YARN based execution model is more generic than earlier MapReduce based execution model).

3.1.4 Hadoop MapReduce

Hadoop MapReduce is a simple programming model to support the development of distributed applications (which process massive amounts of data stored in HDFS for example). Hadoop MapReduce hides underlying detail from the programmer, including details related to the parallelization of the computation, monitoring and recovery from failure, data management and load balancing onto the underlying physical infrastructure.

The key principle behind MapReduce is the recognition that many parallel computations share the same pattern; in other words:

- Break the input data into a number of chunks.
- Carry out initial processing on these chunks of data to produce intermediary results.
- Combine the intermediary results to produce final output.

The specification of the associated algorithm can be expressed in terms of two functions; one to carry out the initial processing and the second to produce the final results from the intermediary values. In this way, we have Map and Reduce functions:

- The Map function takes a set of key-value pairs as input and produces a set of intermediary key-values as output.
- The intermediary pairs are sorted by key value so that all intermediary results are ordered by intermediary key. This is broken up into groups and passed to Reduce instances, which carry out their processing to produce a list of values for each group (form some computations, this could be a single value).

The next figure illustrates the overall execution of a MapReduce program:

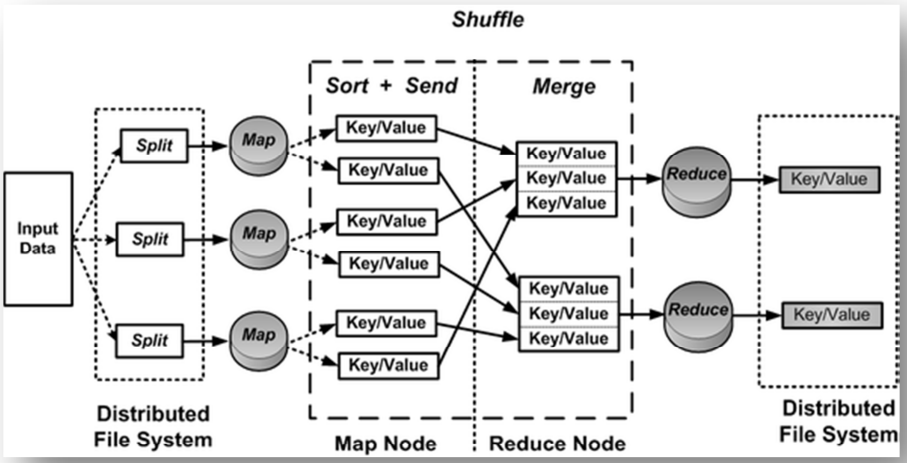


Figure 3.9 The overall execution of a MapReduce program

The process by which the system performs the sort and transfers the map outputs to the Reducers as inputs is known as "*Shuffle and Sort*". It is very important to keep in mind that the whole process (from the point where a Map produces output to where a Reduce consumes input) is very complex and its explanation and details are beyond the scope of this project [50].

Moreover, we show the overall MapReduce word count process, which is typically one of the simplest programs possible (besides, figure illustrates a simple input file, a simple input format as well as a simple process split). Next figure illustrates the whole process:

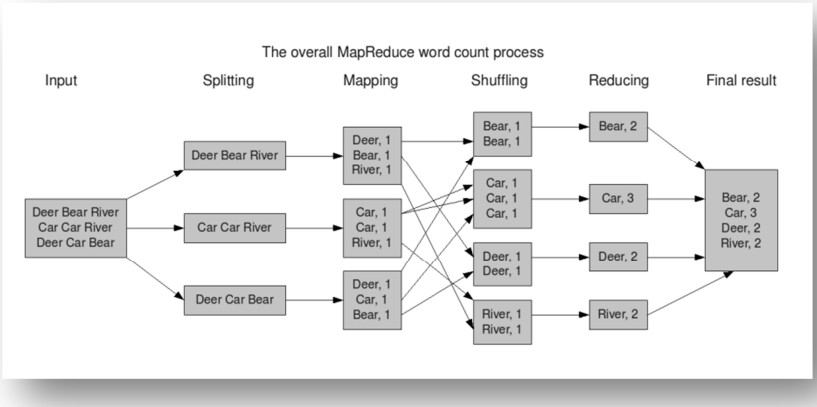


Figure 3.10 The overall Mapreduce WordCount problem

To better understand the operation of MapReduce, let us consider a simple example. As we mentioned earlier, in a Hadoop cluster, typically files are broken into 128MB blocks and each block is replicated three times on distinct nodes in the cluster. In this case, we have only one file, so it is composed of three blocks, making a total of nine blocks.

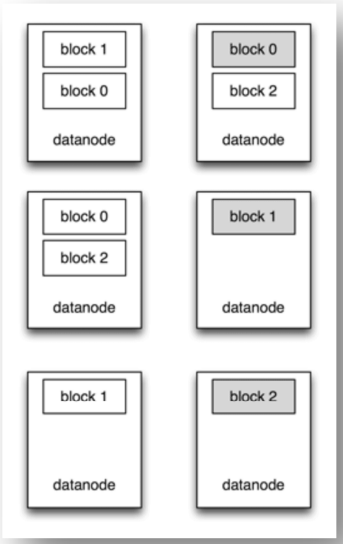


Figure 3.11 HDFS block division

Here is where MapReduce engine takes advantage of above properties. First, by choosing to run a Map function for each block, the amount of work that each Map does is relatively small (in the order of seconds) and can be performed in parallel across the cluster. Second, Maps generally runs on the same node as one of the block's replicas, which achieves data locality as well as minimizing network bandwidth.

The next figure shows three Maps instances running simultaneously on three different parts (called file splits) of the input dataset (the three highlighted HDFS blocks). The output from the maps is stored on local disk, and not written back to HDFS:

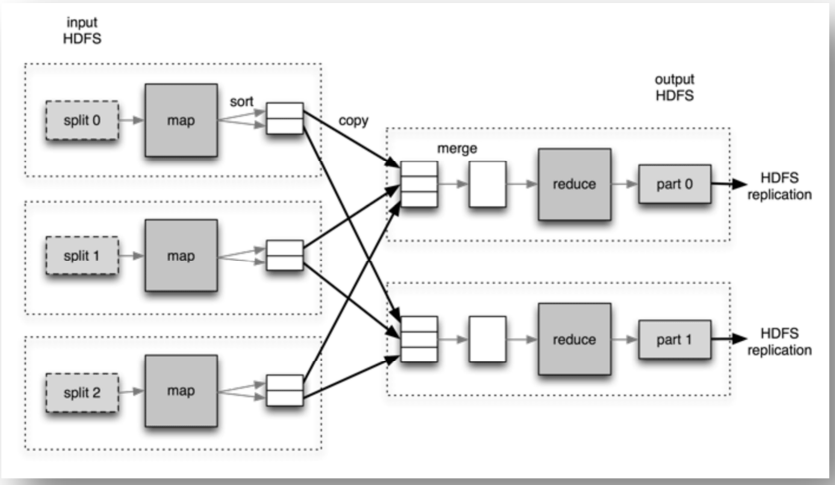


Figure 3.12 Three maps running simultaneously.

Notice that the Map outputs are partitioned, with one partition for each Reducer. The partitioning function is normally simple hash partitioning, but it is possible to override it. Here, we have shown two partitions, corresponding to two Reducers (on a typical cluster, it would be much larger.) Since the partitions for each Reducer are spread across the cluster, it is not possible for the Reducers to run on the same node as their input. Instead, the partitions are copied across the network to the Reducers in a process known as *"Shuffle and Sort"*, which is shown as dark arrows.

Finally, we have to take into account some key aspects such as:

- The number of Maps instances is completely dependent on total size of input, input split size and structure of input files (taking into account small files problem [51]).

- The number of Reduce instances is determined by *"mapreduce.job.reduces"* property, which is set by the *"setNumReduceTasks()"* method.

3.1.5 Hadoop execution flow

Once we have seen all main components of Hadoop, we going to describe the whole execution process of a MapReduce job; that is, the interaction among all components. The process of running a job is shown in the figure below:

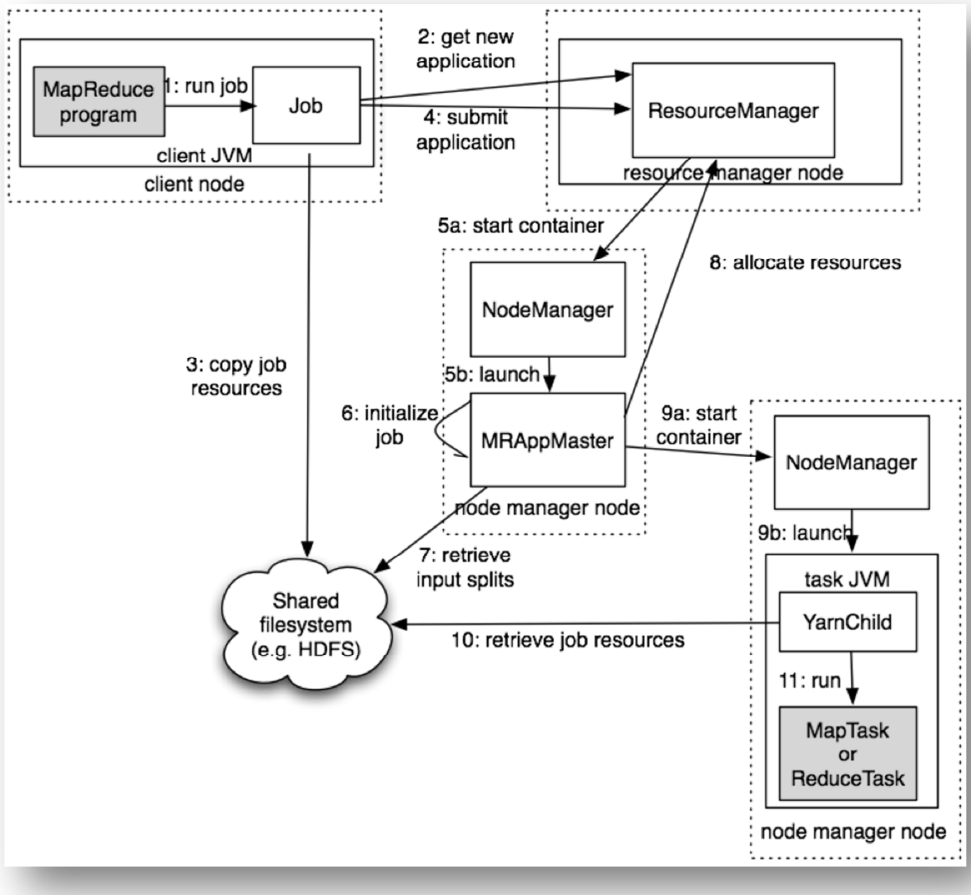


Figure 3.13 Hadoop execution flow

- **Step 1**: Client submits MapReduce job by interacting with Job objects (Client runs in its own JVM).

- **Step 2**: Job's code interacts with ResourceManager to acquire application ID.

- **Step 3**: Job's code computes input splits and copies all job resources to HDFS to make them available for the rest of de job.

- **Step 4**: Job's code submits the application to ResourceManager.
- **Step 5**: ResourceManager chooses a NodeManager with available resources. The scheduler allocates a container and then, the ResourceManager launches the application master's process under NodeManager management (steps 5a and 5b).

- **Step 6**: The ApplicationMaster for MapReduce jobs is a Java application whose main class is MRAppMaster. It initializes the job by creating a number of objects to keep track of the job's progress, as it will receive progress and completion reports from the tasks.

- **Step 7**: MRAppMaster retrieves the input splits computed in the client from the HDFS. Then, it creates a map task object for each split as well as the reduce tasks (which are determined by *themapreduce.job.reduces* property).

- **Step 8**: MRAppMaster negoatiates with ResourceManager for available resources (ResourceManager will select NodeManager with most resources).

- **Step 9**: Once a task has been assigned to a Container by the ResourceManager's scheduler, the ApplicationMaster starts the container by contacting the NodeManager (steps 9a and 9b).

- **Step 10**: The task is executed by a Java application whose main class is YarnChild. YarnChild acquires job resources from HDFS, which will be required to execute map and reduce tasks.

- **Step 11**: YarnChild executes map and reduce tasks.

3.2 Apache Spark

Apache Spark is a fast and general-purpose cluster computing system, which was originally developed in 2009 in UC Berkeley's AMPLab, and open sourced in 2010 as an Apache project. Spark is an in-memory data processing framework based on a generalization of MapReduce (multi-stage in-memory paradigm), which makes it much faster in processing than MapReduce (two-stage disk-based paradigm). According to certain studies, Spark enables applications in Hadoop clusters to run up to 100 times faster in memory and 10 times faster even when running on disk.

In MapReduce, intermediate data is stored in the disk; consequently, data access and transfer makes it lower. However, in Spark intermediate data is stored in-memory (as long as data fits in memory); additionally, Spark utilizes multiple threads instead of multiple processes to achieve high parallelism on a single node.

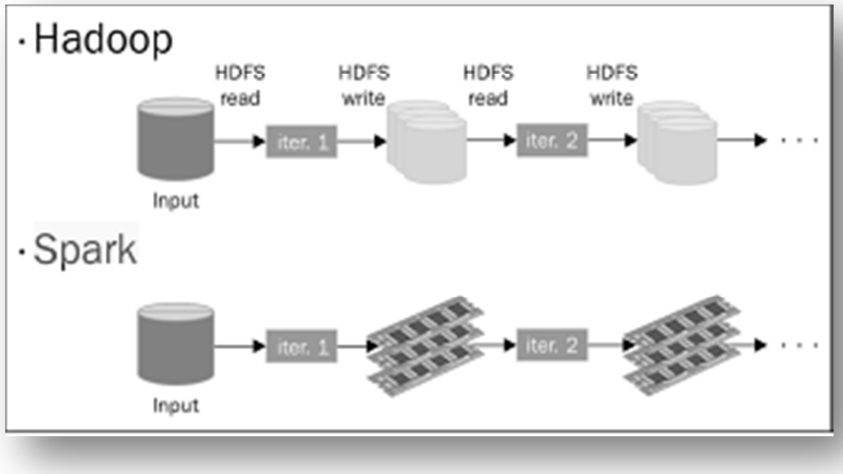


Figure 3.14 Multi-step data flows in Hadoop and Spark

Due to its high performance and parallelism, Spark is widely used for streaming data analytics, graph analytics, fast interactive queries and machine learning through different projects (see *3.2.2*). Spark had in excess of 465 contributors in 2014, making it not only the most active project in the Apache Software Foundation [52] but one of the most active open source Big Data projects.

Spark has numerous features and capabilities worth mentioning as follows:

- Can process iterative and interactive analytics (Spark can cache data and query it repeatedly).
- Many functions and operators available for data analysis (Spark supports more than just Map and Reduce functions).
- DAG [53] framework to design functions easily.
- In-memory based intermediate storage (Spark also supports on-disk execution engine).
- Easy to use and maintain.
- Written in Scala [54] and runs in JVM environment (although Spark applications can be written in Scala, Java, Python and R API's).
- Runs in environments such as Hadoop, Mesos [55], Standalone [56] or in Cloud environments [57].
- Great scalability (to over 8000 nodes in production environments).
- Interactive command line interface (in Scala or Python) for low-latency and scalable data exploration.
- Extensible through its rich framework (see *3.2.2*).

3.2.1 Hadoop vs Spark

Hadoop (see *Apache Hadoop)* is a Big Data processing technology (based on MapReduce programming model) which has been proven for 10 years and has demonstrated to be a great solution for processing large data sets with a parallel, distributed algorithm on a cluster.

MapReduce is a great solution for one-pass computations but not very efficient for use cases that require multi-pass computations and algorithms. That is, each step in the data processing workflow has one Map phase and one Reduce phase respectively. The job output data between each step has to be stored in the Distributed File System (such as HDFS) before the next step can begin. Therefore, this approach tends to be slow due to replication and disk storage. Moreover, programming issues can arise; developers need to know how to convert any use case into MapReduce pattern to adapt their solutions.

Spark is an in-memory cluster computing for iterative and interactive applications. It is designed to be an execution engine that works both in-memory and on-disk. Spark holds intermediate results in memory rather than writing them to disk, which is very useful especially when you need to work on the same dataset multiple times. Thus, Spark will attempt to store as much data as possible in memory and the rest of it will spill to disk (that is, it can store part of a data set in memory and the remaining data on the disk). Besides, through Spark operators, it is easy to perform external operations when data does not fit in memory. For this reason, it is responsibility of the developer to evaluate both data and memory requirements.

In this way, Spark allows programmers to develop complex and multi-step data pipelines using directed acyclic graph (DAG) pattern (it also supports in-memory data sharing across DAGs, so different jobs can work with the same data). Spark generally runs on top of existing Hadoop 2.0 infrastructure (see *Hadoop YARN (Yet Another Resource Negotiator)*) to provide improvements and additional functionalities.

Spark takes MapReduce to the next level with less expensive shuffles in the data processing. With capabilities like in-memory data storage and near real-time processing, the performance can be several times faster than other Big Data technologies. Therefore, Spark can be thought as an alternative to MapReduce due to the limitations and overheads of the latter, but no as a replacement (each one of them will be suitable in different situations).

Finally, we can summarize both alternatives as follows:

Paradigm	**MapReduce**	**Spark**
API	Poor (Java), although it is also possible to develop through Hadoop Streaming and Hadoop Pipes	Rich (Java, Python, Scala, R)
Boilerplate code	A lot	Almost no
Database query interface (SQL)	Hive	Spark SQL
Exploration of data	Not possible easily	Spark Shell allows quick and easy data exploration
Extensibility (Ecosystem)	A lot of tools available but integration is not easy (it requires a lot of effort)	Unified infrastructure (it unifies a lot of projects like Spark SQL, Spark Streaming and son on) into a single abstraction of RDD
Fault Tolerance	Through persisting the results of each of phases	Through immutability of RDD
In-memory computations	Not possible	Possible
Iterative processing	Non trivial	Straightforward
Performance	High latency	Low latency
Testability	Possible via libraries but non trivial	Very easy through Spark Shell
Workflow type	Poor (two-phases paradigm), only two possible phases like Map and Reduce operations	Rich (multi-stage paradigm) many stages of processing are possible

Table 3.2 Comparison table: MapReduce vs Spark

3.2.2 Spark framework

Spark contributors have utilized the core Spark Framework and have developed different libraries on top of Spark to improve its capabilities. These libraries can be plugged in to Spark according to requirements:

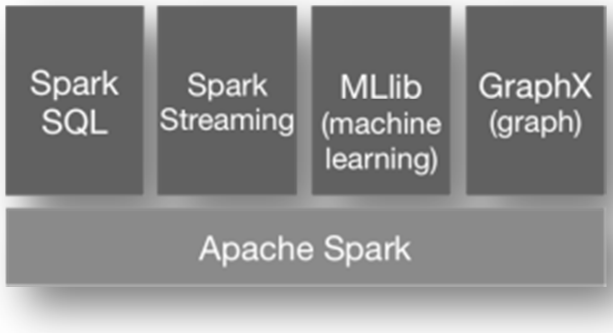


Figure 3.15 Apache Spark Ecosystem

- **Apache Spark (Spark Core and Resilient Distributed Datasets (RDD's))**: Spark Core is the foundation of the overall project. It provides distributed task dispatching, scheduling, and basic I/O functionalities.

  The fundamental programming abstraction is called Resilient Distributed Datasets (RDD's) [12] a logical collection of data partitioned across machines. RDD's can be created by referencing datasets in external storage systems (such as HDFS) or through transformations operations (e.g. map, filter, reduce, join) on existing RDDs. RDD's simplifies programming complexity because the way applications manipulate RDDs is similar to manipulating local collections of data. We will deepen around this concept in next subchapters (see *Resilient Distributed Datasets (RDDs)*).

- **Spark SQL**: Spark SQL is a wrapper of SQL on top of Spark. It transforms SQL queries into Spark jobs to produce results. In other words, it provides the capability to expose the Spark datasets over JDBC API as well as running the SQL like queries on Spark data using traditional BI and visualization tools.

- **Spark Streaming**: Spark Streaming is a library that enables Spark to perform scalable, faul-tolerant, high throughput system to process streaming data in near real-time (a technique known as micro-batching, see *Stream processing*). Spark Streaming takes the input data from a source and breaks it into batches. Finally, the batch is stores as an internal dataset (RDD) for processing. Next figure shows the fundamental principle:

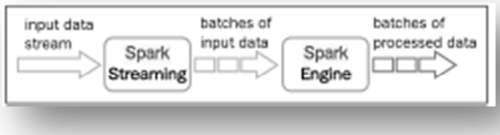


Figure 3.16 Spark Streaming fundamental principle

- **MLlib (machine learning)**: MLlib [58] is a scalable machine learning library that works on top of Spark. The machine learning library consists of common learning algorithms and utilities, including classification, regression, clustering and so on. It is considerably easier to use and deploy and its performance can be optimized to be 100 times faster than MapReduce.

- **GraphX (graph):** GraphX [59] enables working with graph-based algorithms (it has a wide variety of graph-based algorithms already implemented). Some examples are PageRank, Connected components, Label propagation and so on. At a high level, GraphX extends main RDD idea by introducing the concept of Resilient Distributed Property Graph (RDPG) [60]: a directed multi-graph with properties attached to each vertex and edge, as well as different fundamental operators.

3.2.3 Spark architecture

Broadly, Spark requires a Cluster Manager and a Distributed Storage System. For cluster management, Spark currently supports Standalone (native Spark cluster that makes it easy to set up a cluster), Hadoop YARN or Apache Mesos (in addition of Amazon EC2, which launches scripts that make it easy to launch a Standalone cluster on Amazon EC2). For distributed storage, Spark can interface with a wide variety, including Hadoop Distributed File System (HDFS) which is the most common option, Cassandra, OpenStack Swift [61], Amazon S3 [62], Kudu [63] or even a custom solution can be implemented.

As we mentioned earlier (see *Hadoop architecture*) thanks to Hadoop 2.0 architecture, it is possible to implement other data processing engines than MapReduce. Due to the high modularity, the most common implementation of Spark for large cluster of machines is based on Hadoop 2.0 ecosystem; that is, HDFS as distributed storage and YARN as cluster manager.

Figure below shows the basic architecture of Hadoop 2.0 ecosystem, where we can run natively many processing solutions:

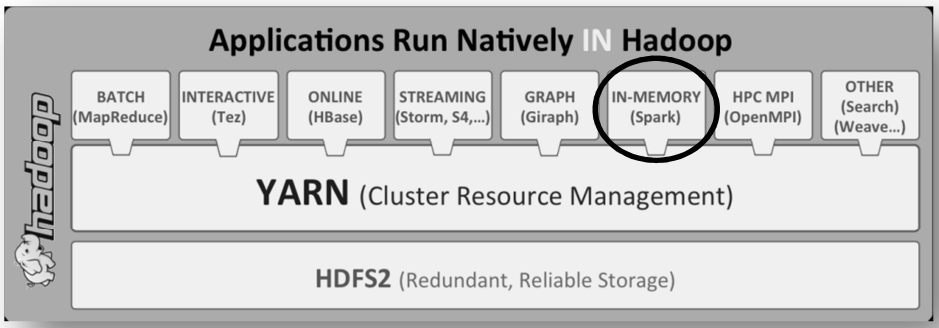


Figure 3.17 Hadoop 2.0 ecosystem

Previously (see *Hadoop HDFS (Hadoop Distributed File System)*, we explained and described the most important features of HDFS. In the same way (see *Hadoop YARN (Yet Another Resource Negotiator)*, we analyzed both architecture and core behavior of YARN. In this point, it is also important to understand why YARN is the recommended cluster resource manager for Spark; next we can see several benefits over Spark Standalone and Mesos:

- User can dynamically share and centrally configure the same pool of cluster resources among all frameworks that run on YARN.

- User can use all the features of YARN schedulers for categorizing, isolating, and prioritizing workloads.

- User can choose the number of executors to use.

- Spark can run against Kerberos enabled Hadoop clusters and use secure authentication between its processes.

Once again, the best combination (distributed storage system and cluster manager) in terms of solution is not unique, so that we need to keep in mind many factors such as the size of our commodity cluster, application requirements, type of scheduler and associated policy, etc.

All things considered, we can examine the top tier of Spark architecture as follows:

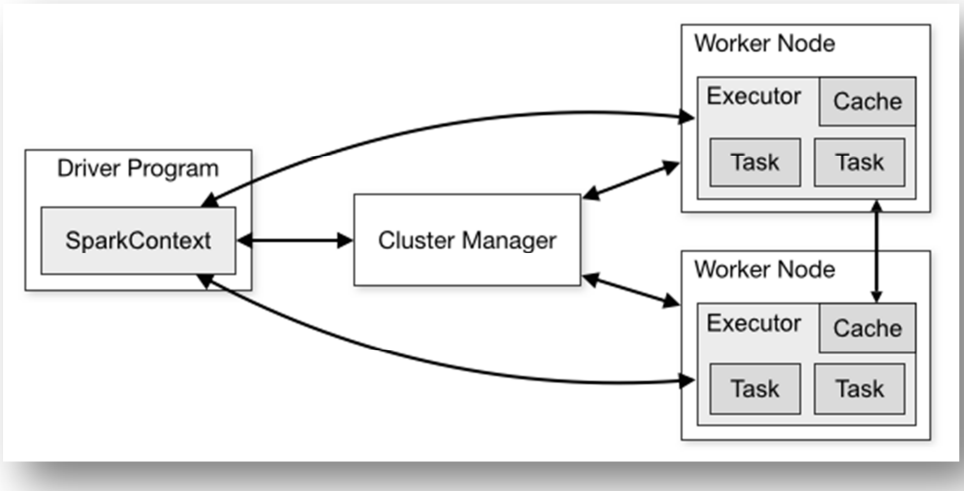


Figure 3.18 Spark architecture

As we can see in figure above, Spark Architecture follows typical master/worker architecture and consists of Driver Program, Cluster Manager and Executors.

Spark applications run as independent sets of processes on a cluster, which are coordinated in terms of job flow and scheduling tasks by the SparkContext object in the main program (called Driver Program). Next, once SparkContext has connected to Cluster Manager (which is responsible for starting executor processes and where and when they will be run), Spark acquires Executors on nodes in the cluster, which are processes that run computations and store data of the application. Next, it sends application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext can send tasks to the executors to run.

There are several things to note about Spark architecture:

- Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs). However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system.

- Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes and these communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g. Mesos/YARN).

- When running Spark on YARN, each Spark executor runs as YARN Container (collection of physical resources such as CPU cores or memory on a single node at a cluster).

- Tasks represent a unit of work on a partition of a distributed dataset. Besides, tasks can share variables through cache partition.

- When running Spark on YARN, it supports two modes for running: Yarn-Client mode and Yarn-Cluster mode:

    - In Yarn-Client mode, the Driver Program (SparkContext) runs in the Client Process and the ApplicationMaster is only used for requesting resources from YARN. This mode makes sense for interactive and debugging purposes since user can see application's output immediately (on the Client Process side).

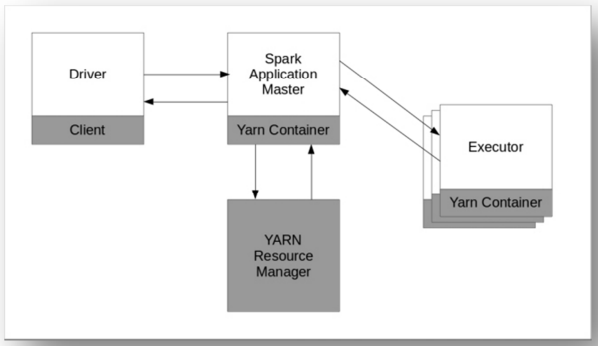


Figure 3.19 YARN Client-Mode

    - In Yarn-Cluster mode, the Driver Program (SparkContext) runs in the ApplicationMaster process. This means that the same process (which runs inside a YARN Container) is responsible for both driving the application and requesting resources from YARN. In this case, the Client can go away after initiating the application. This mode makes sense for production jobs.

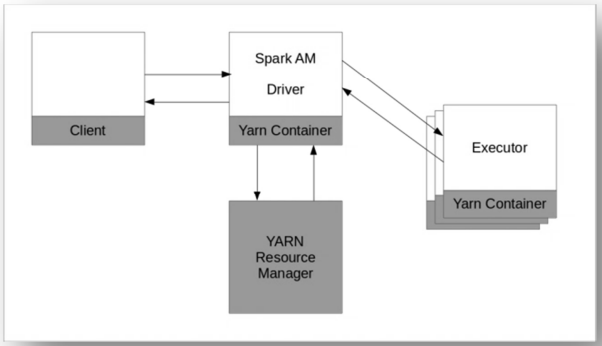


Figure 3.20 YARN Cluster-Mode

.

Finally, we can summarize both modes as follows:

Mode	YARN Client-Mode	YARN Cluster-Mode
Driver program runs in:	Client side	ApplicationMaster process
Main purpose:	Debugging, testing	Production environments, long jobs
Persistent services	YARN ResourceManager and NodeManager	YARN ResourceManager and NodeManager
Supports Spark Shell?	Yes	No
User needs to wait to get the result?	Yes	No
Who requests resources?	ApplicationMaster process	ApplicationMaster process
Who starts executor processes?	YARN NodeManager	YARN NodeManager

Table 3.3 Comparison table: YARN Client-Mode vs YARN Cluster-Mode

3.2.4 Resilient Distributed Datasets (RDDs)

The key concept in Apache Spark is the Resilient Distributed Dataset (RDD). An RDD is a fault-tolerant collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. This abstraction was proposed in a 2012 research paper, Resilient Distributed Datasets: *A Fault-Tolerant Abstraction for In-Memory Cluster Computing* [12].

Conceptually, RDDs are an immutable resilient distributed collection of records, which can be stored in the volatile memory or in a persistent storage (HDFS, HBase, etc.). RDDs can only be created through deterministic operations on either other RDDs or data in stable storage.

Broadly, RDDs can perform two types of operations transformations and actions. On the one hand, transformations do not return a single value; they return a new-modified RDD based on the original (nothing is evaluated). Several transformations are available through the Spark API, including map, filter, groupbykey or union among others. On the other hand, actions evaluate and return a new value; when an action is called on a RDD object, all the data processing queries are computed at the time and the result is returned. Some examples of actions supported by the Spark API include count, collect, reduce or lookup among others. Another key aspect is the *lineage* of an RDD. We could define the *lineage* like the information about how an RDD was derived from other datasets. This is a powerful property; if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition. In this way, lost data can be recovered, often quite quickly, without requiring costly replication. The figure below shows the basic idea of execution flow:

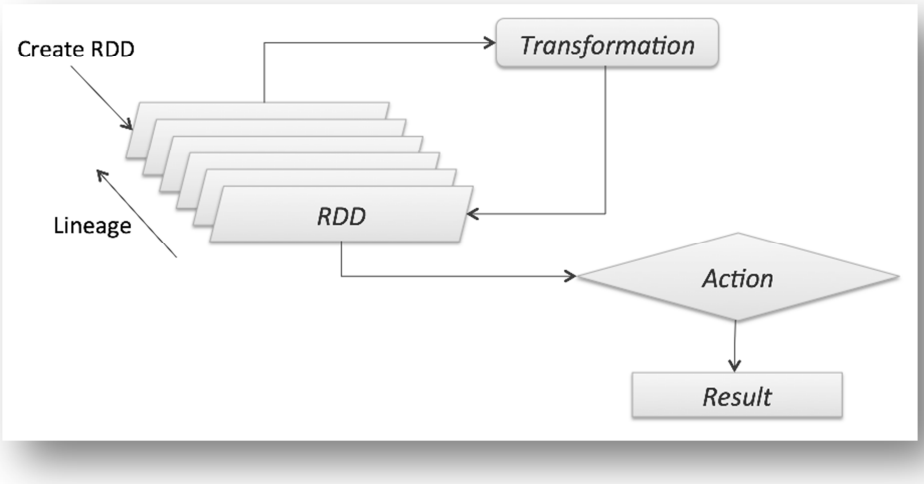


Figure 3.21 RDD execution flow

Internally, logical graph of RDD operations can be represented as a Directed Acyclic Graph (DAG) of operators (where the programming model follows a pattern in which data flows in multiple steps). DAG is compiled into stages (set of tasks that run in parallel) and each stage is executed as a series of tasks (fundamental unit of execution). Thus, each Spark job is represented by a DAG of task stages to be performed on the cluster. Next figure illustrates the execution graph of a basic operation:

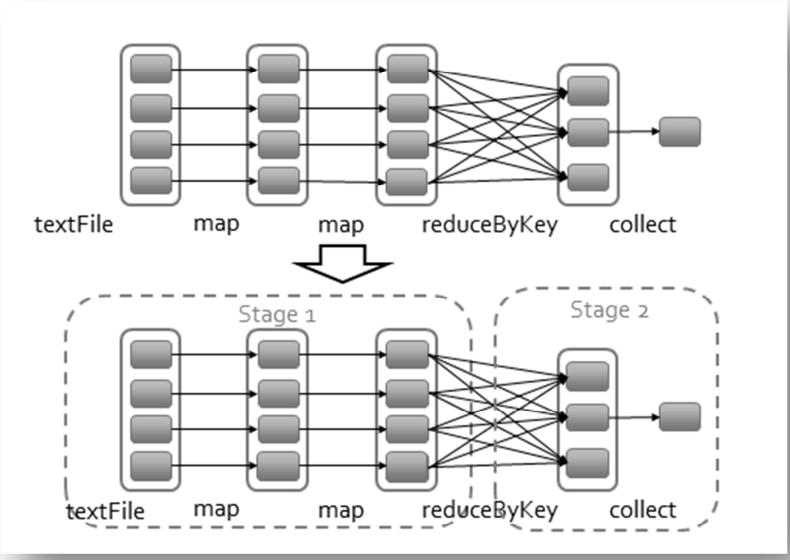


Figure 3.22 DAG execution graph

It is also interesting to understand the execution pipeline of a task. It is composed of three basic phases:

- **Fetch:** input from InputFormat or a Shuffle
- **Execute:** execute the task
- **Write:** materialize task output as Shuffle or driver result

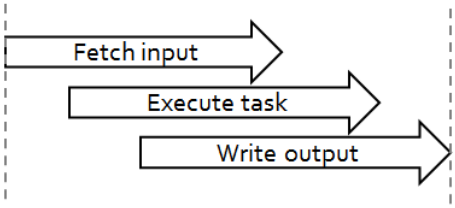


Figure 3.23 Task pipelined execution

The same idea can be transferred to a multicore environment as follows:

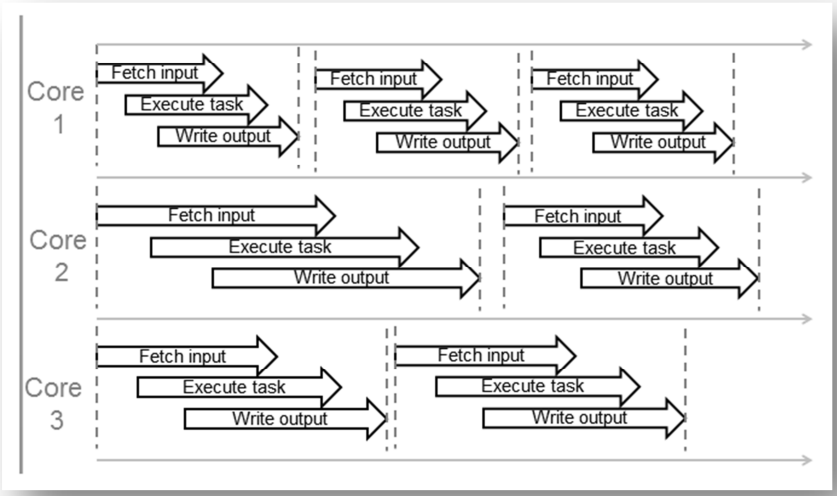


Figure 3.24 Task pipelined execution in multicore system

Additionally, we can talk about RDD dependencies, which can be of two types: narrow and wide. Narrow dependencies occur when a partition of an RDD (chunks in Hadoop's terminology, block in HDFS terminology) is used by only one partition of the next RDD. Wide dependencies occur when a partition of an RDD is used by multiple partitions in the next RDD (usually in groups and joins operations). The following figure shows the two types of dependencies

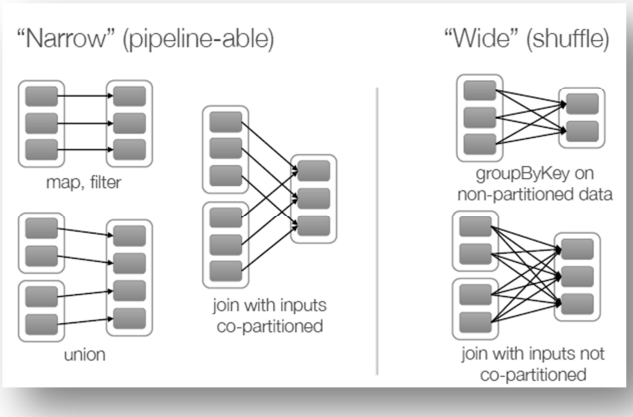


Figure 3.25 RDD: Narrow and Wide dependencies

We have seen how by using RDDs, programmers can pin their large data sets to memory, thereby supporting high-performance and iterative processing. Compared to reading a large data set from disk (for every processing iteration), in-memory computing is an efficient solution. Thanks to this approach, computation becomes faster. It is very important to keep in mind that the whole process of Spark application execution (at internal level) is very complex and its explanation and details are beyond the scope of this project [64].

Finally, we can mention some features of RDDs as follows:

- Resilient and fault tolerance. In case of any failure they can be rebuilt according to the data stored
- Distributed
- Datasets partitioned across cluster nodes
- Immutable
- Memory-intensive
- Caching levels conFigureble according to the environment
- Each RDD consists of 5 basic properties: partitions, dependencies, compute as well as partitioner and preferred locations (both of them optional)

4 Big Data architecture implementation

In this chapter we present our Big Data architecture implementation. In the first part, we show both low-level logical architecture (hardware tier, specifications) and high-level logical architecture (software tier, associated roles) as an architecture solution. In the second part, we perform a full installation of a platform for Big Data; the procedure includes installing and tuning a GNU/Linux environment as well as the use of Cloudera (Cloudera's open-source Apache Hadoop distribution, CDH) as infrastructure solution.

4.1 Solution architecture

Our cluster environment is located in the cloud and it is composed of four physical servers. It is completely reachable through Internet (although only one node must be reachable using HTTPS and SSH connections). In Big Data terms, we could talk about a Hadoop Cluster in the Cloud. A brief description of the hardware environment can be found in previous subchapters (see *Requirements*).

The detailed low-level logical architecture is shown below:

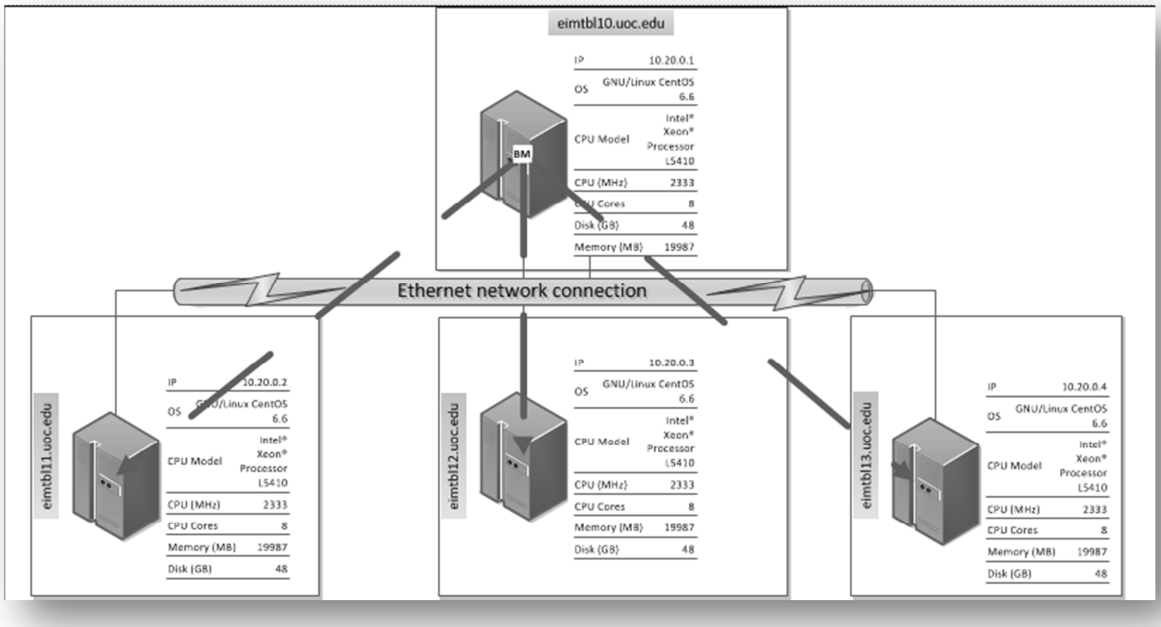


Figure 4.1 Low-level logical architecture

In the same way, from the point of view of logical roles (HDFS, YARN, Haddop and Spark), we can rank them as follows:

- **eimtbl10.uoc.edu** (master node): HDFS NameNode, HDFS Secondary NameNode, YARN ResourceManager, YARN Job History Server, Spark History Server (History Servers hold information about the history of completed applications).

- **eimtbl11.uoc.edu** (slave node): HDFS DataNode, YARN NodeManager.

- **eimtbl12.uoc.edu** (slave node): HDFS DataNode, YARN NodeManager.

- **eimtbl13.uoc.edu** (slave node): HDFS DataNode, YARN NodeManager.

The detailed high-level logical architecture is shown below:

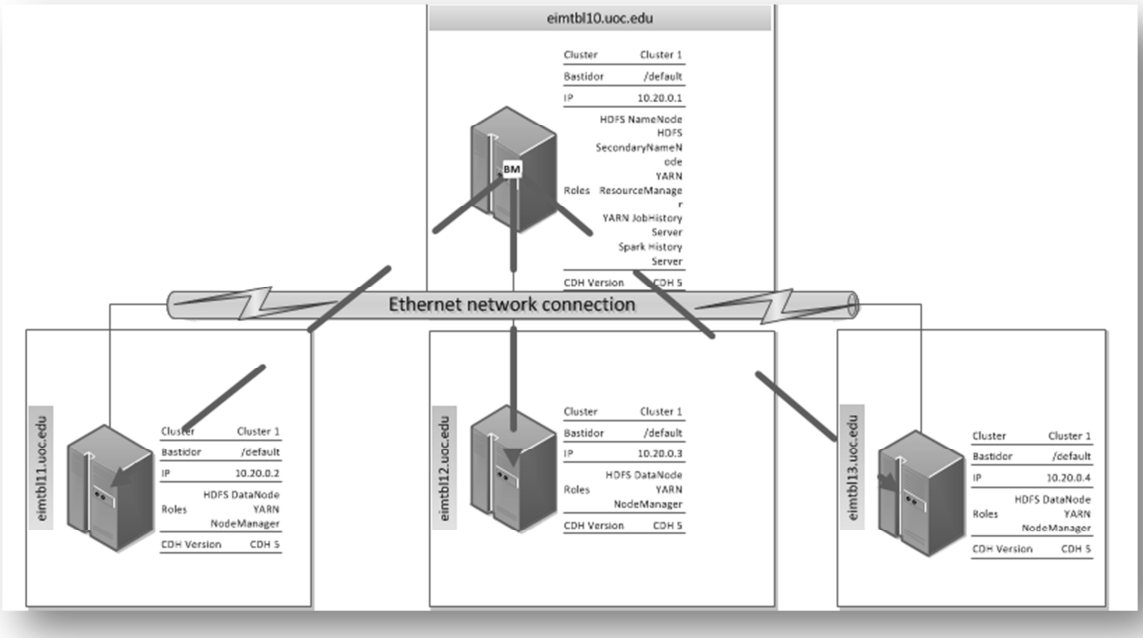


Figure 4.2 High-level logical architecture

4.2 Cloudera: The Platform for Big Data

Cloudera (Cloudera's open-source Apache Hadoop distribution, CDH) is the platform for Big Data most known. Also known as *The Platform for Big Data*, it will allow us to have a free Hadoop architecture/environment (including Apache Spark) in a few steps thanks to one of its key components: Cloudera Manager.

However, before we can work with Cloudera as shuch, we need to get ready our environment. That is, we need to perform a full installation of a compatible GNU/Linux operating system (CentOS 6.X) in our cluster of four machines, as well as tuning of the environment in order to make it suitable for the use of Cloudera Manager (5.X).

Explaining the entire process and every step performed (from the operating system installation to Cloudera Manager installation) is beyond the scope of this project. Since we are performing a demonstration and proof of concept deployment, Cloudera Manager deployment is the recommended method for installing CDH and managed services (Cloudera Manager automates the installation and configures the entire installation process, where the most options are by default). In any other case, as for example production deployments or special cases, we might need specific requirements (such as external database), so that we should perform a manual installation of certain components. Anyway, it is interesting to keep in mind a few details, especially the part of tuning of the environment. For this reason, we show the most important requirements and steps that must be performed in order to avoid compatibility problems between operating system level and Cloudera Manager:

- Accomplish CDH/ Cloudera Manager hardware requirements [65]
- Recommended GNU/Linux operating system: CentOS 6.X (update 4 or later recommended)
- Disable SELinux (Security-Enhanced Linux)
- Disable IPtables firewall
- Share the same /etc/hosts file for the four servers (all of them with static IP address as well as a proper DNS resolution)
- Select a proper SSH authentication method (preferably Public key authentication or Password authentication according to our needs)
- Select the appropriate runlevel for slaves servers (preferably equal to 3) to improve overall performance
- Set default Swappiness value equal to 0 to improve overall performance
- Disable transparent hugepage compaction to improve overall performance

To conclude this subchapter, we show a few captures of the installation process, which demonstrates how our Hadoop Cluster finished successfully by using Cloudera Manager (note how we mapped logical roles to appropriate servers such as we mentioned earlier):

Consulta ampliada	Nombre de host (FQDN)	Dirección IP	Actualmente gestionado	Resultado
eimtbl10.uoc.edu	eimtbl10.uoc.edu	192.168.1.37	No	✓ Host preparado: tiempo de respuesta de 1 ms.
eimtbl11.uoc.edu	eimtbl11.uoc.edu	192.168.1.42	No	✓ Host preparado: tiempo de respuesta de 10 ms.
eimtbl12.uoc.edu	eimtbl12.uoc.edu	192.168.1.43	No	✓ Host preparado: tiempo de respuesta de 2 ms.
eimtbl13.uoc.edu	eimtbl13.uoc.edu	192.168.1.44	No	✓ Host preparado: tiempo de respuesta de 8 ms.

Figure 4.3 Cloudera Manager: host detection successfully

Nombre de host	Dirección IP	Progreso	Estado	
eimtbl10.uoc.edu	192.168.1.37		✓ La instalación se ha realizado correctamente.	Detalles
eimtbl11.uoc.edu	192.168.1.42		✓ La instalación se ha realizado correctamente.	Detalles
eimtbl12.uoc.edu	192.168.1.43		✓ La instalación se ha realizado correctamente.	Detalles
eimtbl13.uoc.edu	192.168.1.44		✓ La instalación se ha realizado correctamente.	Detalles

Figure 4.4 Cloudera Manager: cluster installation successfully

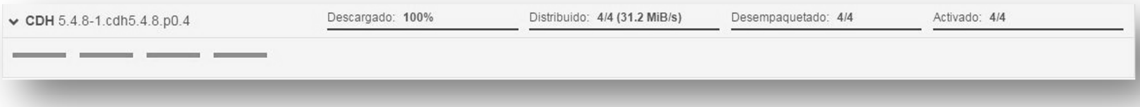


Figure 4.5 Cloudera Manager: parcels deployment successfully

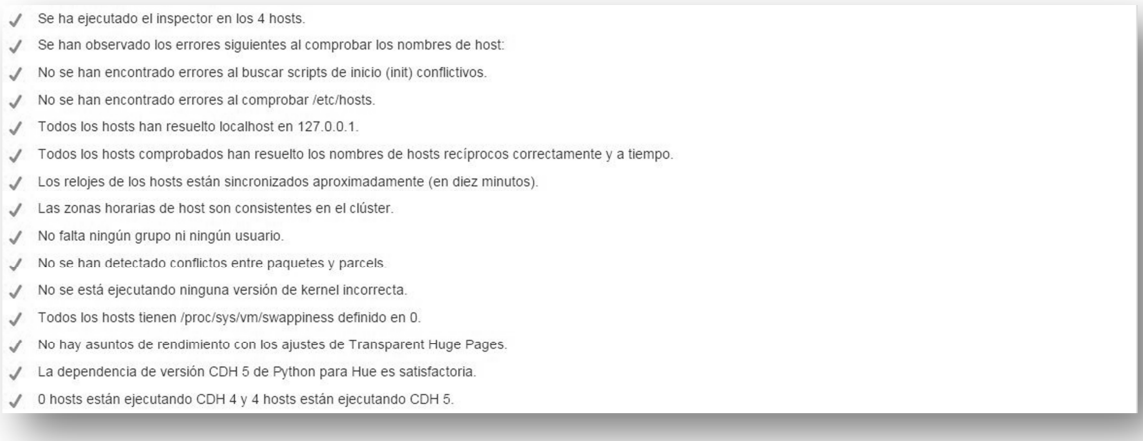


Figure 4.6 Cloudera Manager: all requirements met successfully

Figure 4.7 Cloudera Manager: core installation with Spark

Figure 4.8 Cloudera Manager: HDFS roles

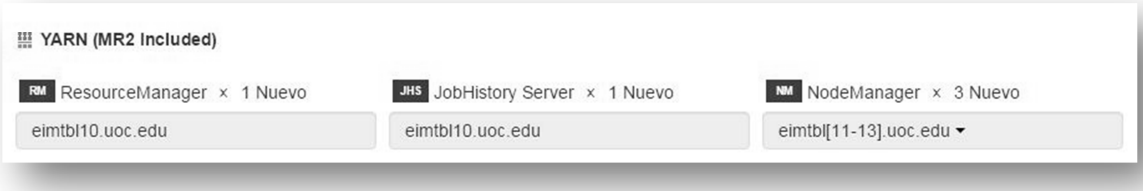


Figure 4.9 Cloudera Manager: YARN roles

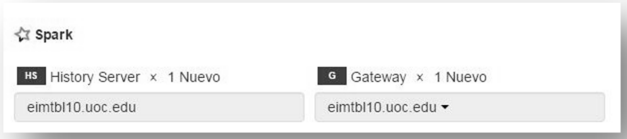


Figure 4.10 Cloudera Manager: Spark roles

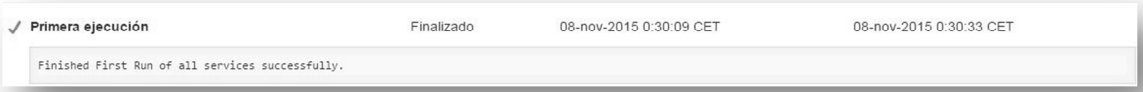


Figure 4.11 Cloudera Manager: first successful execution

Figure 4.12 Cloudera Manager: final message

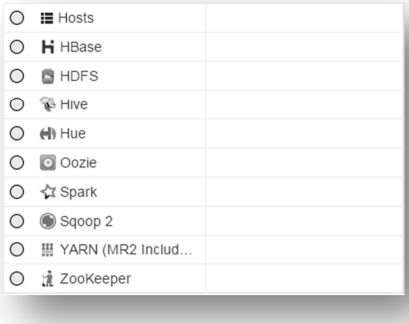


Figure 4.13 Cloudera Manager: initial status

5 System performance analysis and benchmarking

In this chapter we describe the methodology that we have followed in order to perform system performance analysis and benchmarking. The main goal is to perform through some benchmarks available, system performance analysis and benchmarking of both large scale distributed processing solutions (Hadoop and Spark). In this point, we present three different applications, where each one of them perform different tasks and follow a different programming paradigm. Additionally, we show different performance metrics as well as statistical data.

5.1 WordCount problem

The easiest problem in MapReduce paradigm is the Word Count problem; for this reason, it is also called MapReduce's "*Hello World*" by many people. Maybe, Word Count is the best example for understanding the basic concepts of MapReduce paradigm.

The goal of the Word Count problem is to find the number of occurrences of each word in a given input set, which is typically a text file. So, the input is text/s files and the output is text/s files, each line of which contains a word and the count of how often it occurred, separated by a tab.

At MapReduce level (Hadoop), each mapper takes a line as input and breaks it into words. It then emits a key/value pair of the word and 1. Each reducer sums the counts for each word and emits a single key/value with the word and sum. More information about this example can be found in previous subchapters (see *Hadoop MapReduce*). In Spark, algorithm follows mainly the same methodology; that is, through "*map*" and "*reduceByKey*" RDD operations, it can perform the same task. Other steps are done by Spark internal core.

Hadoop comes with its WordCount java implementation and Spark comes with both java and python implementations respectively. However, in order to better perform system performance analysis and benchmarking, we have developed our own version of Word Count example in python programming language for MapReduce framework (we have chosen python over java due to its simplicity at code level). The main idea behind this decision is the next: both implementations must be implemented in the same programming language in order to avoid performance problems. Some examples of problems that can arise are the next: different intermediate code generated (bytecode), performance issues between interpreters and jvm runtimes, data serialization and so on.

Moreover, it is important to understand that we only need to implement both map and reduce functions; MapReduce engine handles the process known as *"Shuffle and Sort"* by itself. Further explanation about it can be found in previous subchapters (see *Hadoop MapReduce*).

Our MapReduce python implementation will run on Hadoop through Hadoop Streaming, which is a utility that comes with the Hadoop distribution and allows us to create and run map/reduce jobs with any executable or script as the mapper and/or the reducer. In Spark, we will run its python implementation of the Word Count problem through Spark-submit script (this is the only way to submit jobs on Spark) [66].

In Hadoop, Hadoop Streaming utility takes input from standard input (stdin) and provides output through standard output (stdout). Internally, the streaming engine reads and writes data appropriately, invoking applications as needed. Figure below shows Hadoop Streaming basic data flow:

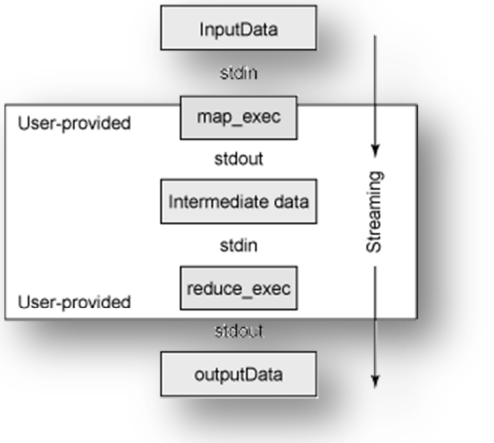


Figure 5.1 Hadoop Streaming data flow

In Spark, Spark-submit script allows us to submit compiled Spark applications (application jar or python file) on YARN with several options. In our case, we will run a python application in YARN cluster mode. Further explanation about Spark architecture and Spark execution flow can be found in previous subchapters (see *Spark architecture*). Next figure illustrates basic data flow through Spark-submit in a python environment with YARN:

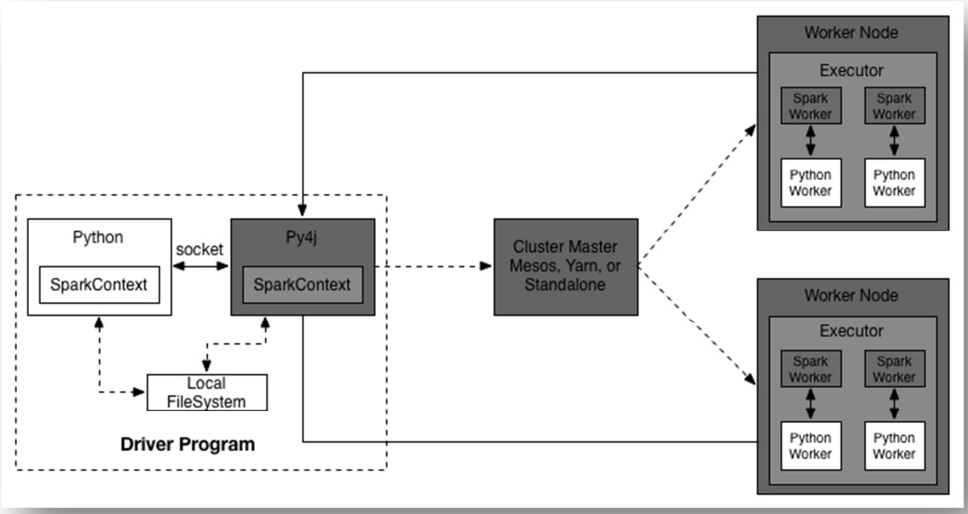


Figure 5.2 Spark-submit in a python environment with YARN

5.2 TeraSort problem

The TeraSort benchmark is probably the most well-known Hadoop benchmark. Back in 2008, Yahoo! set a record by sorting 1 TB of data in 209 seconds [67] – on a Hadoop cluster of 910 nodes as Owen O'Malley of the Yahoo! Grid Computing Team reports. One year later in 2009, Yahoo! set another record by sorting a 1 PB (1'000 TB) of data in 16 hours [68] on an even larger Hadoop cluster of 3800 nodes (it took the same cluster only 62 seconds to sort 1 TB of data, easily beating the previous year's record!).

Basically, the goal of TeraSort problem is to sort 1TB of data (or any other amount of data) as fast as possible. It is an ideal benchmark for testing both storage layer (HDFS) and processing layer (such as MapReduce or Spark in our case) of a Hadoop cluster.

Internally, TeraSort benchmark consists of the following three steps:

1. Generating the input data via TeraGen benchmark.
2. Running the actual TeraSort benchmark on the input data.
3. Validating the sorted output data via TeraValidate benchmark.

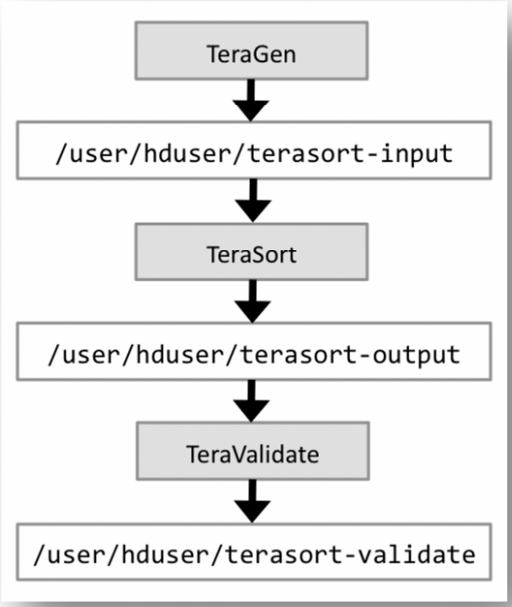


Figure 5.3 Terasort benchmark data flow

As we can see in the figure above, TeraGen generates random data that can be conveniently used as input data for a subsequent TeraSort run. TeraGen can generate many rows of data as we wish, each of which having a size of 100 bytes by default (for example, 10000000000 rows * 100 bytes row to achieve 1TB of data). The rows follow the next data format:

10 bytes key | 2 bytes break | 32 bytes acsii/hex | 4 bytes break | 48 bytes filler | 4 bytes break | \r\n

Next, TeraSort runs the actual TeraSort benchmark (the sorting is performed by the 10-byte ASCII key). Finally, TeraValidate validates the sorted output data of TeraSort.

At MapReduce level (Hadoop), each phase will manage both map and reduce tasks in a different manner:

1. TeraGen will run map tasks to generate the data (one for each HDFS block of data as usual) and will not run reduce tasks.

2. TeraSort will run map tasks to sort the data (one for each HDFS block of data as usual) and will be one reduce task by default (it will depend on environment-specific configuration).

3. TeraValidate will run map tasks to validate the data (one for each HDFS block of data as usual) and will be one reduce task by default (again, it will depend on environment-specific configuration).

In Spark, algorithm follows mainly the same methodology. On the one hand, Teragen generates records (data output) through random number generator and some random bytes up to 100 bytes. On the other hand, TeraValidate performs different comparisons in order to determine proper order of data. The main phase (TeraSort) just executes "*sortByKey"* function on the RDD to sort data. Other steps are done by Spark internal core.

Hadoop comes with its TeraSort java implementation; however, Spark does not have any implementation for this benchmark. In order to compare both frameworks efficiently, we have two choices: to develop our own version for Spark or find some equivalent code, which must be used by a huge community (in this way the comparison will be valid and reliable). Fortunately for us, there is a version written by one member of DataBricks [69] community (Evan Higgs) at Github [70], which is highly used for similar purposes to ours. It is developed in scala language and follows the same methodology of the MapReduce version (the three specific phases shown above); besides, it is fully compatible with any input data used by the MapReduce version. To make it suitable, we only need to package the scala code by using Apache Maven [71].

However, if we take a look at Evan's page, we can observe the next problem at "Known issues" section: "*This terasort doesn't use the partitioning scheme that Hadoop's Terasort uses. This results in not very good performance*." Specifically, Evan's version of Terasort uses a very simple partitioner (which uses the first 7 bytes for determining the partition of a key/value pair, which can result in performance problems). Custom partitioning in Spark is an advanced technique (optional) which tries to achieve better performance by balancing partitions (splits of RDD) among worker nodes properly (overriding Spark defaults options [72])

While MapReduce implementation uses 12 reducers as range partitioner by default (this number can be defined by setting "*mapreduce.job.reduces*" variable), Evan's implementation uses "random" value for such purpose. Since it exists one job/task for each partition and both frameworks work better with a small number of large files than a large number of small files, it is obvious that we do not want extra tasks which can incur in overhead problems or unbalanced distribution problems. Thus, in order to make a fair comparison between both frameworks (as well as avoiding performance problems), the main idea is to modify Evan's code to make it comparable with MapReduce implementation; in other words, we need to establish the same range partitioner as Hadoop TeraSort algorithm (12 tasks in our case).

Fortunately for us, we have found a good implementation which achieves previous concept [73].

The algorithm follows the same methodology described above (Evan Higgs algorithm) and it is fully compatible with any input data used by the MapReduce version. The only difference is that we can now specify range partitioner according to our needs. Additionally, we will make some code modifications (simplifying code arguments, establishing several constants according to our needs, as well as handling Spark's context in order to obtain job statistics properly). Only doing this, we can make a fair comparison.

Unlike WordCount problem (see *WordCount problem*) where both implementations are developed in the same programming language, in this case we perform system performance analysis and benchmarking from another perspective: each implementation is developed according to the native programming language of each processing framework respectively; that is, java implementation for Mapreduce and scala implementation for Spark. It is fair enough, since both implementations are fully compatible as we mentioned above. Moreover, we have to take into account that any scala source code is intended to be compiled to Java bytecode, so that the resulting executable code runs on a Java virtual machine. Besides, Java libraries may be used directly in Scala code and vice versa (language interoperability). Consequently, we might say that the generated bytecode is basically the same.

We will run both packages following the standard methodology for each one of them. On the one hand, in the Mapreduce case, we will use "hadoop-mapreduce-examples.jar" package through "hadoop jar" instruction, which contains TeraSort class. On the other hand (Spark's case), we will use "spark-submit" script to submit the compiled Spark application (package jar in our case) which also contains TeraSort class. In both cases, YARN will be used as a cluster manager in cluster mode.

5.3 PageRank problem

PageRank is an algorithm used by Google Search to rank websites in their search engine results. Developed by Larry Page [74] (one of the founders of Google), PageRank aims to measure the importance of website pages. According to Google:

> *PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.*

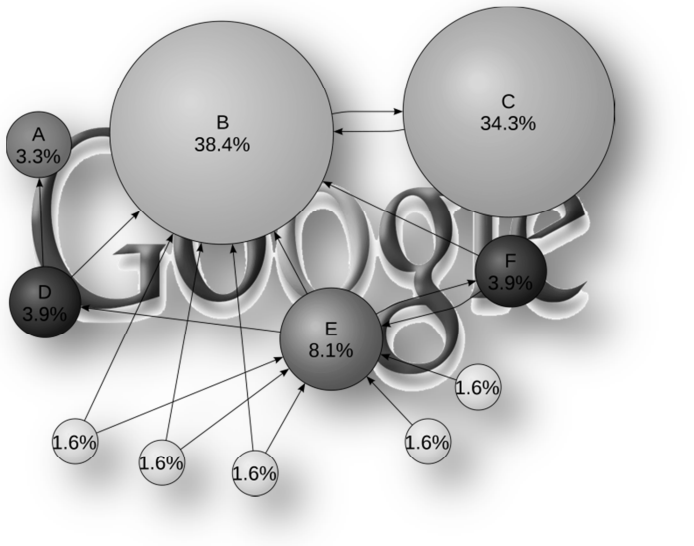


Figure 5.4 Google PageRank algorithm

Note that PageRank is not the only algorithm used by Google to order search engine results, but it is the first algorithm that was used by the company and it is the best-known. Although we treat with a straightforward implementation of a more complex algorithm, without any doubt it is a great parallel algorithm candidate for our case study.

In general terms, the main goal behind PageRank algorithm is to determine which page is more important than others. In other words; one web page will have high rank if it obtains links from many web pages or obtains link from a high-rank web page. We can illustrate it through a simple example: we have 4 pages (A, B, C and D as non-existing page) represented as follows:

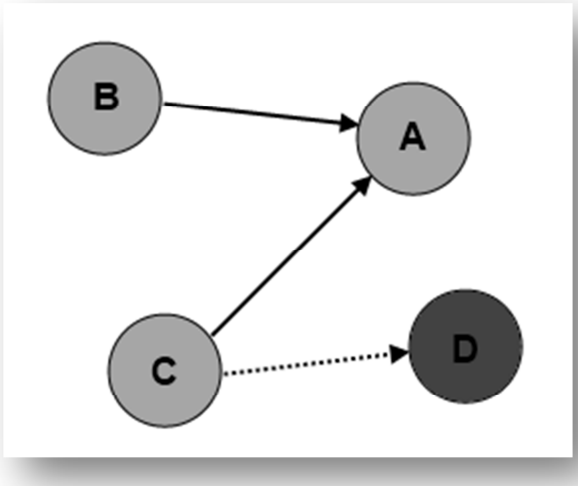


Figure 5.5 PageRank algorithm for a simple network

"D" is a page that has not been created yet or removed, but it is being linked from "C" page (we could talk about as a broken link). From previous figure, we can determine how rank of "A" page will be the highest, because it obtains point from "B" and "C" pages respectively. The formula of calculating the points is such as:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

However, we can still simplify the formula above as follows:

PageRank(A) = (1-d) + d (PageRank(B) / Count(B) + PageRank(C) / Count(C) + ...)

The *d* in the formula is the "*damping factor*" to simulate 'a random surfer' and it is usually set to *0.85*. Further explanations and details about it can be found in wiki pageranking page [75].

In this way, by applying previous formula, we can obtain rank of "A" page such as:

***PageRank*(A) = 0.15 + 0.85 * (PageRank(B)/outgoing links(B) + PageRank(C)/outgoing link(C))**

Thus, if we take into account basic steps of algorithm such as:

1. Start each page at a rank of 1.
2. On each iteration, have page p contribute"rankp / |neighborsp| to its neighbors.
3. Set each page's rank to 0.15 + 0.85 × contribs.

The first iteration of algorithm above would provide the following output:

- PageRank(A)= 1,425
- PageRank(B)= 0,15

- PageRank(C)= 0,15

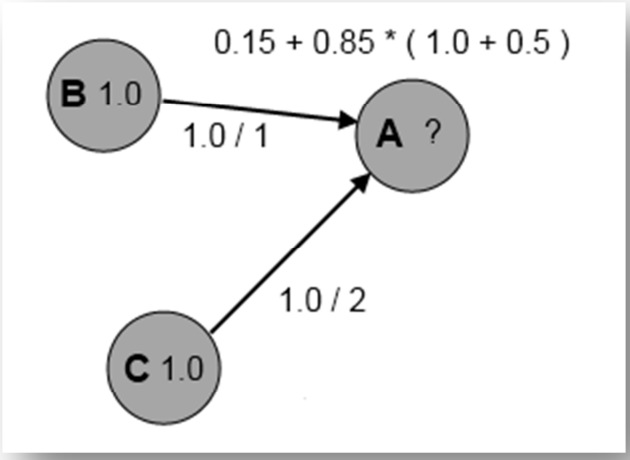


Figure 5.6 PageRank algorithm for a simple network: contributions at first iteration

Note that we omit PageRank(C), because it is a non-existing web page. Broadly, it is common to perform over 10 iterations to obtain more accurate ranks (algorithm is based on previous calculations and will get more accurate ranks after more runs).

At MapReduce level (Haddop), in the mapper phase, each map task matches each outgoing link to the page with its rank and total outgoing links. In the reduce phase, each reduce task simply calculates the new page rank for the pages by applying general formula shown above. In Spark, algorithm follows mainly the same methodology. To sum up, it executes "*map*" and "*groupByKey"* functions to handle data properly and then it calculates contributions for each link through "*join*" and "*map*" functions. At last, by using "*reduceByKey*" and "*mapValues*" functions, it can obtain final ranks for each given link. In this case, it is interesting to understand how Spark can cache datasets in memory to speed up reuse. PageRank algorithm can load just the links in RAM using "*cache*" function on a RDD. Other steps are done by Spark internal core.

Hadoop does not have any official implementation of PageRank algorithm, while Spark comes with all possible implementations (java, python and scala). Again, we have two choices: to develop our own version for Hadoop or find some equivalent code, which must be used by a huge community (in this way the comparison will be valid and reliable). In this case, it does not exist any equivalent code of such algorithm for MapReduce framework, so that we need to develop our own code or find some similar code to make it suitable in order to better perform system performance analysis and benchmarking.

We have found many examples through different code repositories, but any of them follows the methodology used in scala code (basic steps of algorithm). Besides, none of them is fully compatible at input data level, so that we must develop our own version. Here, python is not a good choice, so java has been selected as a programming language given its relationship with scala language. Further explanation about it can be found in previous subchapters (see *WordCount problem* and *TeraSort problem*). The java code has been developed in Eclipse IDE [76] and compiled with required packages for hadoop-mapreduce framework [77] .

We will run both packages following the standard methodology for each one of them. On the one hand, in the Mapreduce case, we will use "hadoop-mapreduce-pagerank.jar" package through "hadoop jar" instruction, which contains our own PageRank class. On the other hand (Spark's case), we will use "spark-submit" script to submit the compiled Spark application (package "spark-examples-1.3.0-cdh5.4.8-hadoop2.6.0-cdh5.4.8.jar") which contains SparkPageRank class. In both cases, YARN will be used as a cluster manager in cluster mode.

5.4 Benchmarking

Since the most reliable metric when we want to compare several algorithms (either sequential or parallel) is the time execution, we focus on it in order to perform system performance analysis and benchmarking of previous problems.

We are going to study time execution from two different points of view; on the one hand, we focus on the middleware tier (YARN), which will provide us an idea of the processing time when we consider the whole involved process (see *Hadoop execution flow*). This metric can help us to determine the behavior of YARN regarding to other resource managers at performance level. On the other hand, we focus on framework-specific processing time; that is, the whole internal processing time on each of processing framework when they are starting a certain job (when the ApplicationMaster process is able to start the first task). Moreover, if applicable, we show aggregate resource allocation metric (MB-seconds, vcore-seconds), which shows the amount of consumed resources by all the tasks from two perspectives: internal memory (RAM) and processing time (CPU). Thus, the first metric is extracted from Resource Manager's Web UI (*http://server:8088*), while specific-processing time at framework level is extracted from both History's roles Web UI (*http://server:19888/jobhistory* for MapReduce and *http://server:18088* for Spark). Regarding to aggregate resource allocation metric, it is also extracted from Resource Manager's Web UI (*http://server:8088*).

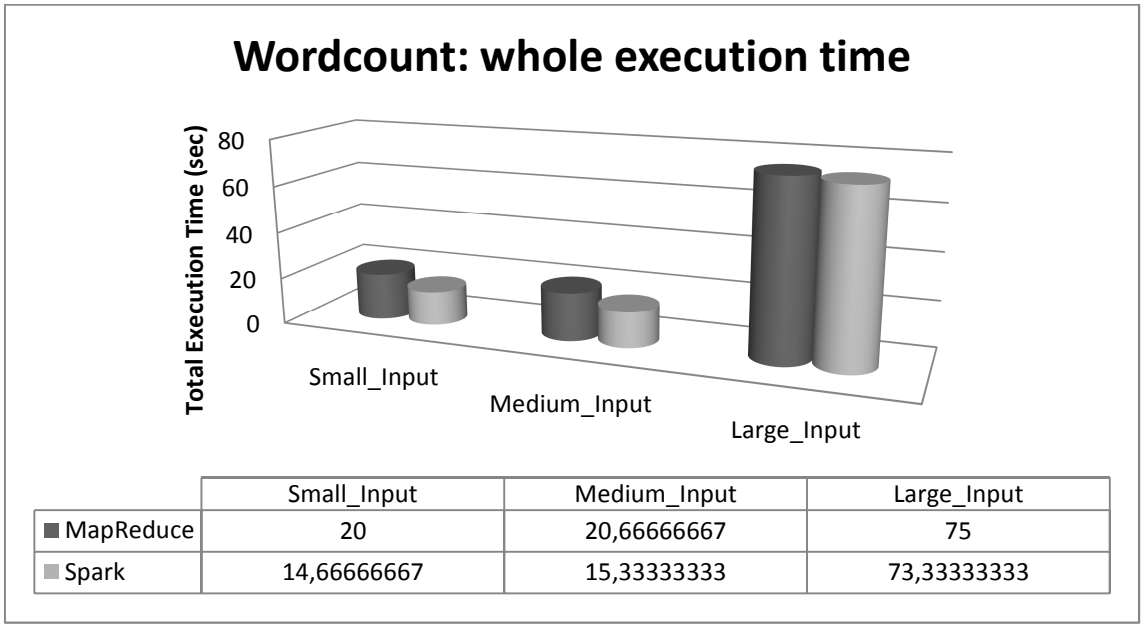
All things considered, we going to show different performance metrics as well as statistical data related to each of the algorithms.

The WordCount algorithm will be the first case to be analyzed. In this case, we perform benchmarking through several ebooks [78]; it is quite reasonable, since the goal of WordCount algorithm is to find the number of occurrences of each word in a given input set (which is typically a text file). The next table shows three different sizes of input data; we specify the length of each file in words, as well as the size of each file.

Small input: *The Outline of Science, Vol. 1 (of 4) by J. Arthur Thomson*	Medium input: *The Notebooks of Leonardo Da Vinci*	Large input: *The Notebooks of Leonardo Da Vinci* (x 95)
809 words (~5KB)	255384 words (~1,36 MB)	24261480 words (~128MB)

Table 5.1 WordCount input data

The figure below shows the whole execution time from YARN's perspective:



	Small_Input	Medium_Input	Large_Input
MapReduce	20	20,66666667	75
Spark	14,66666667	15,33333333	73,33333333

Figure 5.7 WordCount algorithm: whole execution time

In the same way, we show the total execution time at framework level:

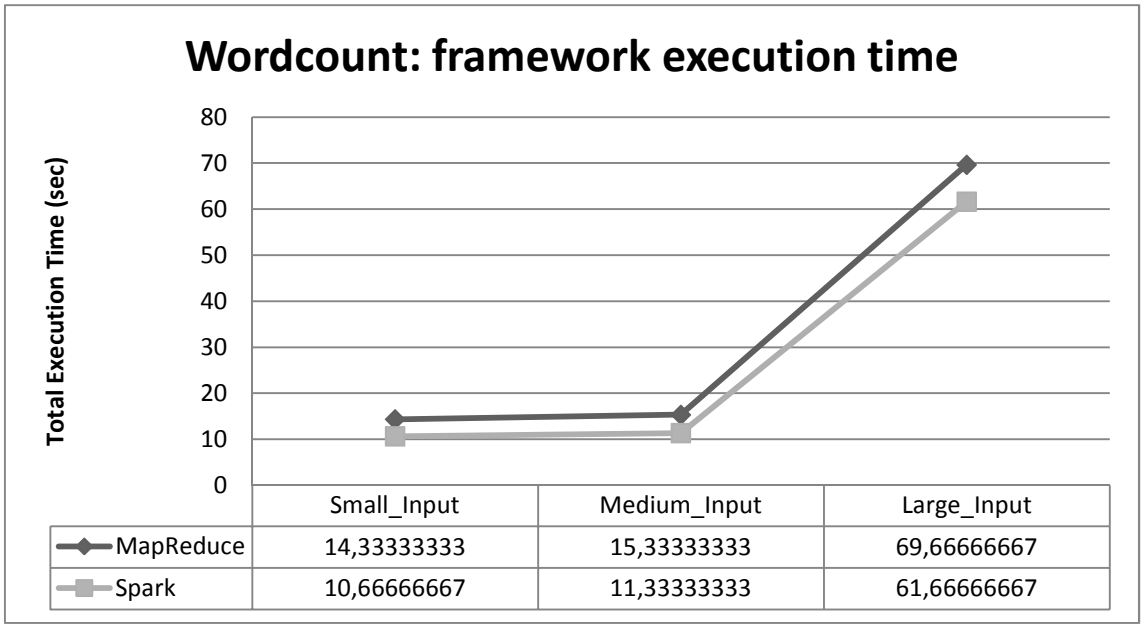


Figure 5.8 WordCount algorithm: framework execution time

As we can see in the figures above, the first conclusion that we can extract is that the execution time required by YARN is greater than framework-specific processing time. Somehow, it is something we could intuit; before a certain NodeManager can start a task, several steps must be performed in order to ensure the entire job properly (driving job submission, allocating resources and so on).

At framework-specific processing time level, we can see how differences between both processing frameworks are minimal; however, Spark overcomes MapReduce (even if we use a more large input data). Both algorithms basically perform the same task and they are designed in the same way. Why is Spark more efficient than MapReduce? The WordCount algorithm is so simple that allow us to understand a few key concepts around Spark (which make the difference):

- Faster task startup time. Spark works at thread level (fork operation for each new task) while MapReduce works at JVM level (brings up a new JVM for each new task).

- Faster shuffles. Each algorithm performs one all-to-all operation (see next case for further details) which means to perform a "*shuffle process*". Spark puts the data on disk only once during shuffles (in the best case) while MapReduce does it twice (by default).

- Caching concept. Spark can cache data into memory (see next case for further details) which can improve performance considerably. Despite MapReduce can cache data through HDFS, this technique is not at the same level that in-memory caching.

Above all, we need to keep in mind that we have developed a simple version of WordCount algorithm (where the goal is simplicity and ease of understanding); that is, we do not use neither Combine [79] function at MapReduce level nor iterators/generators [80] at python level; both concepts can help a lot in terms of computational expensiveness or memory consumption, depending on the task at hand.

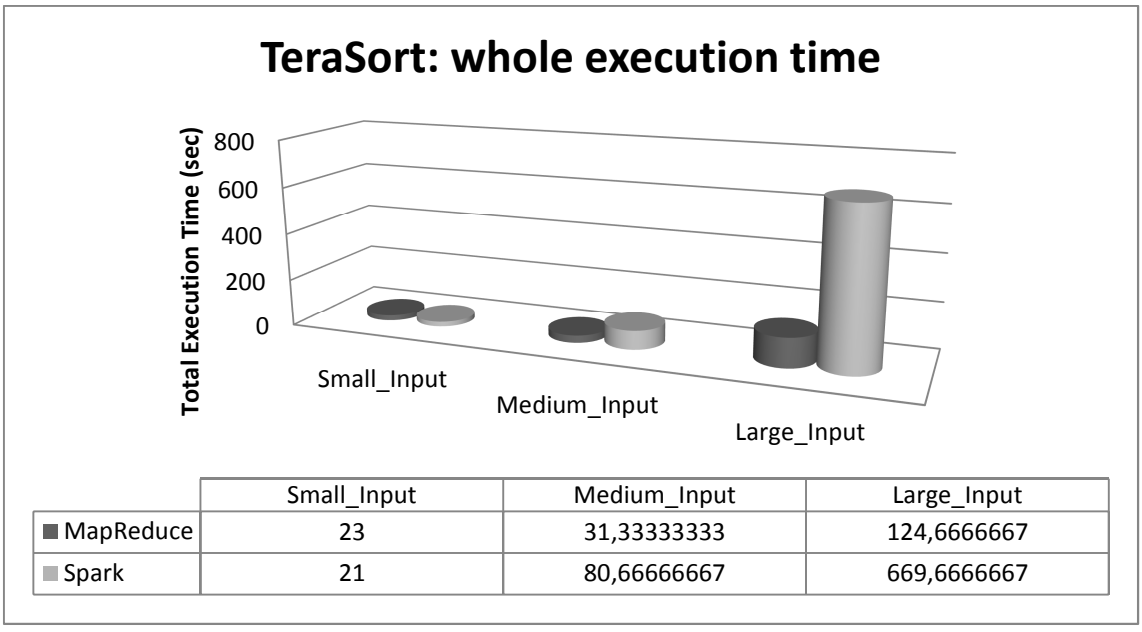
Although we neither talk nor show aggregate resource allocation metric (MB-seconds, vcore-seconds), it is also increased regarding to the input data set (many tasks mean more resource allocation for each of them at time level). Here, Spark tries to set the number of partitions automatically based on our cluster (2 tasks to 2 executors), while Hadoop performs 14 tasks by default (2 mappers and 12 reducers). Having said that, it does not make sense to compare both frameworks from such perspective due to each framework uses different partitioning mode.

It is time to analyze TeraSort algorithm. Again, we show three different sizes of input data. Note that although we do not use 1TB of data (which is the typical case) due to characteristics of our cluster architecture, the selected sizes provide us an idea of the behavior of this algorithm. The table below describes the selected input data sets:

Small input	Medium input	Large input
100 MB	1 GB	10 GB

Table 5.2 TeraSort input data

Next, we show the whole execution time from YARN's perspective:



	Small_Input	Medium_Input	Large_Input
MapReduce	23	31,33333333	124,6666667
Spark	21	80,66666667	669,6666667

Figure 5.9 TeraSort algorithm: whole execution time

In the same way, we show the total execution time at framework level:

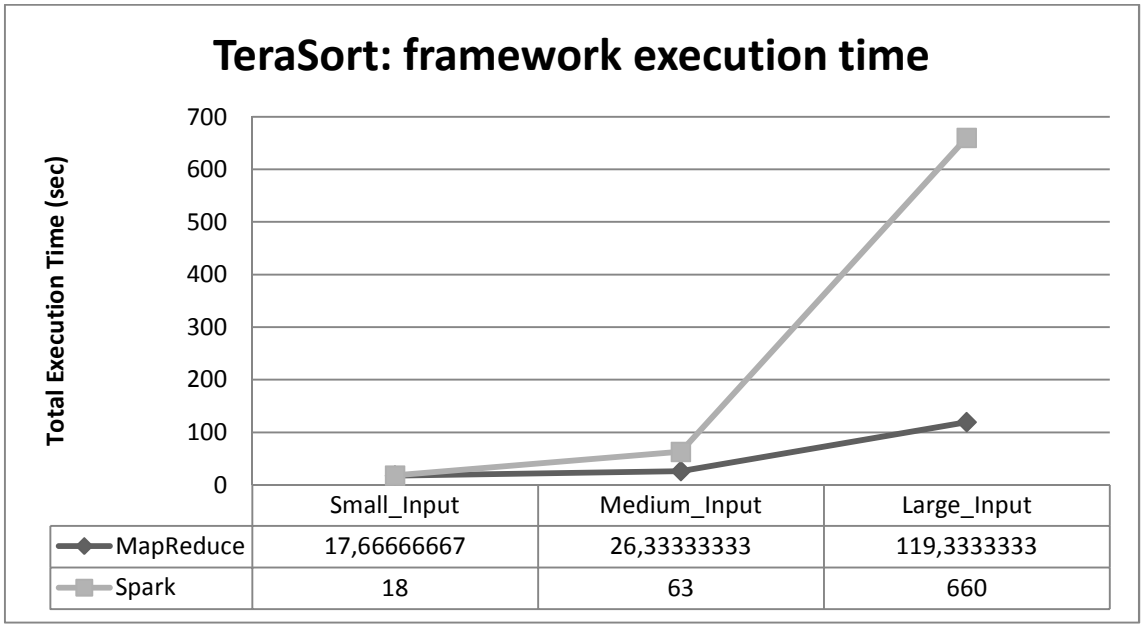


Figure 5.10 TeraSort algorithm: framework execution time

One more time, we can see how the execution time required by YARN is greater than framework-specific processing time; which is again quite reasonable.

At framework-specific processing time level, we can see how Hadoop overcomes Spark according to input data is increased. In other words, when data set is not very large (100 MB in our case), Hadoop's execution time is smaller than Spark's execution time; otherwise, Spark takes a long time to process it. To know the why of it, it is mandatory to understand Spark architecture in a deeper way, as well as Spark main programming abstractions (RDD and DAG). As we can imagine, this kind of insight is beyond the scope of this project due to its complexity; however, we going to try to explain a few details in order to make it understandable.

In this point, we need to understand how Spark handles executors (JVM processes) and how it configures and uses their heap memory. Next, we show the diagram of Spark memory allocation inside of the JVM heap [81]:

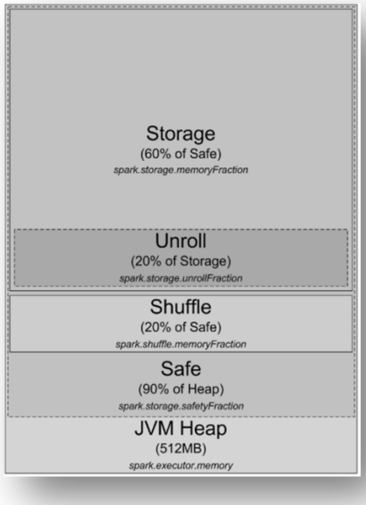


Figure 5.11 Spark JVM heap

By default, Spark starts with 512 MB *JVM Heap* (for Driver Program) and 1024 MB for Executors (all of this is totally configurable through spark default settings or command line arguments of the different executions). Though, in order to avoid “OutOfMemory” errors, Sparks only allows utilizing around 90% (*Safe zone*). Moreover, Spark allows storing some data in memory (it utilizes this memory for its LRU cache [82]); thus, some amount of memory is reserved for the caching of the data that are being processed (around 60%, *Storage zone*). Within of it, we can see a certain amount of memory (around 20%, *Unroll zone*), which is the amount of RAM that is allowed to be utilized by "unroll" process. The “unroll” process unrolls blocks of data into the memory, so they can be used properly. Basically, data can be stored both serialized (serialized caching) and deserialized (raw caching) form. Since data in serialized form cannot be used directly, they must be "unrolled" previously (so this is the RAM that is used for "unrolling"). Each one has different advantages as well as drawbacks; but one key concept that we need to take into account is that raw caching (deserialized data) consumes a lot of memory (one RDD can grow up to 4 times).

The *Shuffle zone* (around 20%, although it exactly takes around 16%) deserves special mention. Spark uses this memory (also known as “*shuffle buffer*”) for "shuffling" process. Broadly speaking, several operations such as sorting, data aggregation or cogroups (among other) involve shuffle processes. Specifically; when we perform different transformations such as *groupByKey* and *reduceByKey* (which have wide dependencies), Spark must execute a shuffle, which transfers data around the cluster and results in a new stage with a new set of partitions. As we can imagine, shuffle incurs heavy disk and network I/O, so it results in an expensive operation, which should be avoided when possible. At any given time, if we do not have enough memory to store the whole “map” output, Spark might need to spill intermediate data to the disk.

Now we have more knowledge about Spark architecture, we can explain the reason of why Spark takes long processing time for both medium and large input data. If we take a look at source code, when we perform *repartitionAndSortWithinPartitions* operation [83], Spark splits data according to the given partitioner and, within each resulting partition, sort records by their keys (*sortByKey*). Given that the values for the same key must be on the same machine in order to sort data, data must be transferred, which implies "shuffling" process explained above. In our case, “map” output does not fit into shuffle buffer, so Spark needs to spill some intermediate data to disk (that is, twice). All of this explains poor performance when data set is considerably large. According to some studies, most performance, scalability, and reliability issues observed in production Spark deployments occur within the shuffle process.

In this case, it makes sense to compare both processing frameworks from aggregate resource allocation metric perspective, due to both frameworks have the same partitioning level. In any case, the number of reducer tasks for MapReduce framework will be 12 by default (so mappers will be the difference between total tasks and reducers); Spark does not distinguish roles, so the amount of tasks (among all the stages per job) will be the given number. The next table shows the total number of tasks for a given input data:

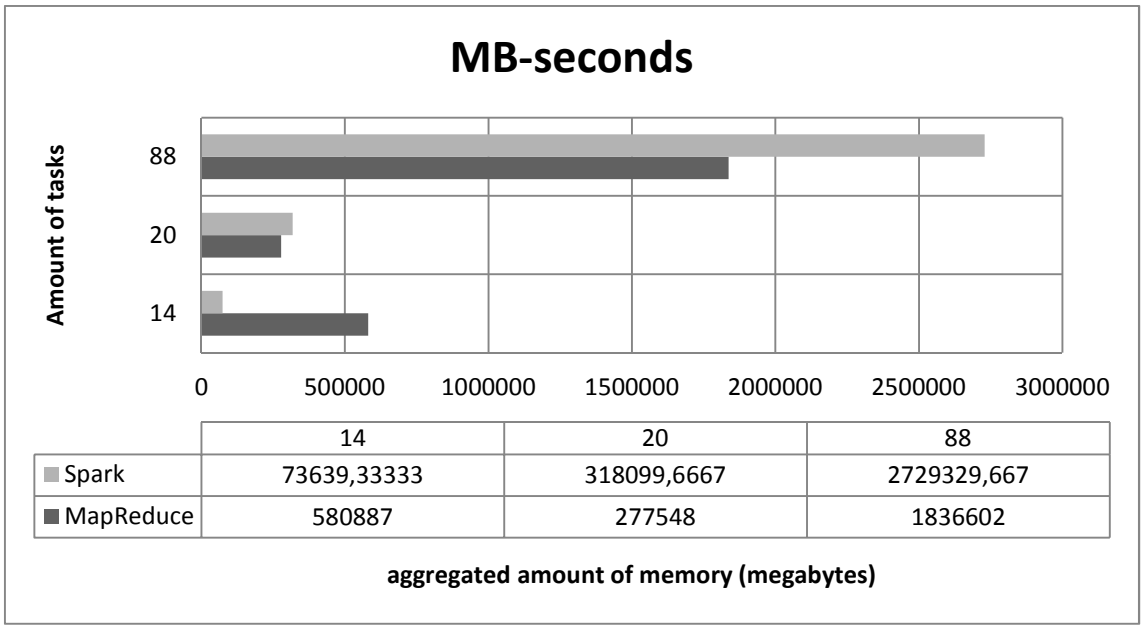
Small input	Medium input:	Large input:
14 tasks	20 tasks	88 tasks

Table 5.3 TeraSort amount of tasks per job

If we keep in mind that:

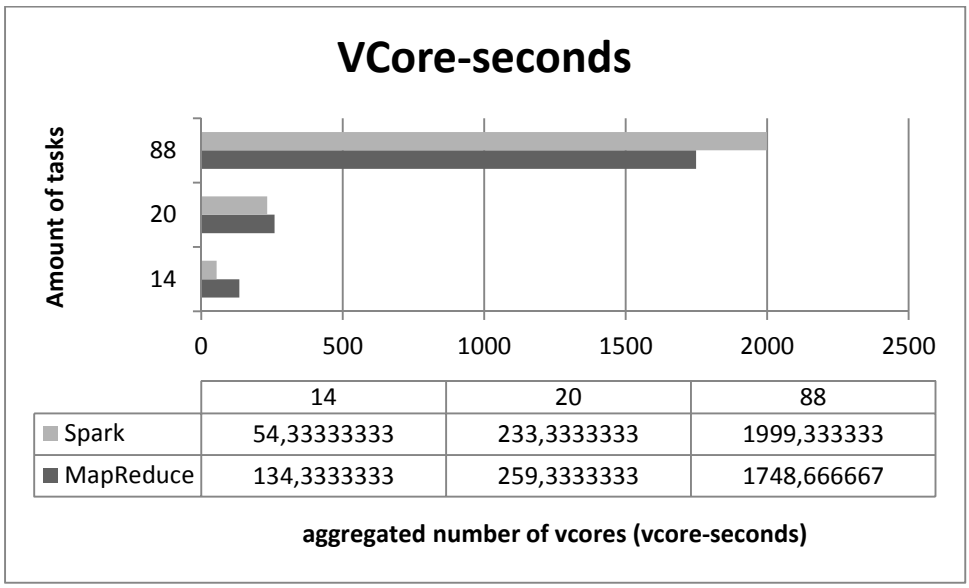
- **MB-seconds**: The aggregated amount of memory (in megabytes) the application has allocated times the number of seconds the application has been running.
- **Vcore-seconds**: The aggregated number of vcores that the application has allocated times the number of seconds the application has been running.
- **Active nodes in our cluster**: 3 (worker nodes).
- **Memory Total of our cluster**: 20,66 GB (maximum memory per node: 7052 MB).
- **VCores Total of our cluster**: 24 (maximum cores per node: 8).
- **Maximum number of containers per node**: ~7 (minimum number between memory bounds and cpu bounds).

We can show the next figures:



	14	20	88
Spark	73639,33333	318099,6667	2729329,667
MapReduce	580887	277548	1836602

Figure 5.12 TeraSort MB-seconds



	14	20	88
Spark	54,33333333	233,3333333	1999,333333
MapReduce	134,3333333	259,3333333	1748,666667

Figure 5.13 TeraSort VCore-seconds

In this point, we can conclude that:

- For a small data set, we can see how Spark is more efficient than MapReduce in both metrics. That is, when data set is relatively small (smaller than block size in this case), data set fits perfectly in memory and all tasks can handle data in an efficient way. Thus, each YARN container takes less time (on an average) to execute a certain task than in MapReduce engine.

- When we have a medium data set, Spark starts suffering known issues when all data set almost does not fit in memory, so it needs to spill (performing both data compression and serialization) some data to disk. We need to understand that all shuffled data per reduce task must fit into memory; it happens when task involves an all-to-all operation (*sortByKey* in our case). However, it is interesting to see how from processing time point of view, Spark is still more efficient than MapReduce official implementation (each container needs, on an average, less time to execute a certain task).

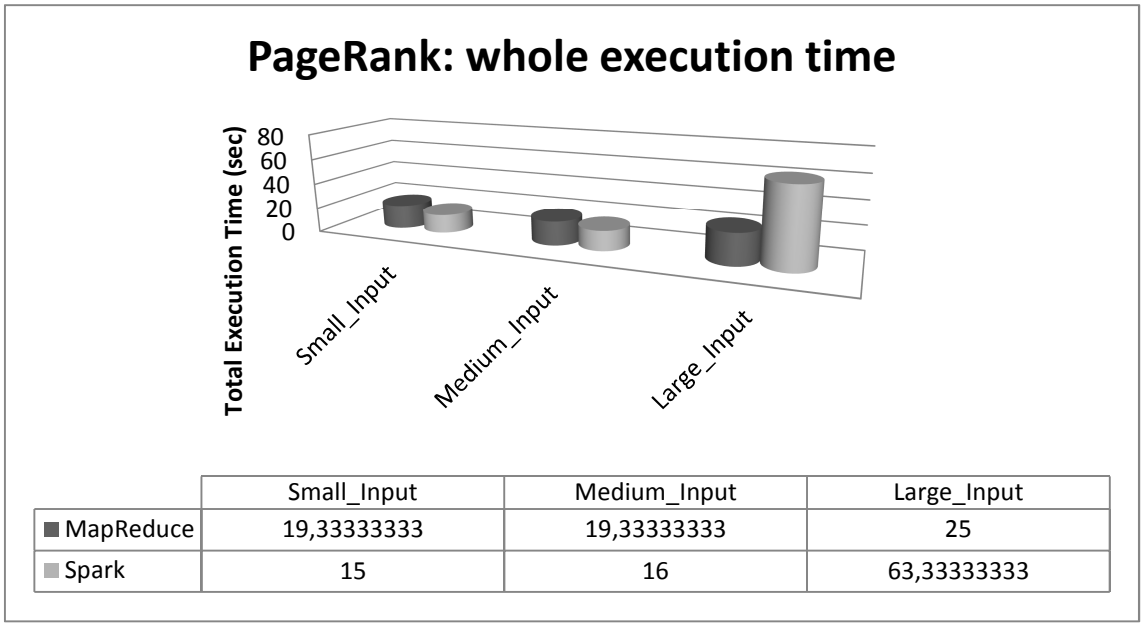
- A large data set produces a reverse trend; like previous case, Spark needs to spill a lot of data to disk. Moreover, it causes a long processing time (each container takes, on an average, more time to execute a certain task than in MapReduce engine). Here, Spark clearly does not overcome MapReduce official implementation.

Finally, we have to analyze PageRank algorithm. Again, we show three different sizes of input data; in this case, we specify, for each given file, the amount of nodes as well as the amount of edges (which represent the whole simulated network). Last input data (large data set) deserves further consideration, since we are using a real representation of Amazon network (co-purchasing network from March 2003). The table below summarizes such information:

Small input:	Medium input:	Large input:
86 nodes, 430 edges (~8KB)	709 nodes, 3545 edges (~60KB)	262111 nodes, 1234877edges (~16,4MB)

Table 5.4 PageRank input data

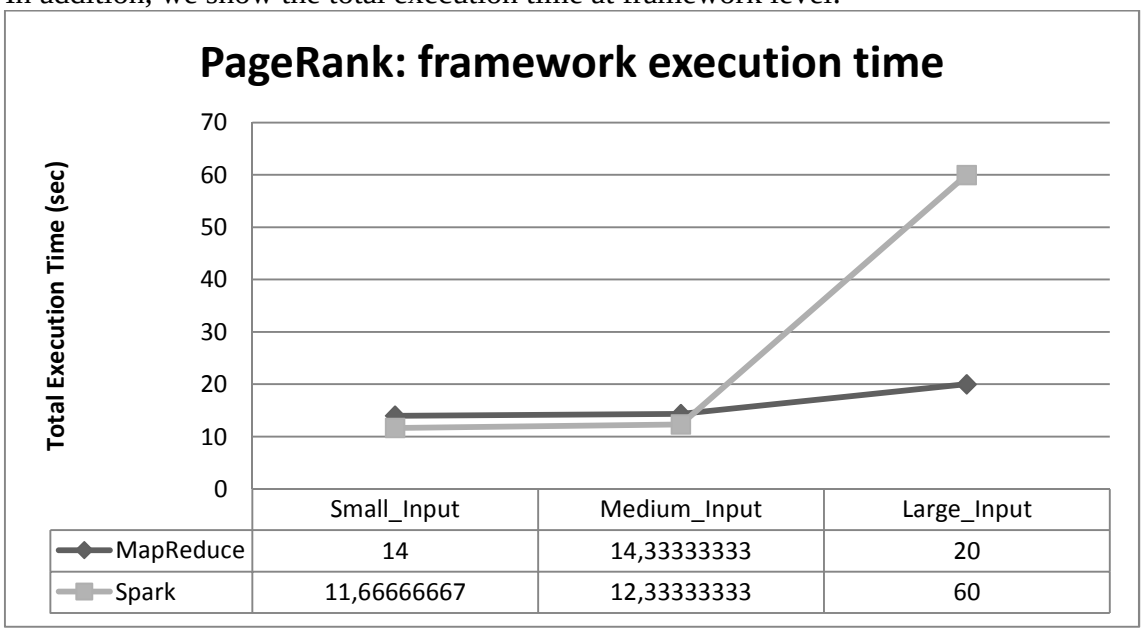
Such as we did before, we show again the whole execution time from YARN's perspective:



	Small_Input	Medium_Input	Large_Input
MapReduce	19,33333333	19,33333333	25
Spark	15	16	63,33333333

Figure 5.14 PageRank algorithm: whole execution time

In addition, we show the total execution time at framework level:



	Small_Input	Medium_Input	Large_Input
MapReduce	14	14,33333333	20
Spark	11,66666667	12,33333333	60

Figure 5.15 PageRank algorithm: framework execution time

One more time, we can see how the execution time required by YARN is greater than framework-specific processing time; which is quite reasonable again.

At framework-specific processing time level, we can see how Spark overcomes Hadoop when input data size is not very large. However, when we work with a large input (a lot of records of RDD key/value pairs in our case), Spark reduces its performance. The reason was mentioned earlier; Spark must spill a certain amount of data to disk (data does not fit into the shuffle buffer, so it needs to spill data to disk twice). Despite Spark uses *cache* operation in order to cache data into memory (which might lead to a great performance), the point is that PageRank (scala algorithm) utilizes many all-to-all operations (*groupByKey*, *join* and *reduceByKey*), which implies several stages with multiple shuffle operations among each of them. As we explained above, each shuffle process causes a great impact on the performance. The scala source code is a good example of working with RDD key/value pairs; it allows us to get input data into a key/value format and provides special operations (such as aggregation or transformations) on RDDs containing key/value pairs. However, when we handle a lot of records, it starts suffering a great impact at performance level (long processing time, high memory demands) without forgetting multiple shuffle processes due to its design (which incurs on disk I/O and high network latency). Thus, we can confirm that our java implementation of PageRank is more efficient than Spark's implementation when we handle large input data set (note that Hadoop/MapReduce is specially designed to treat with huge input data set).

In this case, it does not make sense to talk about show aggregate resource allocation metric (MB-seconds, vcore-seconds), since each processing framework utilizes different partition level: Hadoop uses 14 tasks by default (2 mappers and 12 reducers) while Spark uses 4 tasks within 4 stages.

6 Conclusions

In this BS Final Project we have introduced us to Big Data field. Although it has been only "the tip of the iceberg", it has provided us an excellent insight of what is and what means the famous word “*Big Data*”.

In the first part of this BS Final Project, through several examples, we have provided a definition of the Big Data term. Next, we have also described challenges and opportunities arising from Big Data. In addition, we have dissected the Big Data lifecycle management from two different perspectives as well as describing the Big Data characteristics (Gartner's model). In this point, we have focused on velocity as key feature, which lead us to different processing models (Batch, Real-time and Stream processing).

In the second part of this BS Final Project, we have studied and discussed the two most well-known processing frameworks for Big Data applications. Mostly, we have focused on the two major current proposed large scale distributed processing solutions: Apache Hadoop and Apache Spark. We have seen how each one may be categorized according to different processing models; while Hadoop follows Batch processing model (MapReduce paradigm), Spark is commonly categorized into Real-time processing model (In-Memory computing paradigm). Moreover, we have described each processing framework from different perspectives (architectonic solution, basic design patterns as well as several key features).

In the third part of this BS Final Project, we have presented our Big Data architecture solution. We have designed the whole architecture taking into account both low-level and high-level requirements. We have shown a step-by-step of the entire procedure installation; starting from installing and tuning a GNU/Linux environment as well as the use of Cloudera (Cloudera open-source Apache Hadoop distribution, CDH) as infrastructure solution.

The last part of this BS Final Project covers system performance analysis and benchmarking. We have described the methodology that we have followed in order to perform system performance analysis and benchmarking of both large scale distributed processing solutions. In the first part, we have presented three different problems, where each one performs different tasks and follows a different programming paradigm. In some cases, it has been necessary to adapt some algorithm in order to make it suitable to our architecture; in other cases, it has been necessary to develop our own algorithm in order to make a fair comparison. In the second part, we have shown different performance metrics as well as statistical data, which have allowed us to understand, in a deeper way, both processing solutions.

Currently, Apache Spark is becoming in the new trend in the Big Data field. The figure below shows search results (Google Trends [84] for both Hadoop and Spark) for the last two years:

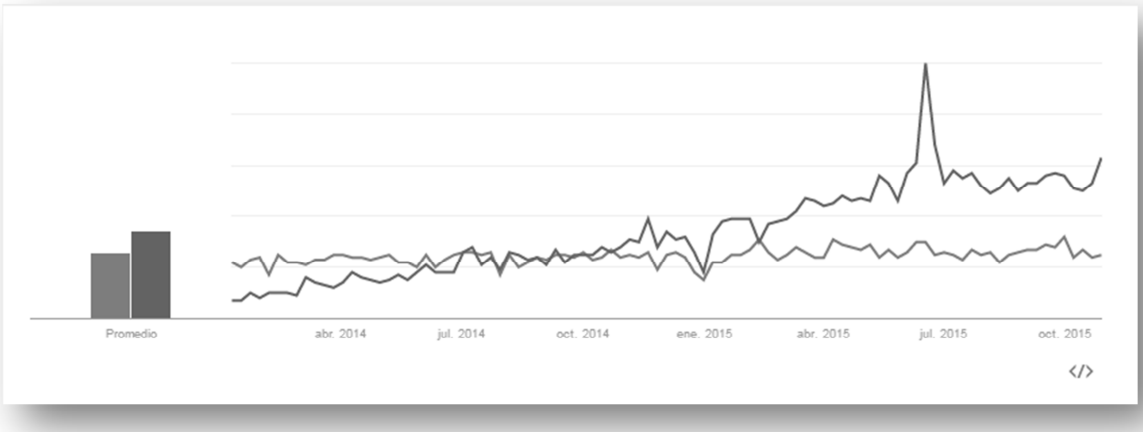


Figure 6.1 Google Trends: Haddop vs Spark (January 2014 - October 2015)

Clearly, Spark (top line) is becoming more and more popular among the end customers, so they are performing multiple web searches related to Spark. But, does it mean that Spark is going to replace Hadoop definitely? Is Hadoop/MapReduce an obsolete processing framework? Thanks to this BS Final Project, we have proved how this myths and rumors are completely false.

Firstly: Hadoop is not just a processing framework. As we have seen before, Hadoop is an ecosystem or collection of additional software packages that can be installed on top of or alongside. Thus, Spark can run on top of Hadoop, benefiting from Hadoop cluster manager (YARN) and underlying storage (HDFS) (although it can also run completely separately from Hadoop, maybe it is not the best option nowadays). Secondly, Hadoop is a more mature platform. MapReduce has proven its experience for the last years, with multiple successful cases. Furthermore, its integration with third-party software is better in comparison with Spark: we have seen how Cloudera's behavior is better when we perform MapReduce tasks instead of Spark jobs (we achieve more and better information about final results). Lastly, we have seen how Hadoop achieves better results when we handle large input data sets (note that Hadoop/MapReduce is specially designed to treat with huge input data set.

However, Hadoop also has its own weaknesses as well as drawbacks. Firstly: if we take a look beyond, we realize that the MapReduce programing model is not a new thing; in other words, the main concept takes a lot of time being implemented following different strategies (divide&conquer strategy or reduction clauses implemented by several parallel languages such as OpenMP/MPI). Thus, in many cases, maybe we do not need Hadoop in order to achieve our goals (note that any algorithm must be adapted into MapReduce pattern). Secondly, we have seen how in equal conditions (identical algorithm design), Spark overcomes Hadoop achieving better execution times. Lastly, we have verified how Hadoop is suitable for one-pass computation but maybe it is not the best option for cases that require multi-pass computations. Due to its data processing workflow, data between each step must be stored in the Distributed File System (HDFS) before the next step can begin, which may impact in the overall performance (Spark can achieve its goals by putting the data on disk only once during shuffles, in the best cases).

So, what about Spark? Like Hadoop, Spark also has its own strengths as well as drawbacks. Firstly: Spark provides a rich API (scala, python, java, R) which make it easy to develop (you can choose your favourite programming language), while Hadoop only supports java "officially" (as we have verified, it is also possible to develop through Hadoop Streaming and Hadoop Pipes). Secondly, Spark provides faster task startup time (by using threads operations instead of bringing up a new JVM). Lastly, we have seen how Spark can cache a certain amount of data, which is without any doubt, one of its key concepts (without forgetting RDD's and DAG pipeline). It is true that Hadoop can cache some amount of data through underlying storage (HDFS) but in general Spark's cache engine is quite good (in fact, nothing revolutionary new, since memory access is always faster than disk access).

As we mentioned above, Spark is not an exception, so it also has several weaknesses as well as drawbacks. Firstly: working with Resilient Distributed Dataset (RDD) is not so simple. For those who are familiar with OOP languages, understanding and managing RDD's can be a bit difficult. Moreover, despite its multiple benefits, we cannot forget that RDD structures consume a lot of storage memory (raw caching, otherwise they cannot be used directly). In fact, we have observed how Spark does not work well when we handle a large input data set (note that Spark is specially designed to achieve minimal processing times instead of handling with huge input data set). Secondly, Spark is not an in-memory technology really; in other words, Spark cannot persist data in memory and process it entirely. Everything Spark can do is to cache data (into a certain amount of cache memory with the LRU eviction rules), which is not equal to the “persistence” concept. Cached data can be easily dropped and recomputed later, based on the other data available in the source persistent store available through connector (that is what really Spark utilizes: pluggable logical connectors for different persistent storage systems like HDFS).

Moreover, if Spark does not have enough memory to store the data, it will spill intermediate data to the disk. Thus, if we keep in mind the "*shuffle process* "explained above, Spark inevitably performs operations into the disk (unless it does not need to shuffle data, which is quite unusual). Lastly, we can conclude that the next assertion "*Spark performs 10x-100x faster than Hadoop*" is in most cases false. Through our system performance analysis and benchmarking of the different problems, we have observed how it never happens. Not even in a so simple case (see *WordCount problem*) where both algorithms are almost identical (and both perform only one shuffle process) we do not achieve this huge performance. Spark performs "*10x-100x faster operations*" when it works with machine learning algorithms. That is, when this kind of algorithms are repeatedly iterating over the same dataset many times, Spark can cache dataset into the memory (it only needs to read it when data is accessed for the first time, after it only needs to read it from the cache memory). Anyway, we are talking about an ideal situation (pure machine learning algorithm, entire dataset fits perfectly into memory and so on).

In summary, Spark is a viable alternative to Hadoop/MapReduce in a range of circumstances, but in no case it is a replacement for Hadoop; we should think in Spark as a great companion to a modern Hadoop cluster deployment. Thus, when someone asks you the next question: Which is better, Hadoop or Spark? The correct answer would be: Neither. It is simply a wrong question. Our research has demonstrated how the best choice between each processing framework will depend on many factors. From my point of view, the most important are as follows: typology of the problem, designing efficient algorithms as well as performing several tuning options (including HDFS, YARN, MapReduce tasks and Spark jobs). Only if we keep in mind all these factors, we can achieve our goals.

6.1 Future Work

On the one hand, keeping in mind that we have performed our comparative tests with the default values, as a part of the future work it would be interesting to perform several tuning options in order to evaluate the set of algorithms. Taking into account the typology of the algorithm and the input data set, we can optimize the overall execution workflow as well as improving the performance of the whole system. Tuning options must include the underlying storage (HDFS), the resource manager (YARN) and the processing layer (MapReduce tasks or Spark jobs). In addition, if we want to achieve excellent results, we must understand the internals concepts of each programming model (both Mapreduce and RDD/DAG). After all, we are treating with parallel algorithms, so a good algorithm design is key concept in order to improve global performance (instead of increasing hardware resources in many cases). Therefore, future evaluations will have to consider both points of view.

On the other hand, due to new trends and requirements, new processing frameworks are becoming more and more popular nowadays. As we commented previously (see *Stream processing*) several processing frameworks are being designed in order to continuously process and handle on the live stream data. Good examples are Apache Storm from Twitter and Apache S4 from Yahoo. Moreover, we cannot obviate hybrid computation models (such as micro-batching), which mixes both batch processing and stream processing techniques and will be suitable in specific cases. Spark Streaming is the most well-known solution that follows this hybrid methodology. Therefore, as a part of the future work, it would be interesting to begin taking a look at these new approaches, which will mark new trends in the future undoubtedly.

Bibliography

[1] Talend, "How Big Is Big Data Adoption?," 2013. [Online]. Available: https://www.talend.com/.

[2] Wikipedia, "Database," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Database.

[3] Wikipedia, "Virtualization," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Virtualization.

[4] Wikipedia, "Cloud computing," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Cloud_computing.

[5] "3D Data Management: Controlling Data Volume, Velocity, and Variety," 2001. [Online]. Available: http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-V and-Variety.pdf.

[6] Wikipedia, "Batch processing," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Batch_processing.

[7] Wikipedia, "Real-time computing," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Real-time_comput

[8] Wikipedia, "Stream processing," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Stream_processing.

[9] A. Hadoop, "Apache Hadoop," 2015. [Online]. Available: https://hadoop.apache.org/.

[10] A. Spark, "Apache Spark," 2015. [Online]. Available: http://spark.apache.org/.

[11] Wikipedia, "MapReduce," 2015. [Online]. Available: https://en.wikipedia.org/wiki/MapReduce.

[12] B. University of California, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," 2012. [Online]. Available: https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf.

[13] Cloudera, "Cloudera," 2015. [Online]. Available: http://www.cloudera.com/content/www/en-us.html.

[14] Cloudera, "Cloudera Manager," 2015. [Online]. Available: https://www.cloudera.com/content/www/en-us/products/cloudera-manager.html.

[15] T. C. Project, "The CentOS Project," 2015. [Online]. Available: https://www.centos.org/.

[16] SDSS, "SDSS," 2015. [Online]. Available: http://www.sdss.org/.

[17] T. L. S. S. Telescope, "The Large Synoptic Survey Telescope," 2015. [Online]. Available: http://www.lsst.org/.

[18] Walmart, "Walmart," 2015. [Online]. Available: http://www.walmart.com/.

[19] Facebook, "Facebook," 2015. [Online]. Available: https://www.facebook.com/.

[20] G. Research, "Gnome Research," 2015. [Online]. Available: http://genome.cshlp.org/.

[21] Wikipedia, "Internet of Things," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Internet_of_Things.

[22] Wikipedia, "Clustered file system," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Clustered_file_system#Distributed_file_systems.

[23] Google, "The Google File System," 2003. [Online]. Available: http://static.googleusercontent.com/media/research.google.com/es//archive/gfs-sosp2003.pdf.

[24] A. Hadoop, "HDFS Architecture Guide," 2015. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[25] Github, "TFS (Taobao File System)," 2015. [Online]. Available: https://github.com/alibaba/tfs.

[26] Wikipedia, "NoSQL," 2015. [Online]. Available: https://en.wikipedia.org/wiki/NoSQL.

[27] Wikipedia, "Dremel (software)," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Dremel_(software).

[28] Cloudera, "Apache Impala," 2015. [Online]. Available: http://www.cloudera.com/content/www/en-us/products/apache-hadoop/impala.html.

[29] A. Drill, "Apache Drill," 2015. [Online]. Available: https://drill.apache.org/.

[30] S. SQL, "Spark SQL," 2015. [Online]. Available: https://spark.apache.org/sql/.

[31] Hortonworks, "Stinger.next: Enterprise SQL at Hadoop Scale," 2015. [Online]. Available: http://hortonworks.com/innovation/stinger/.

[32] A. Hive, "Apache Hive," 2015. [Online]. Available: https://hive.apache.org/.

[33] A. Storm, "Apache Storm," 2015. [Online]. Available: http://storm.apache.org/.

[34] A. S4, "S4: Distributed Stream Computing Platform," 2015. [Online]. Available: http://incubator.apache.org/s4

[35] Hortonworks, "Apache Storm Design Pattern—Micro Batching," 2015. [Online]. Available: http://hortonworks.com/blog/apache-storm-design-pattern-micro-batching/.

[36] S. Streaming, "Spark Streaming," 2015. [Online]. Available: http://spark.apache.org/streaming/.

[37] A. H. N. M. (YARN), "Apache Hadoop NextGen MapReduce (YARN)," 2015. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[38] A. Ambari, "Apache Ambari," 2015. [Online]. Available: http://ambari.apache.org/.

[39] A. Avro, "Apache Avro," 2015. [Online]. Available: http://avro.apache.org/.

[40] A. Cassandra, "Apache Cassandra," 2015. [Online]. Available: http://cassandra.apache.org/.

[41] Chukwa, "Chukwa," 2015. [Online]. Available: http://chukwa.apache.org/.

[42] A. HBase, "Apache HBase," 2015. [Online]. Available: http://hbase.apache.org/.

[43] A. Mahout, "Apache Mahout," 2015. [Online]. Available: http://mahout.apache.org/.

[44] A. Pig, "Apache Pig," 2015. [Online]. Available: http://pig.apache.org/.

[45] A. Tez, "Apache Tez," 2015. [Online]. Available: http://tez.apache.org/.

[46] A. ZooKeeper, "Apache ZooKeeper," 2015. [Online]. Available: http://zookeeper.apache.org/.

[47] H. Streaming, "Hadoop Streaming," 2015. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/streami

[48] P. org.apache.hadoop.mapred.pipes, "Package org.apache.hadoop.mapred.pipes," 2015. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/mapred/pipes/package-summary.html.

[49] Google, "MapReduce: Simplified Data Processing on Large Clusters," 2004. [Online]. Available: http://static.googleusercontent.com/media/research.google.com/es//archive/mapreduce-osdi04.pdf.

[50] T. White, Hadoop- The Definitive Guide, 4th Edition, 2015.

[51] Cloudera, "The Small Files Problem," 2015. [Online]. Available: http://blog.cloudera.com/blog/2009/02/the-sm problem/.

[52] A. S. Foundation, "Apache Software Foundation," 2015. [Online]. Available: http://www.apache.org/.

[53] A. Spark, "Spark Programming Guide," 2015. [Online]. Available: http://spark.apache.org/docs/latest/program guide.html.

[54] Scala, "Scala - Object-Oriented Meets Functional," 2015. [Online]. Available: http://www.scala-lang.org/.

[55] A. Spark, "Running Spark on Mesos," 2015. [Online]. Available: http://spark.apache.org/docs/latest/running-o mesos.html.

[56] A. Spark, "Spark Standalone Mode," 2015. [Online]. Available: http://spark.apache.org/docs/latest/spark-standalone.html.

[57] A. Spark, "Running Spark on EC2," 2015. [Online]. Available: http://spark.apache.org/docs/latest/ec2-scripts.h

[58] A. Spark, "Machine Learning Library (MLlib) Guide," 2015. [Online]. Available: http://spark.apache.org/docs/latest/mllib-guide.html.

[59] A. Spark, "GraphX Programming Guide," 2015. [Online]. Available: http://spark.apache.org/docs/latest/graphx programming-guide.html.

[60] E. U. B. AMPLab, "GraphX: A Resilient Distributed Graph System on Spark," 2013. [Online]. Available: https://amplab.cs.berkeley.edu/wp-content/uploads/2013/05/grades-graphx_with_fonts.pdf.

[61] OpenStack, "Swift," 2015. [Online]. Available: http://docs.openstack.org/developer/swift/.

[62] A. w. services, "Amazon S3," 2015. [Online]. Available: https://aws.amazon.com/es/s3/.

[63] Cloudera, "Kudu: New Apache Hadoop Storage for Fast Analytics on Fast Data," 2015. [Online]. Available: https://blog.cloudera.com/blog/2015/09/kudu-new-apache-hadoop-storage-for-fast-analytics-on-fast-data/.

[64] U. Berkeley, "Optimizing Shuffle Performance in Spark," 2015. [Online]. Available: http://www.cs.berkeley.edu/~kubitron/courses/cs262a-F13/projects/reports/project16_report.pdf.

[65] Cloudera, "Cloudera Installation and Upgrade," 2015. [Online]. Available: http://www.cloudera.com/content/us/documentation/enterprise/latest/topics/installation.html.

[66] Cloudera, "Running Spark Applications on YARN," 2015. [Online]. Available: http://www.cloudera.com/content/www/en-us/documentation/enterprise/latest/topics/cdh_ig_running_spark_on_yarn.html?scroll=concept_us1_hly_ds
.

[67] Yahoo, "Apache Hadoop Wins Terabyte Sort Benchmark," 2008. [Online]. Available: https://developer.yahoo.com/blogs/hadoop/apache-hadoop-wins-terabyte-sort-benchmark-408.html .

[68] Yahoo, "Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds," 2009. [Online]. Available: https://developer.yahoo.com/blogs/hadoop/hadoop-sorts-petabyte-16-25-hours-terabyte-62-422.html.

[69] Databricks, "Databricks," 2015. [Online]. Available: https://databricks.com/ .

[70] G. -. E. Higgs, "TeraSort benchmark for Spark," 2015. [Online]. Available: https://github.com/ehiggs/spark-tera

[71] A. Maven, "Apache Maven," 2015. [Online]. Available: https://maven.apache.org/ .

[72] A. Spark, "Spark Programming Guide - Resilient Distributed Datasets (RDDs)," 2015. [Online]. Available: http://spark.apache.org/docs/latest/programming-guide.html#resilient-distributed-datasets-rdds.

[73] G. -. EastCircle, "Run TeraSort using Spark on Yarn," 2015. [Online]. Available: https://github.com/eastcirclek/terasort/blob/master/src/main/scala/eastcircle/terasort/SparkTeraSort.scala.

[74] Wikipedia, "Larry Page," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Larry_Page .

[75] Wikipedia, "PageRank," 2015. [Online]. Available: https://en.wikipedia.org/wiki/PageRank .

[76] Eclipse, "Eclipse," 2015. [Online]. Available: https://eclipse.org/home/index.php.

[77] A. Hadoop, "MapReduce Tutorial," 2015. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[78] P. Gutenberg, "Free ebooks by Project Gutenberg," 2015. [Online]. Available: https://www.gutenberg.org/.

[79] A. Hadoop, "How Map and Reduce operations are actually carried out," 2015. [Online]. Available: https://wiki.apache.org/hadoop/HadoopMapReduce.

[80] IBM, "Charming Python: Iterators and simple generators," 2015. [Online]. Available: http://www.ibm.com/developerworks/library/l-pycon/ .

[81] A. Spark, "Spark Configuration," 2015. [Online]. Available: http://spark.apache.org/docs/latest/configuration.

[82] Wikipedia, "Cache algorithms," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Cache_algorithms.

[83] A. Spark, "Class OrderedRDDFunctions<K,V,P extends scala.Product2<K,V>>," 2015. [Online]. Available: https://spark.apache.org/docs/1.3.0/api/java/org/apache/spark/rdd/OrderedRDDFunctions.html.

[84] Google, "Google Trends," 2015. [Online]. Available: https://www.google.es/trends/.