SPARK: Concept Review

Rocha Ana, Rodrigues Artur, Morelli Jean, Correia Ricardo, Cruz Rúben

Abstract— The motivation for the writing of this paper was to research and explain Apache spark, a unified analytics engine for big data processing developed by the Apache Software Foundation, exposing its strength as a unified platform for a variety of computing tasks. This paper will introduce concepts like Cloud, Grid and parallel computing, as well as a brief dive into the architecture of Spark. Another aspect that this paper will explore is Spark's programing model and main features of its primary libraries such as Spark SQL, Spark Streaming, GraphX and MLlib, some examples of applications of spark will be introduced and it will be presented a comparison of Spark against Apache Hadoop.

Index Terms— Parallel systems, Distributed Systems

1 Introduction

Parallel and distributed programming has been omnipresent in computation for some time now, it's actually hard to find any desktop, laptop or server that does not have a multi-core processor and many of the services offered nowadays are supported by distributed systems. [1]

Cloud Computing is a next big step of field computing and storage and is hinting at a future in which we won't compute on local computers, but on centralized facilities operated by third-party compute and storage utilities. A Cloud infrastructure can be utilized internally by a company or exposed to the public as utility computing. [4]

Apache spark is a unified engine for distributed data processing with a programming model like MapReduce. [2]

The goal of this paper is to gather some basic information needed to understand how Sparks works and exploring some libraries and his applications, finishing with comparing performance with other tool Hadoop.

This paper is organized as follows. Section 2 and 3 provides concept overview of Parallel and Distributed Systems and Cloud Computing. Section 4 describes an overview of Apache Spark, programming model, libraries, and his applications. Section 5 we are going to compare two similar approaches to big-data processing Apache and Hadoop. We give our conclusions and future work in Section 6.

2 Parallel and Distributed Systems

The idea of parallelism and distribution arouse out of necessity to keep up with the sprouting use of technology and the growing complexity of new applications. Much of the huge increase in single processor performance has been guided by the increasing density of transistors. However, as both of number and speed of transistors increased, their power consumption also increased heating the circuit so much that it would become unreliable. Rather than building faster, complexer and monolithic processors, the industry has decided to build chips with multiple, complete, however relatively simple, processors on it, multi-core processors [5], [6].

Distributed systems in turn consist in a set of hardware and software components connected through a communication infrastructure cooperating between themselves.

To take advantage of the parallelism offered by these multi-core processors and achieve faster programs with more complexity, sequential programs must be rewritten, so that they can make use of multiple cores. This sectionwill provide some introductory definitions and concepts used in parallel programming, along with some techniques to have in consideration when designing reliable parallel programs.

The techniques and concepts presented must be considered when designing parallel systems to build a reliable parallel system. Nowadays there are some tools that can help the programmer to implement, relatively simple, safe parallel systems like Posix Threads and OpenMP [7], [8].

2.1 Processes and concurrent programs

A sequential program specifies sequential execution of a list of statements, its execution is known as a process (Fig. 1). A concurrent program defines two or more sequential programs that their execution may happen concurrently as parallel processes (Fig. 2). [9]

- *R. Ana is with the Faculty of Sciences abd Technology, University of Algarve, Portugal. E-mail: a63971@ualg.pt*
- *R. Artur is with the Faculty of Sciences abd Technology, University of Algarve, Portugal. E-mail: a64592@ualg.pt*
- *Morelli. Jean is with the Faculty of Sciences abd Technology, University of Algarve, Portugal. E-mail: a64014@ualg.pt*
- *C. Ricardo is with the Faculty of Sciences abd Technology, University of Algarve, Portugal. E-mail: a64007@ualg.pt*
- *C. Ruben is with the Faculty of Sciences abd Technology, University of Algarve, Portugal. E-mail: a64591@ualg.pt*

2.2 Difference between concurrent, parallel, and distributed programs

A concurrent program illustrates actions that may be operated simultaneously.

A parallel program is a concurrent program that is de-

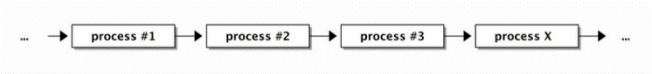


Fig. 1. Illustration of a sequential execution

signed for execution on parallel hardware.

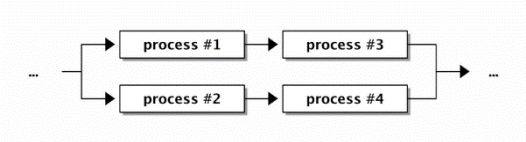


Fig. 2. Illustration of a concurrent execution.

A distributed program is a parallel program designed for execution on a network of autonomous processors that do not share main memory [10].

2.3 Disjoint Process

Disjoint processes may be seen as independent processes defined by concurrent statements that are completely independent of one another. They are usually carried out simultaneously only to utilize the computer more efficiently, but conceptually, they could just be executed sequentially. To make a parallel programming easier to be managed and more efficient, disjoint processes should be used wherever possible. But all parallel programming systems must occasionally allow concurrent processes to exchange some data between them, known as interacting processes [3].

2.4 Communication

Concurrent processes sometimes compete for the use of shared resources or they may cooperate on common tasks. Synchronization is a general term for timing constraints on interactions between concurrent processes. A well-known synchronizing tool, the **semaphore**, invented by Dijkstra in 1965, consists in a variable used to exchange timing signals between concurrent processes using two operations, *wait* and *signal*.

2.5 Critical Regions

Considering concurrent processes that exchange data, it is often important to protect some parts of the program, so it does not lead to an erroneous behavior. These protected parts are known as critical regions, and three assumptions are made about them [3]:

+ Mutual exclusion: Only one process can access the critical region at a time.

+ Termination: A process will always complete its execution in the critical region within a finite time.

+ Fair scheduling: Within a finite time, a process can always enter a critical region.

These critical regions can also be conditional, besides having the same three assumptions as described above, they can only be accessed if a certain condition is satisfied, a common example of this case is a buffer (none of the concurrent processes can access the buffer if it is full).

3 Cloud Computing

Cloud computing can be defined as "A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet. " [4]

There are three main factors contributing to the surge and interests in Cloud Computing: 1) rapid decrease in hardware cost and increase in computing power and storage capacity, and the advent of multi-core architecture and modern supercomputers consisting of hundreds of thousands of cores; 2) the exponentially growing data size in scientific instrumentation/simulation and Internet publishing and archiving; and 3) the wide-spread adoption of Services Computing and Web 2.0 applications [4]

3.1 Cloud computing and Grid computing

In the mid 1990s, the term Grid was coined to describe technologies that would allow consumers to obtain computing power on demand. Costs related to this computing are more elevated compared to cloud computing.

The vision between cloud computing and grip is the same—to reduce the cost of computing, increase reliability, and increase flexibility by transforming computers

		Grid Computing	Cloud Computing
Business Model		One-time payment and project-oriented (e.g.: TeraGrid)	Consumption basis (e.g.: Amazon Compute Cloud EC2 and Data Cloud S3)
Architecture		Address large-scale computation problems 5 layers: 1) fabric, 2) connectivity, 3) resource, 4) collective, and 5) application layers)	address Internet-scale computing problems 4 layers: 1) fabric, 2) unified resource, 3) platform, and 4) application
Services		Grids define and provide a set of standard protocols, middleware, toolkits, and services built on top of these protocols	Infrastructure as a Service (IaaS) (ex.: Amazon S3) Platform as a Service (PaaS) (e.g.: Google App Engine) Software as a Service (SaaS) (e.g.: Live Mesh from Microsoft)
Resource Management	Compute Model	batch-scheduled: a local resource manager (LRM) manages the compute resources for a Grid site, and users submit batch jobs (via GRAM) to request some resources for some time (queuing system)	Resources in the Cloud being shared by all users at the same time
	Data Model	Data Grids – Virtual Data that gives location transparency and materialization transparency	Data, Clouding Computing, and Client Computing.
	Data Locality	Data storage usually relies on a shared file systems (e.g. NFS, GPFS, PVFS, Luster),	When a file needs to be processed, the job scheduler consults a storage metadata service to get the host node for each chunk, and then schedules a "map" process on that node, so that data locality is exploited efficiently (e.g.: MapReduce)
	Virtualization	No	Yes. Virtualization allows: 1) abstraction and encapsulation, 2) server and application consolidation, 3) configurability, 4) increased

from something that we buy and operate ourselves to something that is operated by a third party.

In cloud computing, cloud servers are owned by infrastructure providers. In Grid computing, grids are owned and managed by the organization. Cloud computing uses services like Iaas, PaaS, and SaaS. Grid computing uses systems like distributed computing, distributed information, and distributed pervasive.

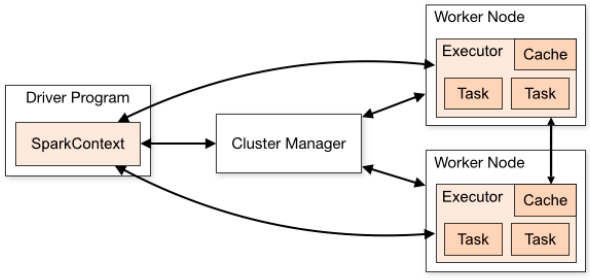


Fig. 5. Apache Spark architecture [13]

4 Spark

Apache Spark is an open-source framework that handles big data processing with a distributed system in different clusters and it offers a set of libraries for its unified engine for big data [2].

The unified API that Spark provides has several benefits. For example, it is much easier to develop applications because of its unified API. Also, it is much more efficient, there is no need to write the data to storage to pass it to another engine. Functions in Spark are often run-in memory.

Apache has grown to be the largest open source data processing project with more than 1,000 contributors and is in use in more than 1,000 organisations[2].

4.1 Motivation

The amount of data that is consumed is increasing rapidly, in 2018 this article from Forbes indicates that there were 2.5 quintillion bytes of data created each day and that Google process more than 40,000 queries every second [11]. With all the data on the internet people saw the opportunity to use this data to extract usable information. This process is called Data processing, which consist in collecting and then translating the data into usable information. With the need to handle large amounts of data, it was natural the development of data processing frameworks.

4.2 History

Started in 2009 as a research project, at the UC Berkeley AMPLab, to design a unified engine for distributed data processing. It was open sourced in 2010, in 2013 it became an Apache top level project and today it is the largest open source data processing project [12]

4.3 Architecture

Apache Spark architecture contains a driver, executors and a cluster manager between the driver and the executors.

The master node contains the driver program, inside the driver programs the Spark Context is created. Spark Context takes the job and breaks into tasks. It will also create RDDs, the building blocks of any Spark application and will be discussed more in detail later in this article, for now it can be seen as immutable collection. The cluster manager is responsible to distribute the tasks and the partitioned RDD over the worker nodes. Executors run on these nodes and will execute the tasks given to each worker. These tasks will perform operations on the RDD and then return the results to the Spark Context. [14]

Another abstraction that Apache Spark architecture is based on is the **Directed Acyclic Graph**, or DAG, which is Apache Spark scheduling layer. DAG determines the order and which nodes the tasks are executed.

4.4 Programing Model

4.4.1 Resilient Distributed Datasets

Spark uses a very central form of abstraction that enables it to achieve a great degree of generality while having performance comparable to a specialized system, namely, this programing abstraction are the Resilient Distributed Datasets, or RDDs for short. These are fault-tolerant collections of objects partitioned across cluster that can be manipulated in parallel.

Users can apply two types of operations on RDDs, "transformations" and "action", the former is used to create RDDs and the latter returns a result based on the RDD that this operation was executed on, we will return to this later.

RDDs are exposed through a functional programming API available in Scala, Java, Python and R. This API allows the user to simply pass local functions to run on the cluster.

4.4.2 RDD operations

As previously mentioned, there are two types of operations that a user can apply to RDDs, namely "transformations" and "action", in this part we will explore further such operations.

RDD transformation, such as map, filter and groupBy, are functions that given an RDD as input returning one or more RDDs as output, effectively creating new RDDs. These transformations are lazy in nature, meaning, once called they are not computed immediately, that is Spark only mantains the record of which operation is being called, as well as the relations between RDDs through a directed acyclic graph, DAG for short, also known as the graph of transformations, in parallel Spark also creates an

execution plan based in the graph of transformations, so right after creation an RDD does not contain data. After an action operation is called Sparks looks at the whole graph of transformations used to create the execution plan and finally computes the RDD [15].

This lazy evaluation of operations of type transformation provides several benefits, one of them is that it allows spark to find an efficient plan for the user's computation, that is, for example, if we have several filter operations in a row Spark can merge them all into a single filter, this allows for better performance by reducing the number of queries and allows the development of modular programs without losing performance.

RDD action, like count, collect and take, are functions used to work with the actual dataset created by a RDD transformation and return a value based on the RDD that this operation was executed on. An RDD action doesn't create a new RDD unlike the RDD transformation, so the returned values are non-RDD values that are stored to drivers or to the externel storage system. This not only brings the laziness of RDD into motion but each time a action is called on a given RDD, that RDD must be recomputed for each of the actions, this is caused because RDDs are ephemeral by default, however RDDs have explicit support for data sharing among computation, this will be explored further in the following sub chapter.

4.4.3 RDD data sharing

RDDs provide explicit support for data sharing among computations, meaning, users can choose to persist selected RDDs in memory for rapid reuse, this operation is achieved by calling the persist action, and if the data does not fit in memory, Spark will spill it to disk, it is also possible to choose where to store the dataset. This approach resolves the issue of needing to recompute the same RDD each time an action is called on it, providing speedups of up to 100x and being key to Spark's generality.

When a RDD is persisted, each node saves only partitions of said RDD that it has computed in memory, and reuses in other actions on that dataset, or datasets derived from it.

4.4.4 RDD fault tolerance

Besides data sharing and a variety of operations, RDDs are also capable of automatically recover from failures. This is achieved through an approach named "lineage", where in each RDD tracks the graph of transformations that was used to buid it, and if any fault arises in any machine, Spark can use this graph to rebuild any partition of the lost dataset on said machine by recomputing the RDD

4.5 Higher-Level Libraries

Spark offers a variety of higher-level libraries, targeting many of the use cases of specialized computing engines, such libraries are based on the RDD approach, this allows the libraries to have comparable performance with specialized systems designed with the same use case in mind as well as allowing for easy combining of the different libraries in applications, for example, combining the machine learning library with the streaming library to train an artificial intelligence using live data, or combining the machine learning library with the SQL library to train it using data store in databases, this process is made easy due to this communality of using RDDs making the development of said applications faster.

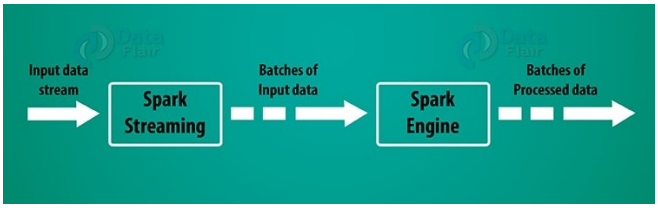
There are four main libraries that we will discuss in detail in the following sub chapters, these libraries are the following

4.5.1 Spark SQL

Spark SQL, as the name implies, is responsible for the implementation of SQL and Dataframes. Spark SQL uses techniques similar to analytical databases, that is, inside an RDD, the data layout is the same as the one used in analytical databases, that being one of compressed columnar storage, that is, each column is its own independent array, holding the values of that column for each row, and for each of these arrays is applied type specific compression, also using cost-based optimization and code generation for query execution.

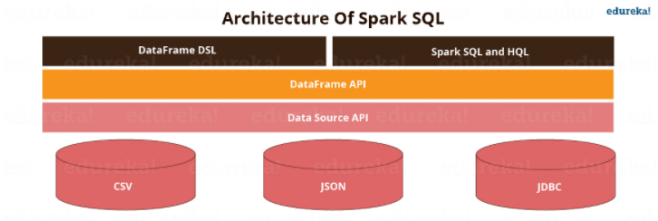
Spark SQL also introduces an extra level of abstraction, that being DataFrames, these being RDDs that have a known schema, in other words, these represent a dataset organized into named columns, being conceptually equivalent to a table in a relational database or a data frame in python/R, but with richer optimizations like using Spark SQL's query planner, making it so user code receives optimizations like: predictive pushdown, operator reordering and join algorithm selection, and much like RDDs, DataFrames also execute lazily.

A DataFrame can be created from a wide variety of



sources, such as: structured data files, like JSON, tables in Hive, external databases or existing RDDs. In short Spark SQL is structured like so:

Fig. 2. Architecture of Spark SQL.



Where Data Source API is responsible for the loading and storing structured data, supporting Hive, Avro, JSON, JDBC, Parquet, and others.

The DataFrame API implements the DataFrame abstraction mentioned earlier.

One technique that is, as of time of writing, not implemented on Spark SQL is indexing, a technique that involves the use a index table that points to a block of data where a specific data entry is located speeding up the query, although other libraries such as IndexedRDDs do use it.

4.5.2 Spark Streaming

Using a model called "discretized streams", the Spark streaming library implements incremental, scalable, high-throughput and fault-tolerant processing of live data streams, this being the core scheduling module of Spark.

Streaming over Spark is achived by splitting the input data into small batches such as every 200 milliseconds that are combined with the state inside RDDs. This way Spark streaming can take adventage of fault recovery, namely the "lineage" approach mentioned before, this being less expensive than traditional methods, offering the low latency required to work with live data. Also offers the possibility to combine the different tools offered with Spark with Spark Streaming easing the processing of the data.

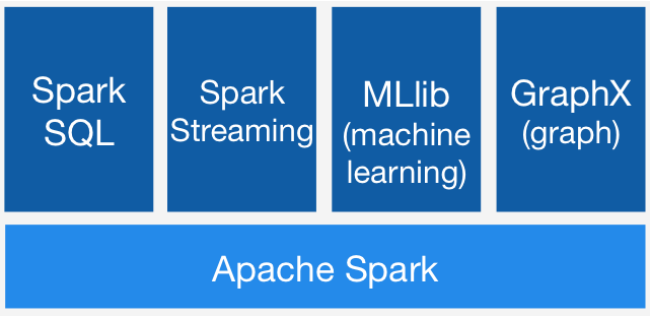


Fig. 6. The main libraries offered by Apache Spark [16]

Fig. 3. Typical structure of a stream processing system in Spark.

Figure 3 represents the typical structure of a stream processecing system, where Spark Streaming receives the input data, sending it to the spark engine to be processed.

4.5.3 GraphX

GraphX is responsible for providing a graph computation interface similar to Pregel and GraphLab, offering the same placement optimizations as these systems, such as vertex partitioning schemes.

4.5.4 MLlib

This library is Spark's machine learning library, implementing over fifty common algorithms for distributed model training like common distributed algorithms of decision trees (PLANET), Latent Dirichlet Allocation, Alternating Least Squares matrix factorization. [17]

Before Spark 2.0 the main machine learning API was RDD based, but after Spark 2.0 the main API is the DataFrame based API, these DataFrames being the same mentioned before in the Spark SQL sub section. With this MLlib includes a framework for creating machine learning pipelines, through the standardization of APIs for machine learning that MLlig offers, making it easier to combine multiple algorithms into a single pipeline or workflow. This standardization is achivied through the introduction of the ML Pipelines concept.

A ML Pipeline in MLlib provide a uniform set of high-level APIs built on top of the DataFrames API and can be divided into five main concepts: DataFrames, Transformer, Estimator, Pipeline and Parameter.

A DataFrame is the same as the one mentioned in the Spark SQL section, so it will not be explored deeply again.

Transformers are akin to transformation operation in RDDs but applied to a DataFrame, that is a transformer implements a method that converts a DataFrame into another.

Estimators are yet another form of abstraction offered by Spark, where an estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on data.

A Pipeline consists basically a sequence of transformers and estimators to be run in a specific order.

Parameters are the parameters for estimator and transformers, like for example the max number of iterations of a logistic regression, parameters can be set individually for an instance or through the passing of a paramMap, that's a set of (parameter, value) pairs. [12]

4.6 Aplications

Spark is used in a wide range of applications, having more than 1000 companies using it, in areas such as web services to biotechnology to finance, also having several scientific applications in academia of several scientific domains, typically combining several of the libraries offered by Spark.

4.6.1 Batch Processing

One of the most common applications of Spark is batch processing on large datasets, including Extract-Transform-Load workloads to convert raw data into a more structured format, and the training of machine learning models. Examples of this workload are: personalization and recommendation at Yahoo, managing a data lake at Goldman Sachs, graph mining at Alibaba, financial Value at Risk calculation, text mining of customer feedback at Toyota and a 8000 node cluster at Tencent that ingests 1PB of data per day. Applications of this type often run only on disk, instead of in memory.

4.6.2 Interactive queries

Interactive use of Spark can be divided into three main classes:

Spark SQL for relational queries, meaning organizations often use Spark SQL to compute interactive queries

on databases, often through business intelligence tools like Tableau. Examples of this include eBay and Baidu.

Using Sparks API through shells or visual notebook environments. This is crucial for developers and data scientists for asking more advanced questions and for designing models that eventually to the production of applications.

Domain specific interactive applications that run on spark, developed by several vendors. Examples of this include Tresata, Trifacta and PanTera.

4.6.3 Stream processing

Another popular use case of Spark is real time processing of data, both in analytics and in real-time decision-making applications. Real world examples of use of this use case include network security and monitoring at Cisco, prescriptive analytics at Samsung SDS and log mining at Netflix.

Applications that use streaming are often combined with batch processing and interactive queries, like for example Conviva, a video company that uses Spark to maintain a model of content distribution server performance, querying it automatically when it moves clients across servers, in a application that requires substantial parallel work for both model maintenance and queries, being that this model is maintained continuously.

4.6.4 Scientific aplications

Spark is also used in several scientific domains including large-scale spam detection, image processing and genomic data processing. One example is Thunder, a platform for neuroscience at Howard Hughes Medical Institute, Janelia Farm, this combines batch processing, interactive queries and stream processing to process brain imaging data from experiments in real time, scaling up to 1TB/hour of whole brain imaging data from organinsm. Thunder allows researchers to apply machine learning algorithms to identify neurons involved in specific behaviours.

5 Spark vs Hadoop-Map Reduce

To adequately analyse the features of *Spark*, we are going to compare it to his main "competitor", another big-data processing framework called *Apache Hadoop*.

Before we start comparing the pros and cons of both programming approaches, it's important to clarify what exactly is *Hadoop* and *Map Reduce*.

Map Reduce is nothing more than a programming model, introduced by Google [18], whose purpose is dictate a way to process and generate large data sets in a scalable, reliable and fault-tolerant way [2]. It essentially consists of two global operations which, as its very own name indicates, are *map* and *reduce*. Users are able to specify a *map* function that processes key/value pairs, outputting another set of intermediate key/value pairs, and a *reduce* function that essentially merges all the values who follow a certain criteria-set into a single value [3].

Apache Hadoop [4] is essentially an *Java* *open-source* implementation of *MapReduce*, used commonly for many different classes of data-intensive applications. It's composed of two layers, a data storage one via *HDFS*, and a data processing one, using the *Hadoop* *Map-Reduce* implementation . *HDFS* stands for *Hadoop distributed file system* and is file system for distributed systems that can be characterized by being highly fault-tolerant and designed using low-cost hardware. [5]

As a way to better organize the analysis of *Apache Spark* versus *Hadoop Map-Reduce*, we'll divide the analysis in 9 primary points, which are :

* **Performance**
* **Cost**
* **Fault-tolerance**
* **Data-processing**
* **Ease-of-use**
* **Scability**
* **Security**
* **Machine-learning**
* **Scheduler**

5.1 Performance

Apache Hadoop *MapReduce* has been around for years, being one of the first frameworks for data processing, and is widely used and loved by the community. Over the years it has evolved and improved but *Apache Spark* can perform 10 to 100 times faster than *MapReduce*. [13]

Apache Spark outstanding performance of 100 times faster than its competitor is due to the in memory characteristic instead of the disk.

![Screenshot 2021-05-05 at 17.06.14](/Users/jeanmorelli/Documents/UALG/spd_grupo/spd_artur/table.png)

**Fig x - ** pdf referencia [14]

The figure **x** was taken out of the article [14] where both frameworks were tested. It shows that *Apache Spark* outperformed *Hadoop* in all scenarios, in memory and on the disk.

5.2 Cost

From a cost point-of-view, *Apache Hadoop* is undeniably less expensive as it only relies on disk memory for its computations. Since *Spark* requires big quantities of *RAM* to process and save RDD's in memory, and the cost of *RAM* is significantly more expensive than disk storage, in the end it ends up costing significantly higher.

Using a more practical example for a high-level

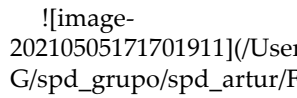
point of comparison, if right now you were to choose a *compute-optimized* *EMR cluster* for *Hadoop* [7], the cost for the smallest available option would be around 0.022 euros per hour. If we were to do something analogous for *Apache Spark*, using the *smallest-memory cluster optimized* for *Spark* [8], the respective cost would be 0.056 euros per hour [6]. Therefore, on a per hour basis, the value for the *Hadoop* is more than *two* times cheaper than *Spark*.

It is, however, important to note that while *Spark* is indeed the more expensive option, if one was to optimize both for computing time, then *tasks* will usually take less time than the *Hadoop* cluster.

5.3 Fault-tolerance

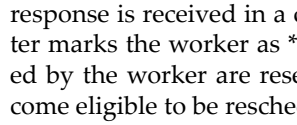
Apache Spark fault-tolerance mechanism lies on the specific behaviour of the RDD's (resilient distributed datasets), using an approach of the name "lineage". Essentially, each *RDD* tracks the graph of transformations that was used to build it and reruns the respective operations on the base data, in order to reconstruct any lost partitions [9]. What this means is that if a partition of an *RDD* is lost, which is what holds the actual data, *Spark* will resort to looking at the graph of the respective *RDD* and rebuild the partition, by applying the transformations that created it on the base data.

Another mechanism that *Apache Spark* has consists on the usage of something called *DAG*, directed acyclic graph, which is used to track the workflow of the various data nodes, which can then be used to rebuild data if so required. This data-structure allows *Spark* to handle failures in a distributed data processing system.



On the other hand, *Apache Hadoop* possesses a highly fault-tolerant mechanism. Essentially what *Hadoop* does is replicate the current data across many of its nodes, meaning that each file is then split into blocks and replicated various times across many machines. This guarantees that if a single machine goes faulty, the file can be rebuilt from other blocks elsewhere [10].

There's also another mechanism implemented in *Hadoop* to deal with faulty nodes. Essentially, there's a master node who keeps track of the status of all the other nodes, who're considered slave nodes. What happens is that the master periodically pings every worker, and if no response is received in a certain amount of time, the master marks the worker as *failed*. Any map tasks completed by the worker are reseted to their initial state and become eligible to be rescheduled to other workers.



5.4 Ease of use

When it comes to ease of use, *Apache Spark* most definitely wins this race. *Spark* provides support to multiple languages, including its native language, *Scala*. The currently supported languages are therefore *Java*, *Python* , *R*, *Spark SQL* and *Scala*. With API support for several languages, it allows developers to use the language they most prefer from a decently sized spectrum of options.

Spark also has another powerful and friendly feature : its interactive mode. It allows the developer to analyze data interactively, providing instant feedback to queries and similar operations.

Finally *Spark* also allows a developer to easily reuse some of their previous written code by providing functions to do so, like for example the *transform()* function. Along side with historical and stream data, this allows developers to reduce application-development time.

As for *Apache Hadoop*, it most definitely contrasts in terms of friendly features with *Spark*. *Hadoop* is a framework based on the *Java* language, with the two main languages to write *Map-Reduce* code being both *Python* and *Java*. With only really two options being available, it stands as a more restrictive tool to use. It also doesn't have any type of support for an interactive mode, providing therefore less feedback to the developer when executing queries and similar operations.

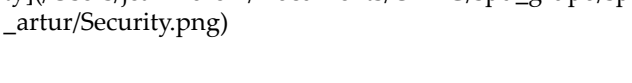
5.5 Security

Hadoop has a very consistent and integral security system, working with multiple authentication and access control methods. One of the main ones is *Kerberos authentication*, which is a third-party authentication service. It essentially works by certifying the identity of the clients, via use of a database of its client and their respective private keys, that are known only to *Kerberos* [11]. There're other ones like *Apache Ranger*, a framework that provides access control capabilities for distributed ecosystems [12], ACL's (access control lists), inter-node encryption, standard file permissions related to HDFS and service level authorization[10].

When it comes to security, *Spark* is most definitely lacking as its security is, by default, off. This means that a new Spark project will be unprotected unless the developer manually enables or adds some. While there're some options to add security to Spark, the amount of possibilities are quite sparse, with the main one being : authentication via secret or event logging.

There're, however, some ways to minimize this flaw. One ironically has to do with integrating *Spark* with *HFDS*, which essentially allows *Spark* to access all the security features and methods present in *Hadoop* and *HFDS*. Additionally, *Spark* can also be ran on *YARN*,

giving him the capacity to utilize *Cerberus*, *Ranger* and many other methods mentioned above.



5.6 Machine learning

As already mentioned previously in this paper, *Spark* has a powerful set of of libraries, including one just for *machine learning* called *MLLib*, mainly used for iterative machine learning applications. This library includes classification, regression, persistence, evaluation, the ability to build machine-learning pipe-lines with hyper-parameter tuning and many more features. It implements more than 50 common algorithms for distributed model training, with some of them being common distributed algorithms for decision trees, La-tent Dirichlet Allocation, and Alternating Least Squares matrix factorization[9].

For *Hadoop*, the framework uses the *Mahout* library as its main machine learning library. Mahout relies on *MapReduce* to perform clustering, classification and recommendations. This library is, however, in process of being phased out in favor of another one called *Samsara*, a *Scala*-based DSL language that allows for in-memory and algebraic operations, also allowing users to write their own algorithms [6].

Mahout has, however, a troublesome flaw. Since machine learning is an iterative process that works best by using in-memory computing and *Map-Reduce* splits jobs into parallel tasks that may be too large for machine-learning algorithms, this might create serious I/O performance issues for *Hadoop* applications using this library.

5.7 Scheduling and Resource Management

Unlike *Spark*, *Apache Hadoop* does not have a built-in scheduler, relying on external solutions to deal with resource management and scheduling issues. As such, if *YARN* is used to run *Hadoop* then it will intrinsically be responsible for the resource and scheduling management of the cluster. It's important to note that *YARN* doesn't deal with state management of individuals applications, only allocating available processing power.

Hadoop can also be used alongside with some third-party plug-ins such as *CapacityScheduler* and *FairScheduler*, who ensure that applications running get the essential resources while also maintaining the efficiency of the clusters.

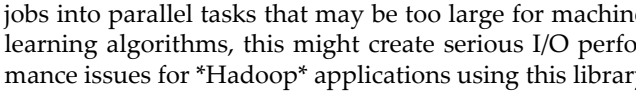
As for *Spark*, he already has these functions built-in. One of these is *DAG scheduler*, responsible for dividing operators into stages, with every stage having multiple tasks that *DAG* schedules and *Spark* then executes.

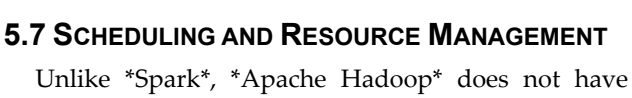
Along side *DAG scheduler*, there's *Spark Scheduler* and *Block Manager* who both perform job and task scheduling, monitoring, and resource distribution in a cluster.

5.8 Data processing

Both this frameworks deal and process data in a completely different way. *Hadoops* data processing works by storing data on disk-memory and then analyzing it parallely in batches, on a distributed environment. Since MapReduce doesn't really require large amounts of *RAM* to handle with vast volumes of data, because it relies on disk-memory, it is best suited for linear data processing (batch processing).

On the other hand, as mentioned before, *Spark* works with *RDD's*. Since they're usually too large to fit in a single node, they usually end up being splitted into partitions and moved to the closest nodes. At the end, all the operations encoded in those partitions are then executed parallely. Since *Spark* ideally wants to have all the present *RDD's* in RAM-memory, this makes the data processing incredibly fast. But not everything is glitters and gold, as *Spark* is incredibly reliant of *RAM*, becoming a serious hindrance when dealing with very large datasets. With the in-memory computations and high-level APIs, *Spark* handles live streams of unstructured data extremely well, therefore shining with real-time processing.





5.9 Scability

When it comes to scability, the differences between *Hadoop* and *Spark* are hard to clearly pin-point. For *Hadoop*, since it uses *HDFS* for its filesystem, as the amount of data grows, it inherently quickly scales in order to accommodate the demands. More so, since it relies on disk-memory to process data, it's also quite scalable in the sense that storage hardware is somewhat cheap nowadays, which allows the creation of extremely big clusters.

As for *Spark*, it doesn't natively have a file system and so, whenever data gets too large to handle, he has to rely on an external file system implementation, with the most commonly used option being *HDFS*. Since *Spark* relies on RAM memory for its processing power and with the prices of *RAM* being reasonably high, even if they've been decreasing with time, it severely limits the

amount of nodes that can exist in a single *Spark* cluster.

To better clarify this notion, so far the biggest *Spark* cluster documented had around 8,000 nodes, working with data-sets in the other of the petabytes, while the *Hadoop* one managed to have close to 42,000 nodes, working with data-sets in the order of exabytes.

6 Conclusion

In the present times, *Apache Spark* stands out as the leading framework for the various fields of Big Data, achieving impressive results in processing time while at the same time being incredibly friendly to use. It makes a big stand by out-performing by an extreme margin, sometimes of over x100 times, his biggest competitor and alternative, know as *Apache Hadoop*, showing an extremely promising future as the best overall Big Data technology.

Spark's rich native libraries, that contain several useful algorithms and data-structures for areas like *Machine Learning*, *Data Streaming* and *Graphs*, facilitates its usage, allowing developers to rapidly take advantage of complex algorithms to solve their problems and develop out of the box solutions.

What makes *Spark* truly innovative is his very unique collection of objects named *RDD's*, which are a central form of abstraction of the technology, allowing it to achieve greats levels of generality while also keeping it's levels of performance and efficiency high. RDD's also have the extraordinary feature of being fault-tolerant, allowing *Spark* to deal with faulty data or nodes in a extremely intelligent manor, not impacting performance levels in a significant way.

Another feature that Spark offers is its ability to persist an RDD in memory, allowing for its reuse without the need to compute it again, this is yet another innovative feature that differentiates Spark from its competitors, giving a significant edge when it comes to performance, becoming indispensable for its versatility.

Spark is therefore an extremely powerful and innovative Big-Data tool, especially ideal for scenarios where the size of the data-sets of the problem in question isn't higher than the total amount of available RAM memory of the cluster . Therefore, if abundant memory is not an issue, *Spark* is most definitely the best tool for any sort of big data problem. Even if that isn't the case, *Spark* is able to produce extremely good results and should still be considered over other alternatives.

References

[1] P. Pacheco, *An introduction to parallel programming*. Elsevier, 2011.

[2] M. Zaharia *et al.*, "Apache Spark: A Unified Engine for Big Data Processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[3] P. B. Hansen, "Concurrent Programming Concepts," *ACM Comput. Surv.*, vol. 5, no. 4, pp. 223–245, Dec. 1973.

[4] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," in *2008 Grid Computing Environments Workshop*, 2008, pp. 1–10.

[5] D. W. Bustard, "Concepts of Concurrent Programming," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1990.

[6] M. J. Sottile, T. G. Mattson, and C. E. Rasmussen, *Introduction to concurrency in programming languages*. CRC Press, 2009.

[7] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, vol. 10. MIT press, 2008.

[8] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[9] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming," *ACM Comput. Surv.*, vol. 15, no. 1, pp. 3–43, 1983.

[10] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 261–322, 1989.

[11] "How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read." [Online]. Available: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=7ce2fd8260ba. [Accessed: 05-May-2021].

[12] "History | Apache Spark." [Online]. Available: https://spark.apache.org/history.html. [Accessed: 05-May-2021].

[13] "Cluster Mode Overview - Spark 3.1.1 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/cluster-overview.html. [Accessed: 05-May-2021].

[14] "Apache Spark Architecture | Distributed System Architecture Explained | Edureka." [Online]. Available: https://www.edureka.co/blog/spark-architecture/. [Accessed: 05-May-2021].

[15] "RDD lineage in Spark: ToDebugString Method - DataFlair." [Online]. Available: https://data-flair.training/blogs/rdd-lineage/. [Accessed: 05-May-2021].

[16] "Scaling relational databases with Apache Spark SQL and DataFrames | Opensource.com." [Online]. Available: https://opensource.com/article/19/3/sql-scale-apache-spark-sql-and-dataframes. [Accessed: 05-May-2021].

[17] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "Planet: massively parallel learning of tree ensembles with mapreduce," 2009.

[18] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.