Politecnico di Milano

Master Degree in Computer Science Engineering

Dipartimento di Elettronica e Informazione

Analysis of Data Movement and Computation Movement with Spark for Fog Environments

Supervisor: Ing. Pierluigi PLEBANI

Advisor: Ing. Mattia SALNITRI

Advisor: Ing. Monica VITALI

Thesis by:

Alessandro MANDELLI Matr. 874642

Academic Year 2018/2019

Contents

List of Figures

Acknowledgements

I would like to thank all the people that encouraged and supported me not only during this period in university, but also throughout my entire school career.
To my parents, who supported me in every decision I made, who pushed me to try and always be better, (almost) without being overly oppressive, but giving their trust in me. Not to mention all the money i made them spend!
To my big brother, who helped me fall in love with technology and computers and was my first IT teacher, and my little brother, who could not care less about engineering, but made me realize that I must not settle for the minimum, but commit to achieve the best possible results.
To my grandmother, who may not have any clue of what my degree is about, but was always eager to know how my exams were going and lit a candle and prayed every time i had to take one.
To professor Plebani, Mattia and Monica, who guided me during the development of this thesis, had the patience to follow my work for months, giving me all the resources and guidelines I needed, and were always available when I had some doubts, questions or requests.
To my study colleagues, with whom I made this journey. Together we had great experiences, took part to some awesome projects overcame the hardest obstacles in our career.
to my friends, who definitely did not encouraged me to study, but were always there to provide a relief and distract me from all the concerns I could have and always congratulated me for the good results (and mocked me for

the bad ones).
Last, but not least, to Birrificio di Lambrate, which gave me and my colleagues shelter in the long days at the Politecnico, where sometimes we actually spent more time than in the classroom.

Vorrei ringraziare tutti quelli che mi hanno incoraggiato e supportato non solo durante il mio percorso universitario, ma anche durante tutta la mia carriera scolastica.
Ai miei genitori, che mi hanno supportato in ogni decisione presa, che mi hanno spinto a provare a migliorarmi sempre, senza (quasi) essere troppo opprimenti, ma dandomi fiducia. Per non parlare di tutti i soldi che gli ho fatto spendere!
A mio fratello maggiore, che mi ha aiutato ad innamorarmi della tecnologia e dei computer, ed è stato il mio primo insegnante di informatica, e mio fratello minore, a cui l'ingegneria non potrebbe interessare di meno, ma che mi ha fatto realizzare che non devo accontentarmi del minimo, ma devo impegnarmi a raggiungere i migliori risultati possibili.
A mia nonna, che potrà non avere la minima idea di cosa riguardi la mia laurea, ma è sempre stata interessata a come andavano gli esami, e accendeva un cero e pregava ogni volta che dovevo sostenerne uno.
Al professor Plebani, Mattia e Monica, che mi hanno guidato nello sviluppo di questa tesi, hanno avuto la pazienza di seguire il mio lavoro per mesi, fornendomi tutte le risorse e le linee guida che mi servivano, e sono stati sempre disponibili per ogni mio dubbio, domanda o necessità.
Ai miei colleghi di studio, con i quali ho compiuto questo viaggio. Insieme abbiamo avuto grandi esperienze, preso parte a fantastici progetti e superato gli ostacoli più difficili della nostra carriera.
Ai miei amici, che non mi hanno decisamente incoraggiato nello studio, ma ci sono sempre stati per darmi sollievo e distrarmi da tutte le preoccupazioni che avrei potuto avere e mi hanno sempre congratulato per i buoni risultati (e preso in giro per quelli cattivi).

Ultimo ma non meno importante, al Birrificio di Lambrate, che ha dato a me e ai miei colleghi un rifugio nei lunghi giorni in università, dove a volte abbiamo effettivamente passato più tempo che in classe.

Chapter 1

Introduction

The service oriented architecture is transforming the fruition of information systems, thanks to the cloud computing paradigm. Research in this field however has not yet taken much care of the data management. There are approaches who aim to provide a cloud database, but this is only a limited aspect of the whole Data as a Service paradigm, since its goal is to take care of the collection, storage, processing and publishing of the data, providing an on demand access regardless of the location.

All these aspects can be particularly important in IoT environments, in which the data is generated at the edge of the network and stored in the cloud, where it can be processed and then supplied to the customer. The high resources of the cloud should ensure high availability and scalability, but, since the data is generated at the edge, the network can have a great influence on the quality of service, adding latency and diminishing the benefits of a cloud based architecture.

A possible solution to mitigate the problem is to adopt the Fog Computing paradigm, which merges the benefits of both the edge and the cloud. Processing data directly on the edge can help to reduce the latency, while the cloud can process a greater quantity of data more efficiently, while allowing the information sharing across the network.

The two main foundations of Fog Computing, especially in data intensive

applications, are the data movement, that covers the placement of the data between edge and cloud, based on the needs of the customer, and the computation movement, the way resources are allocated across the network in order to meet the desired QoS.
The goal of this thesis is the validation of the Data and Computation Movement in a test environment, when adopting Apache Spark as basic technology for data analytics.
Spark is a cluster computing framework, that distributes the task among different machines. We consider it a solution to implement a Fog Computing architecture, especially for what concerns the two movement we focus on:

- the Computation Movement can be achieved thanks to the Spark engine itself, by the reconfiguration of the cluster basted on the requirements of the job, so we only need to find a way to integrate the system with the requirements of Fog Computing.

- The Data Movement instead can be realized thanks to Spark's compatibility with different file systems. Since Spark relies on external data sources, we need to find the right solution to implement the Data Movement, so in this document we confront different file systems, their ability to meet our requirements and their performance, to elect the best candidate.

The work done in the thesis consists in the realization of the fog environment with the help of Spark and the underlying file system and the test of the architecture, through the execution of the movement actions in different scenarios, the analysis of the performances and the collection of the results, in order to validate the quality of the chosen solution.
The environment consists of several machines located in different continents, to simulate a cloud-edge architecture and evaluate the latency of long-distance data transfers and of the Spark cluster communication.
We will present the available resources, with their capabilities and limita-

tions, how they bring the used tools to borderline cases, the consequent choices made in the configuration of the architecture and in the design of the tests, and then we will explain how to implement everything.
We will illustrate how the tests are performed, which kind of data we used, and on what metrics they are focused, then the results will be shown and discussed, aggregated and divided in three main categories.

1.1 Structure of the document

Chapter 2 explains the main terms and concepts necessary to understand the work done, from Fog Computing to Apache Spark, to the file system taken into consideration for the project.
Chapter 3 discusses how the Fog Computing paradigm is interpreted using Apache Spark, and how the movement action can be implemented with this solution. Chapter 4 covers the architecture of the environment in the context of our tests, starting from the available resources, moving then on the configuration choices, based on the given capabilities, regarding the implementation of the fog computing environment and the kind of tests to perform, then illustrates all the steps necessary to set up the environment, from the Spark deployment, to the file systems configuration, to the actual execution of the tests. Chapter 5 enters in the details of the tests performed. The results are grouped by the defined parameters, displayed with box plots and then analyzed and commented.
Chapter 6 summarizes the results of this project, gives a final reasoning on the work done and introduces the next steps to perform for further developments.

Chapter 2

Background

This chapter provides the explanation of the key terms necessary for a complete understanding of the document, from the background, to the tools used, to the goal of the project.

2.1 Fog Computing

Fog Computing is an architecture which aims to provide a seamless integration of resources, data and functionality between the cloud and the edge of a network. It is conceived as a way to improve the Cloud Computing paradigm, especially in relation to IoT, where the latency caused by a vastly distributed network might deteriorate the benefits given by the resources and the high availability of the cloud.
In a cloud-edge architecture the edge nodes (e.g. sensors) have the task to generate the data, which then is transferred to the cloud, where it can be stored, computed and then served to the final user, which will communicate only with the cloud. The benefits of this architecture are the scalability and reliability provided by the cloud. Many users can access the centralized data and computation power without the need of high resources from their side.
The main downside of this architecture is the necessity of a connection between the client and the cloud, since no information is stored locally. This

makes the communication to the cloud the bottleneck for the Quality of Service: the benefit of having a powerful and reliable cloud system is nullified if there is a weak network between to the client, and if the connection is unavailable all the resources become inaccessible. The Fog Computing paradigm aims to overcome these problems, by providing a continuum between the cloud and the edge nodes. It does so by decentralizing the work done by the cloud moving it near the edge, which even in IoT application is becoming more and more powerful, allowing to perform the computation directly where the data is generated.
Fog and Edge Computing have some similarities: they both aim to make the architecture less dependent on the cloud, but they are not to be confused. The goal of Edge Computing is to perform the most possible part of the work on the edge, while Fog Computing tries to provide the best solution for different use cases. In fact the goal is to distinguish the request submitted: if it is a time-sensitive job the computation and data is moved to the edge, near the client, while the less time-sensitive jobs can be submitted to the cloud, which can take care of more complex tasks, like big-data analytics.
According to the NIST definition [8] the following are the characteristics that define the distinction between Fog Computing and other computing paradigms.

- Contextual location awareness, and low latency
  Fog Computing is specifically designed o provide low latency, thanks to the nodes' location awareness, that gives the possibility to move the computation to reduce the cost of the communication between nodes.

- Geographical distribution
  In contrast with the centralized Cloud Computing, Fog Computing is based on a widely distributed network, although its nodes remain geographically identifiable.

- Heterogeneity
  The data collected and processed is of different form factor and different

origins.

- Interoperability and federation
  All the components must be able to interoperate.

- Real-time interactions
  Rather than batch operations, Fog Computing often works with real-time processing.

- Scalability and agility of federated, fog-node clusters
  Fog Computing is able to adapt to different data loads, changes in the network condition and composition of nodes.

Current Initiatives

In order to promote the interest and the developement of this technology, and provide an official definition of Fog computing, Cisco Systems, ARM Holdings, Dell, Intel, Microsoft, and Princeton University, founded the OpenFog Consortium, which provided the reference architecture adopted by the IEEE Standards Association.
Their definition of Fog Computing is:
A horizontal, system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum. [2]
The consortium committed to formalize an open reference architecture, in order to incentivize the diffusion of Fog Computing via a free, standard solution. Their approach give these advantages over other solutions, grouped under the acronym SCALE:

- Security: Additional security to ensure safe, trusted transactions

- Cognition: awareness of client-centric objectives to enable autonomy

- Agility: rapid innovation and affordable scaling under a common infrastructure

- Latency: real-time processing and cyber-physical system control
- Efficiency: dynamic pooling of local unused resources from participating end-user devices

The OpenFog Reference Architecture is driven by eight principles, called pillars, that describe the requirements that a system needs to meet to ensure an horizontal architecture that provides data, computation and control distribution along the cloud-edge continuum. Here they are visualized.

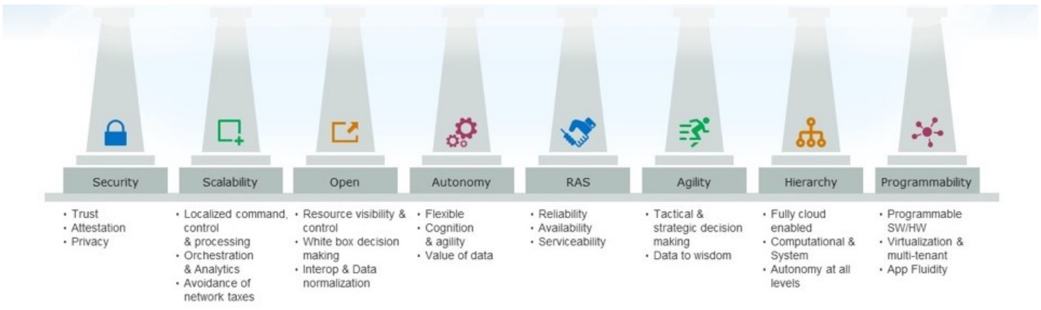

Figure 2.1: OpenFog RA Pillars

Practical example

To help understand how Fog Computing can improve an IoT architecture we give a practical example.

A good implementation in a fog environment is the traffic analysis and control: with the introduction of 5G and the recent improvement in the automotive industries, the communication between vehicles and sensor along highways enables new capabilities in traffic control.

In a traditional cloud-edge infrastructure, the data collected by vehicles and sensors is sent to the cloud, analyzed and then distributed to the requiring nodes. This approach enables great computational capacities and data sharing, since it is all stored in the cloud, which has theoretically infinite resources.

On the downside, however, an instant response from this system is not guaranteed, given all the transferring time between the edge and the cloud, and

all the architecture is dependent on the network connectivity: if the cloud is unavailable, cars and other edge nodes have no data and decision-making capacity to work independently.
On the other side, a solely edge architecture can provide low response time, but does not have the capacity to store and process high quantities of data, and cannot share them with other clients.
Using a Fog Computing approach the system can entrust the cars and the sensors on premise to execute the time sensitive tasks, like live data analysis in relation to autonomous driving, or live traffic control, giving them independence from the cloud, while it can deal with the storage and processing of historic data and the information sharing between all the edge networks.

Fog Computing in Data Intensive Applications

This thesis is focused on a particular application of the Fog Computing architecture, the data intensive environments.
Previous work on this topic [10] tried to provide a definition of the two main pillars of a fog architecture in this scenarios, the data movement and the computation movement, and formalized a decision system, using a goal-based model, that takes into account the requirements of the customer, along with the metrics of data and computation movement, to analyze the job commissioned to the system and execute the right movement actions in order to ensure a better QoS.
For example three factors for a high quality of service are the reliability, the fast response and the data consistency, each then expanded in subfactors. The objective is to validate how the data and computation movement can affect these factors, like the duplication of some data can improve the reliability by increasing the redundancy, while it might decrease data consistency due to synchronization issues between copies.
Now we enter in the details of the two movements.

Data Movement Data Movement takes into account all the actions taken against the data across the network.
Despite the name, it does not only concern the movement, total or partial, of data, but also its duplication and all the possible transformations.
The two main actions are Data Movement and Data Duplication, and each can be divided in four categories: cloud-to-edge, edge-to-cloud, cloud-to-cloud and edge-to-edge.
The Data Movement can occur for example if a set of data is mainly requested by a specific region of the network, so it is moved closer to ensure a better response time, maybe at the cost of a lower scalability. The Data Duplication can be performed to provide a higher reliability or availability to the client, while losing some consistency.
Along with the actions, some transformation can be applied:

Aggregation: when the client request summarized data, like average values, maximum, minimum, it can be performed to save transferring cost and time.

Encryption: if there is a necessity of a secure communication, the data may be encrypted before being moved/copied, adding some computational cost to the action.

Pseudonymization: to ensure privacy some fields of the data record can be manipulated or removed to the data to be transferred.

In this thesis we focus on the actions, in their impact on the performance and how they relate to the decision system to be implemented.

Computation Movement As the Data Movement takes care of the actions regarding the data, the Computation Movement focuses on how to allocate the available resources to the tasks.
Instead of relying entirely on the computing power of the cloud, some tasks can be assigned to the edge or to the nodes closer to the data to process, to ensure (as in the Data Movement) a quicker response to the client. If a task

is instead more complex, it needs more resources, and it can be granted more resources, by assigning more edge nodes, or by moving the computation to the cloud.
Like the Data Movement, the Computation Movement is divided in actual movement and Computation Duplication, although they are not so distinct and mutually exclusive.
The Computation Movement consists in the allocation of a task in a specific node across the network. If the task is normally set to be processed in the cloud and the client has strict time constraints, the computation can be reassigned to a closer edge node, if it has the capabilities, to ensure a faster response time.
The Computation Duplication is the distribution of a task between multiple nodes, to speed up the process. If a job assigned to the edge node requires more resources, the system can add other nodes to the computation in order to redistribute the load and improve performance. In this thesis we cover both the movement and duplication of the computation.

2.2 Apache Spark

Apache Spark is a general-purpose cluster computing framework, used in a wide variety of application, especially the data-intensive ones, for example twitter data analysis [1]. It was built on top of Hadoop MapReduce to implement different types of computation, like interactive queries and stream processing. Hadoop has has been extensively used in the industry, since it provides a simple programming model and it is a scalable, flexible, fault-tolerant and cost effective solution. The main downside of Hadoop is the ability to maintain performance with large datasets.
Apache Spark was developed with the objective of speeding up the computational time of Hadoop, while maintaining its strong points.
The main feature of Spark that contributes to its fast performance is the in-memory cluster computing: instead of fetching the data from disk each

time it is needed, Spark stores the data in memory, in its fundamental data structure, the RDD [13]. This solution gives a much quicker response time with respect to Hadoop and other cluster computing solutions.

History

Before Spark and Hadoop, Google needed a solution to handle the increasingly massive volume of content on the web, so they developed MapReduce [6], a resilient distributed computing framework, that enabled them to distribute the load across a large cluster of servers. The Google strategy consisted in 3 main concepts:

Data Distribution: when uploaded, the data is split across the nodes and replicated among the cluster.

Computation Distribution: the map/reduce job is distributed across the cluster: firstly a partial mapping and reduction is executed in the different nodes across the cluster, then the partial results are aggregated and computed to reach the final result.

Fault Tolerance: both data and computation are resilient to the failing of a node, by the reallocation on another node.

A year after the publication of Google MapReduce, Apache Hadoop was created, following the principle of the previous solution.
Finaly in 2009 Spark came to life as a project of the AMPLab at the University of California, Berkeley, as an attempt to keep the benefits of MapReduce's scalable, distributed, fault-tolerant processing framework, while making it more efficient and easier to use. The project was then donated to the Apache Software Foundation, where it became one of the most active projects managed by the foundation, with the contribution of many multinational companies, like IBM and Huawei.
The main benefits of Spark with respect to MapReduce are:

- much faster execution by caching data in memory across multiple parallel operations, whereas MapReduce involves more reading and writing from disk..

- multi-threaded tasks inside of JVM processes allow faster startup, better. parallelism, and better CPU utilization.

- richer funcional programming model, and wider support for different computations.

- parallel processing of distributed data with iterative algorithms.

Spark Architecture

The spark framework is composed of a main component that provides the basic functionalities, and other four components that expand the capabilities in different use cases.
The Spark Core is the foundation of the project, it provides task management, scheduling, I/O capabilities, and the fundamental structure of the framework, the Resilient Distributed Datasets (RDD), explained in depth further in this section.
The four other components are:

- Spark SQL provides a new data abstraction called DataFrame (improved by DataSet), which provides support to structured and semi-structured data, and SQL support.

- Spark Streaming takes advantage of the fast scheduling capabilities to provide support to streaming analysis in addition to batch computation.

- MLib is a Machine Learning framework that, thanks to the distributed memory-based architecture of Spark can be as nine time faster than concurrent solutions, like Apache Mahout

- GraphX is a graph-processing framework that provides APIs for graph computation, based on RDD.

The main abstraction of the Spark framework, as previously mentioned, is the RDD. It is a fault-tolerant (Resilient) collection of data (Dataset) that can be computed in parallel across the cluster (Distributed). It is an immutable, read-only collection, and the data is partitioned and distributed across the cluster. It achieves its resiliency by duplicating the chunks across multiple nodes, so if one fails, another one can take over. RDD operations are executed with a lazy evaluation, which means that the various transformation are queued and executed only when triggered by an action, such as the collection of the results. It can be done to another key feature of Spark: the Directed Acyclic Graph (DAG), a graph that collects and stores all the transformations to be applied to the data, allows Spark to optimize the computation and together with RDD gives fault tolerance.

Spark has a master/slaves architecture, summarized in the following diagram.

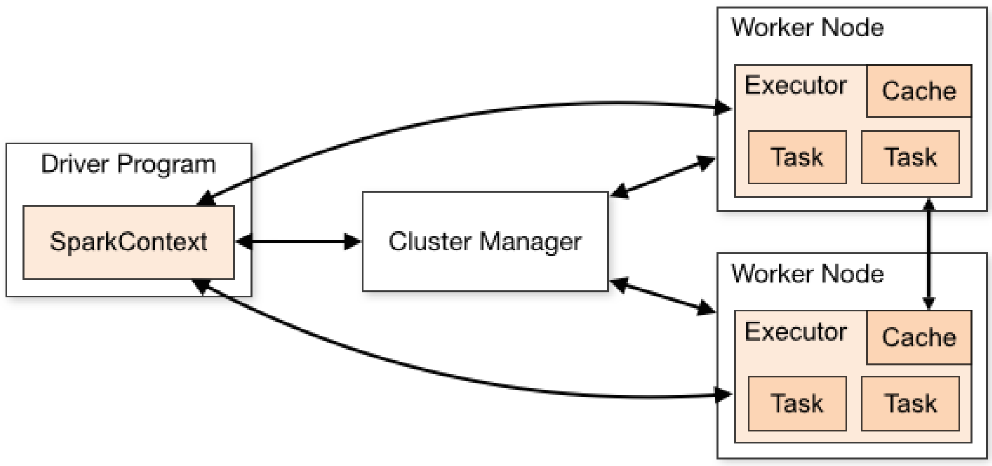

Figure 2.2: Spark Cluster Architecture

In the master node resides the driver program, which runs the application. The code written and submitted to Spark acts as the driver, and the object

that coordinates the work and offers all the functionalities is the SparkContext, through which all what is done within Spark goes.
The SparkContext and the driver take care of the job execution and its partitioning in the cluster, while the Cluster Manager has the job to control the nodes and provide the resources requested by the driver to perform the tasks submitted by the spark context. Everytime an RDD is created it is distributed to the worker nodes and cached there.
The worker node have the solely job to execute the tasks assigned to them on their partition of the data, and then send the results to the driver, which will aggregate the partitions and compute the final result.
Now we see the complete workflow of a Spark application:

1. Firstly the client submits the application code to the spark cluster. When the driver reads the code, it transforms all the actions and transformations in the DAG, and performs some optimizations.

2. Then the DAG is converted in a physical execution plan, organized in different stages. Each stage is divided in tasks, that are sent to the cluster.

3. The driver then communicates with the cluster manager and requests the resources. By the driver demand, the cluster manager launches the executors on the worker nodes, and the driver sends the tasks to them, according to the data locality. all the executors are registered with the driver, which in this way can have a complete view of the job.

4. The driver monitors the execution of the tasks by the workers and schedules the future tasks.

5. At the end of the execution, each task result is sent to the driver, that performs the final aggregation and can save the final results to disk.

Spark is compatible with different cluster managers, like Apache Mesos, Hadoop Yarn and Kubernetes, but is also deployed with its own standalone cluster, which makes the configuration the system easier.

File systems

In order for a Spark application to work properly, each node of the cluster needs to have access to the data. To do so Spark offers a native compatibility with the Hadoop Distributed File System (HDFS), a solution to store data across a cluster of servers.
HDFS is composed of namenodes, which store the metadata, and datanodes, which store the actuale chunks of data. When a file is uploaded to HDFS, it is striped in fixed partitions, and distributed to the datanodes. Depending on the configuration, other copies of the chunks are distributed as well, to provide redundancy and fault tolerance.
In a Spark environment, HDFS is typically deployed on the same nodes on which resides the Spark workers, in order to take advantage of the data locality. In fact the driver program of Spark automatically schedules the tasks based on the proximity of the workers to the data, to reduce the access time and latency.
This however is not the only solution to distribute the data, since Spark is compatible with every file system which can be mounted by the operating system, so virtually any kind of solution is adoptable.
To use Spark in a test environment it is even possible to copy the data on the local file system of each machine of the cluster, although this is logically an unfeasible solution in a working deployment.
Alternative solutions may include other distributed file systems, like GlusterFS of CephFS, which we cover in the dedicated section.

Why choose Spark

Among all the different reasons to choose Spark, we can provide three main pillars:

- Simplicity: Spark provides a rich set of APIs, for different programming languages, that, along with a standalone cluster, allows developers and scientist to easily configure and run it.

- Speed: Spark was built with this objective in mind. Its ability to process in-memory data for interactive queries makes Spark as much as 100 times faster as Hadoop MapReduce.
- Support: Spark supports a large number of programming languages, has a wide integration with many storage solutions, has a vast community all around the world, and a large number of companies is supporting and integrating it in their products.

Use Cases

Spark was conceived to be a general-purpose framework, was set to expand the capabilities of Apache Hadoop, and provides extensive APIs for different programming languages, namely Scala, Java, Python and R, so it can be applied to a wide variety of use cases.
The different fields of application can be divided by the main components of Spark.
Developers have to face increasingly more often with large streams of data, like log files, or sensor readings, coming from different locations. While it can be possible to store such data to be later computed in batches, it can be necessary to process them as they arrive.
For example, the data streams related to a security system in an oil rig can be processed in real time to instantly react to malfunctions and avoid a possible disaster. The real-time processing can be achieved thanks to Spark Streaming capabilities.
One of the most developing branches of computer science in the latest years is machine learning, thanks to the availability of larger data pools to train the software. Thanks to its in-memory processing and its dedicated libraries, Spark is a great choice to implement machine learning algorithms, since it can perform fast repeated queries, bringing to a quicker training of the model. Spark's ability to quickly respond and adapt to alterations allow data scientist to implement interactive queries, that can take into account the response

of the first query and then adjust it or study it in depth.
Spark, along with Hadoop, is often used to reduce the time of Extract, Transform and Load processes (ETL), that consist in the collection of heterogeneous data from different systems, their cleaning and standardization in order to be loaded in a separate system for data analysis.

Relevant Users

A large number of companies is adopting Spark to extend their big data analytics to branches like machine learning and interactive querying.
IBM and Huawei have heavily invested in technologies to integrate Spark in their solutions, and actively funt the Apache project, while many startups are mostly dependent on Spark.
The team responsible for the creation of Spark went on to found Databricks, an end-to-end data platform powered by Spark.
The principal Hadoop vendors have integrated in their solution a YARN-based Spark; web-based companies rely on Spark, like the search engine Baidu, the e-commerce platform Taobao, and the social network Tencent.
Even pharmaceutical companies use Spark, like Novartis, which relies on it to reduce the time to retrieve modeling data for the researches.

2.3 File Systems

Every node of the Spark cluster needs to access the data source, therefore we took into consideration several options to distribute the data.
For the goal of our project we are not looking for the typical criteria to choose a suitable distributed file system, such as high availability, fault tolerance and location transparency, instead we have to be able to choose where a file is located, and easily adjust the file system configuration to adapt to the use cases, so this is not a thorough comparison between file systems, but an analysis of some peculiar capabilities.
The first alternative, that does not require any prior architectural configura-

tion, is to copy the files in advance on all the nodes, in the same path. This is the simplest solution, but is not feasible, both in terms of performance and flexibility: copying the required files on all the nodes adds a sensible overhead to the computation, and the data duplication needs a significant amount of space.
The second alternative is to rely on the recommended file system for Spark: HDFS. This should the best solution in terms of performance, since the files are partitioned and distributed across the nodes, and the cluster can take advantage of the data locality (although it is not true, as proven in further testing). However the file distribution does not allow us to decide where the data is stored, so the data movement testing cannot be performed.
The third alternative is to use a different distributed file system, the two most promising choices being GlusterFS and CephFS
CephFS is quite tedious to set up, has a more complex architecture with respect to GlusterFS -it is more resource hungry-, so it is not the perfect candidate. GlusterFS is easy to configure, in a matter of minutes you can create a new volume and mount it on all the cluster nodes, it can be scaled quickly and can run on a single machine with modest resources. Unlike HDFS and CephFS, the data is not randomly scattered among the cluster nodes, but, provided there is enough space, it is stored in a single node, so we can upload a file and keep track of its location, and this is a fundamental aspect for the data movement tests. To be thorough, however, we tested the performance of the three file systems and, as shown in chapter 5.2.1, GlusterFS resulted to be the quickest in most of the scenarios.
In the end, taking into account all the different capabilities and tests, the file system that better suits our needs is GlusterFS.

Chapter 3

Fog environment with Spark

This chapter extends the previous definition of Apache Spark, its characteristics, what are the connections with fog environments, why it is the right choice for their implementation and how to realize that.

3.1 Spark architecture

Apache Spark, as introduced in chapter 2, is a framework for distributed computation, and follows a master/slave architecture: the master node contains the driver of the application, while the slaves contain one or more workers each.
Spark is deployed on a cluster, i.e. Mesos, Yarn or its own standalone cluster, which will handle the connection between machines, the creation and deletion of slaves, the monitoring of the jobs.
Every node needs access to the data source, so it must be either replicated on all the machines or provided by a distributed file system. Spark can interact with virtually any kind of file system, and it has a native integration with HDFS. Spark is highly tweakable, both at design time and at runtime, and can scale from a simple single machine, single cpu configuration, to a configuration with hundreds of multi-cpu slaves. This flexibility allows the adaptation of the cluster to fit the requirements of each submitted job.

3.2 Spark interpretation of Fog Computing

Fog Computing is a specific field of distributed computing, it is a paradigm focused on the improvement of Cloud based architectures, through the redistribution of the computation and its movement from the cloud closer to the edge nodes, where the data is generated, or wher the final user resides
Since Spark is the most popular framework for distributed calculus, it is reasonable to think that it is one of the best candidates also in fog environments. Its flexibility and scalability pair well with the heterogeneity of a fog architecture. In fact each node can provide a different amount of resources (edge vs cloud), and Spark can automatically adapt and partition the job to be manageable by the given computational power. By communicating with the cluster manager, the Spark driver program knows what are the available resources, and can split the computation in a way that can be optimally distributed across the cluster.
This architecture is very useful for what concerns the computation movement: the cluster can be reconfigured following the requirements and assign the right resources to Spark, that will automatically adapt the program.
Spark compatibility with a wide variety of storage solution allows to explore different possibilities in the Data Movement area: even if this action cannot be performed strictly by Spark, we can choose the best distributed system that will enable such capabilities.

3.2.1 Computation Movement

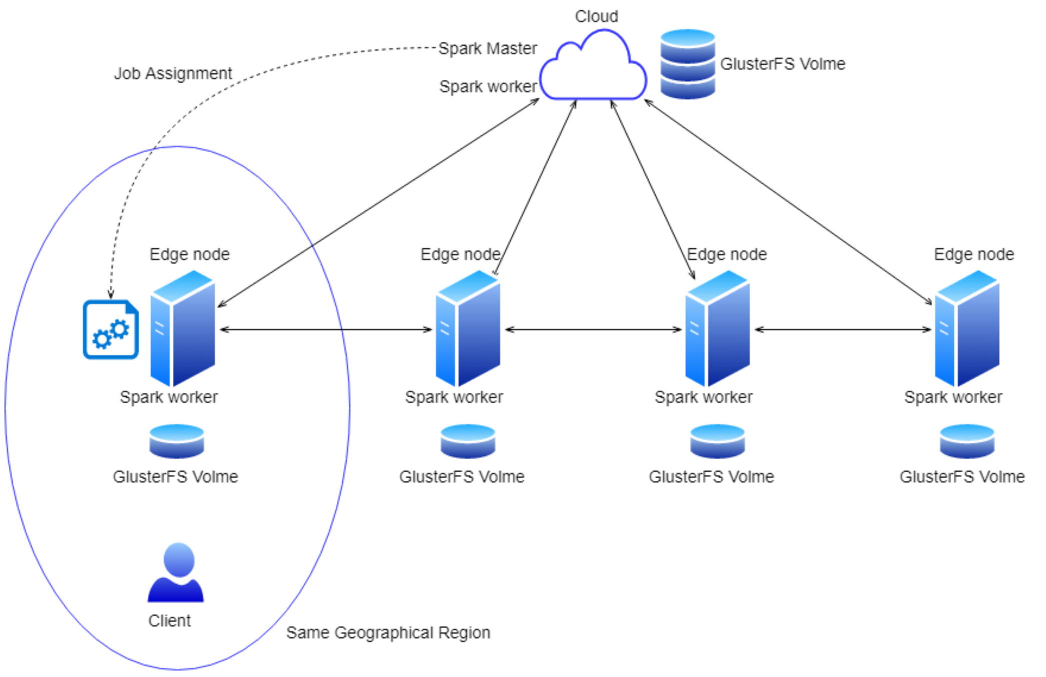

Figure 3.1: Computation Movement with Spark

The Computation Movement consists in the choice of the best nodes of the network to perform a job. It can simply be the movement of the processing from the cloud closer to the data to reduce latency, or in the duplication of the nodes to assign more resources and speed up a job.
To realize the Computation Movement in Spark, we have to rely on the cluster manager's capability of scheduling the job between the slaves.
As previously stated, to schedule the job the driver of a Spark program talks with the cluster manager, which allocates the resources among the available nodes and supplies them back to the driver, which in turn splits the job in tasks and then sends them to the workers.
The best way to implement the Computation Movement in this architecture would be to instruct the cluster manager at runtime to choose a specific set of nodes, dictated by the requirements of the job, and provide them to the Spark application.

However we were not able to modify the Spark standalone cluster in such way, so the alternative solution is to reconfigure the cluster beforehand. Prior to the submit of the job, the required nodes are added to the cluster, which assigns all of them to the application once it is launched, then at the end of the job all the nodes are removed, leaving the cluster empty and ready for another node reassignment.
This method seems too complex and time consuming, but thanks to the scripts provided for the management of the cluster and a little bit of scripting, we can add the chosen nodes and submit the job in a matter of seconds.
Although a complete knowledge of the physical configuration of the cluster is necessary: in order to be able to correctly move the computation, we have to know the geographical location of the nodes, the data and the client, and the address of all the machines.

3.2.2 Data Movement

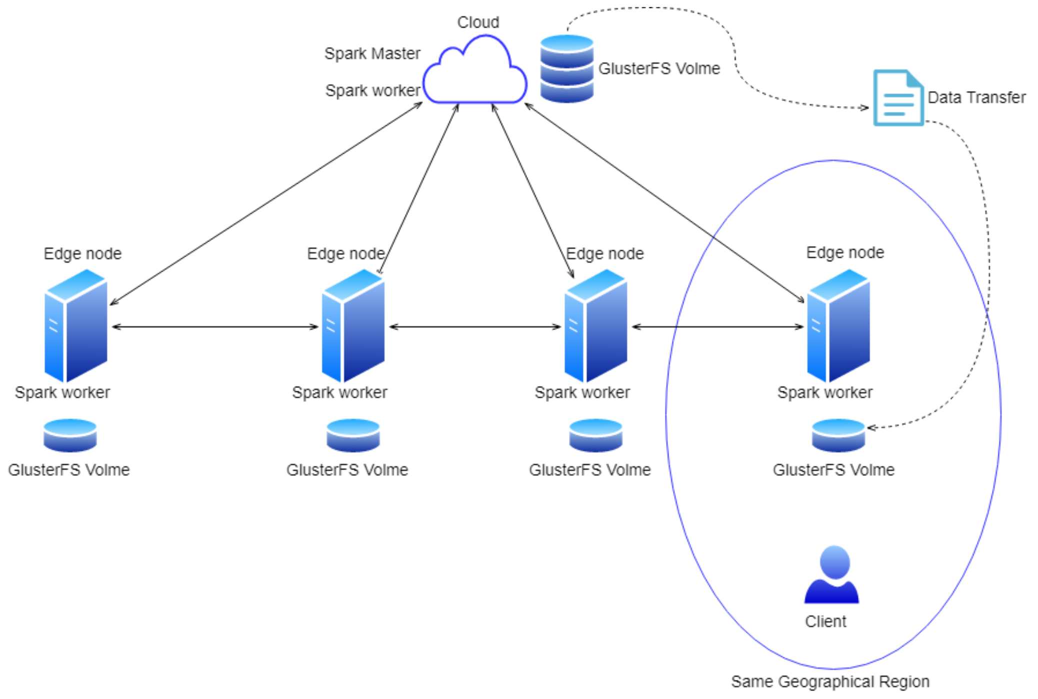

Figure 3.2: Data Movement with Spark and GlusterFS

Since Spark relies on external file systems and the only data movement that it performs internally is the aggregation of the results, it cannot technically take charge of the Data Movement in a fog environment, but the task is remitted to the underlying file system.
The driver program of a Spark application tries to optimize the task assignment according to the data locality, so it automatically schedules a task on a node closer to the physical data location.
This provides great performances in a distributed environment, but does not allow to arbitrarily move the data across the cluster.
Thus to achieve the Data movement we need to find the right storage solution that will integrate well with Spark.
An obvious solution is to manually perform the Data Movement and Duplication on the local file system of each machine of the cluster, but this is only feasible in a limited test environment, and not at all flexible and scalable.
Thanks to Spark's compatibility with different distributed file systems, we can explore these solutions and find the best one that fits the requirements.
The typical reason for the use of a distributed file system is the abstraction of the physical layer: all the different disks and machines that compose the cluster are made invisible by the logical file system.
This is convenient in standard use cases, but not by a Data Movement perspective. In fact, even if it might provide redundancy an thus enable the Data Duplication, this architecture does not give the control on the physical location of the data.
In order to find the right candidate to implement the Data Movement in our environment, we included different solutions in our tests, as explained in chapters 2.3 and 5.2.1, and we identified GlusterFS as the enabler of the Data Movement. Its architecture and configurations allow to set up a cluster, place a file on a specific machine without it being scattered among the other nodes, and make it available to Spark for computation.
This ability of placing the data on a specific machine is the very definition of Data Movement.

The choice of a distributed file system can also give Data Transformation capabilities, such as the encryption during transfer.

3.3 Implementation

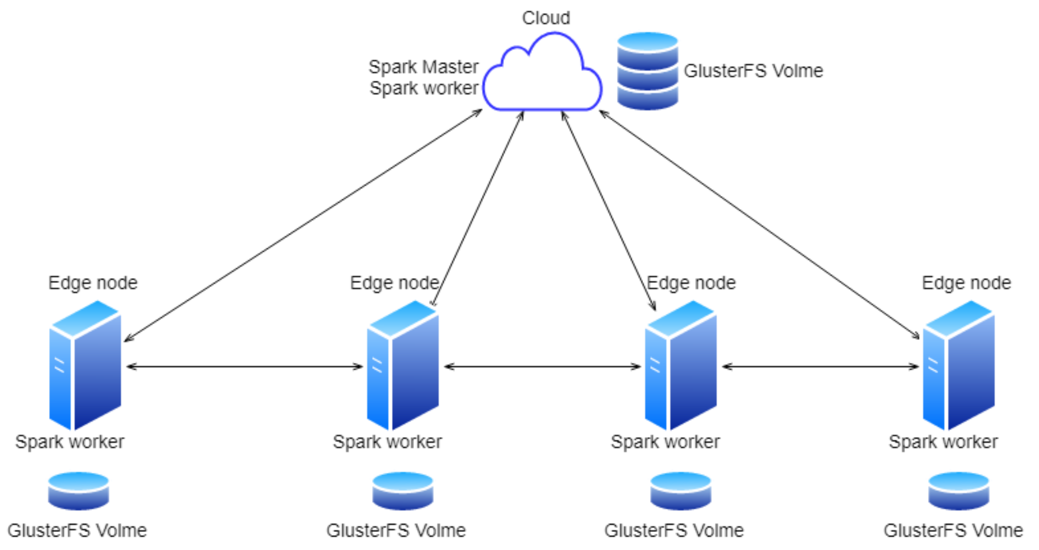

Figure 3.3: Fog Architecture with Spark and GlusterFS

After reasoning on the relation between Spark and Fog Computing, how the Data and Computation movement can be achieved, we now summarize how to implement this system.
We identified Spark and GlusterFS the two enabler of a Fog environment in a distributed system.
Provided a cluster of machines, which can comprise cloud servers, edge machines and IoT devices, we deploy a Spark cluster on all the wanted nodes, placing the master node preferably on the cloud, given its centrality in the network and its high resources.
The GlusterFS cluster is as well deployed on all the chosen nodes.
When a client submits a job, he communicates his location, the data requested and some constraints. If a job consists in the computation of a

large quantities of data, without latency constraints, the Spark cluster will be configured to work in the cloud, which has more capabilities to handle big volumes, but can be prone to latency caused by the distant location.
If instead the job is a computation of live data coming from a sensor, the node closer to the sensor is added to the cluster to provide a more prompt response.
If a piece of data is mainly requested by client located in a specific region, this data can be moved to the closer GlusterFS node(s), to reduce the transfer time.

Chapter 4

Configuration of the test environment

After the reasoning on how to implement fog environments using Spark, in this section we enter in the specifics of the project and explain how we actually realized the architecture using the available resources and how we configured the test environment, with the goal of validating the goodness of the solution we proposed.

4.1 Architecture

The architecture available for the tests is comprised of six machines, all running Ubuntu, distributed across different continents, in order to test both short and long distance communications:

- Provided by Cloudsigma:
  - three machines in Frankfurt
  - one machine in Miami
- Provided by AWS:
  - one machine in Ohio

- One remote machine in Italy

Since all the machines have limited resources (with the exception of the remote one), our Spark instance is set up to work with a single worker per node, with one cpu and 512MB of memory dedicated. This fact limits our possibility in the tests regarding the scalability of resources, although by varying the number of machines we can partially compensate for the restriction.
The available data source is a set of randomly generated blood tests, stored in a JSON file, which needs to be formatted to conform to Spark requirements. Spark natively provides libraries for the reading and parsing of json files into relational tables, on which we can perform queries, but the file has to be formatted in the right way.
Our data is generated as an array of blood tests, but, in order to be correctly partitioned, it has to be formatted such that the file is comprised of a single JSON object per line [3] [12].
Because of the necessity of formatting the file once it is downloaded, and the small computational power of the machines, we have to limit the file size, otherwise the formatting would not be possible.

4.2 Configuration choices

We previously introduced the available machines, discussed about their limitation and how they affect our choices, now we explain how the testing environment is configured according to them, which test we perform, why we choose them and how we collect the results.
Firstly, given the provided data set, the chosen test to perform consists in calculating the average values of the blood tests. Each test has several fields, so we compute the average value of each of them, discarding the null ones, and display the final result on the client terminal
The data is comprised of 200000 blood tests, that are stored in a 800MB JSON file; this particular size is chosen for a couple of reasons: as mentioned in the previous section we are limited by the available resources, secondly we

chose a value that would allow us to speed up the tests, which can take up to 30 minutes for a single job, without flattening the results.
Moving on to the cluster configurations, given the available machines listed in the previous section, the following are the configuration we consider to be sufficient to evaluate all the possible data and computation movement. We list them along with their relation to the fog architecture
-N.B. The master is always located in Frankfurt, which acts as the cloud-

- one slave on the master node, corresponding to the computation on the cloud.
- one slave in Frankfurt, corresponding to the computation on an edge node near the cloud.
- one slave in Miami, corresponding to the computation on an edge node far from the cloud.
- one slave on AWS, corresponding to the computation on an edge node far from the cloud, in a different network.
- one slave on the remote machine, corresponding to the computation to a very remote edge node, with poor network capacity
- two slaves in Frankfurt, corresponding to the data duplication near the cloud.
- three slaves in Frankfurt, corresponding to the computation duplication near the cloud.
- four slaves on the Cloudsigma machines, corresponding to the computation duplication both on the cloud and on the edge.

To make a decision on the optimal filesystem to adopt, we took into account three alternatives: HDFS, GlusterFS and CephFS. After some considerations and testing, covered in chapter 2.3 and 5.2.3, The best one resulted to be GlusterFS.

To simulate different configurations of Data Movement, along with the comparison between file systems, the JSON file is distributed in four different ways, and the tests for all the cluster configurations are repeated in each data configuration. The four data sources are:

- a GlusterFS volume located on the master node, in Frankfurt, corresponding to the file stored in the cloud.
- a GlusterFS volume located on the AWS machine, in Ohio, corresponding to the data moved to an edge node.
- a HDFS cluster deployed on the four Cloudsigma machines, to confront the performances between the storage solutions.
- a CephFS cluster deployed on the four Cloudsigma machines, to confront the performances between the storage solutions.

These configurations allow us to confront the performances of the different file systems taken into consideration, in order to choose the best one, and to evaluate how the computation time varies by moving the data closer to or further from the computation node(s).
As previously mentioned, for each data source we run the tests in each of the cluster configuration listed above, in order to have all the possible combination and make a comprehensive analysis of the architecture.

4.3 Environment setup

Now we briefly explain how the testing environment was set up, from the spark cluster, to the various distributed file systems.

4.3.1 Spark cluster setup

Configuring Spark is quite straightforward [9], so we will briefly explain the main steps to perform.

In order to be able to communicate with the workers, the master node needs passwordless ssh access to the slave nodes, so we need to generate a key pair and distribute the public one to all the necessary machines.
Spark requires a Java and a Scala installation on every node to work, so we have to install them on all the cluster machines.
Then we need to download the Spark binaries, extract them and place them in the same path in all the nodes.
We now need to edit the configuration files. In the `./conf/slaves` file in the master node we will add all the machines used as worker nodes, while in the `./conf/spark-env.sh` file of every slave we will specify the resources dedicated to the worker, in this case 1 cpu and 512mb of memory.
In the `./sbin` folder are located the script used to start the cluster: start-master.sh, launched on the master node, will start the Spark master, start-slaves.sh, run on the master node, will launch all the slaves specified in the `./conf/slaves` file, start-all.sh starts both the master and the slaves. To start a single slave we have to launch start-slave.sh in the worker node, passing as a parameter the address of the Spark master. The cluster web interface can be accessed at master-ip:8080.
Likewise we can use similar scripts to stop the various components: stop-master.sh, stop-slaves.sh, stop-all.sh on the master, and stop-slave.sh on the worker.
To be able to analyze the jobs submitted to the cluster after their completion, we need to enable the history server. On the master node, in the `./conf/spark-defaults.conf` file we uncomment the line `spark.eventLog.enabled true` and specify the log directory in the line `spark.eventLog.dir`, and start the server with the script `./sbin/start-history-server.sh`. Now the history server can be access atmaster-ip:18080.

4.3.2 Filesystems setup

As previously explained, all the Spark workers need to access the data, and we need a tool to implement the Data Distribution in our architecture, so we need to setup a distributed file system. Here is explained how to configure the three chosen ones.

GlusterFS

Since we want the file placed in a specific location in order to simulate the Data Movement correctly, we choose to deploy the GlusterFS volume on a single machine each time, so we do not cover the set up of multiple nodes. Again as with Spark, many resources are available online which explain in detail how to set up the cluster [11].
To set up a GlusterFS volume we need to install the gluster-server package on the cluster nodes. The gluster daemon needs to be started, then the volume is created with the command

```
gluster volume create {volume-name} {node-ip}:/path/to/volume/folder
```

Then on all the client nodes we can install the cluster-client package and mount the volume with the command:

```
mount -t glusterfs {server-ip}:{volume-name} /path/to/mount/folder
```

Now Spark can access the data stored in the Glusterfs volume via the mount folder of each node, as if it were a local file.

CephFS

Setting up CephFS is a more cumbersome operation. An automated tool is available, however we needed to manually execute some steps due to some errors. Given the longer process, we will not explain it, but here is the official documentation that covers in detail each step [4].
To mount the volume on the client nodes we use the mount.ceph command, provided by the ceph package. The command syntax is:

```
mount.ceph {mon-ip}:/ /mnt/foo
```

Once the volume is mounted on the machines, Spark can access the files in the same way as with GlusterFS

HDFS

To install HDFS we need to download the binaries from the Apache foundation site and extract it on all the desired cluster nodes [5]. Then we need to edit the configuration files: in `./etc/hadoop/hadoop-env.sh` we add the correct Java path at the `export JAVA_HOME` line, in `./etc/hadoop/core-site.xml` we set the namenode location, while in in `./etc/hadoop/workers` we add all the datanodes we need. Then we format the file system with

```
./bin/hdfs namenode -format
```

and then start the HDFS with the command

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

Spark is able to access the file in HDFS via the following path format:

`hdfs://{namenode-ip}:9000/path/to/file`

4.3.3 Data set

To edit the data in the correct format we wrote a simple script that automatically formats the JSON file, to make it optimized for the Spark partitioning. The low resources available for the machines, however, limit the file size to a maximum of about 1GB, as, if trying to format a bigger file, the script will crash.
Once the file is downloaded and formatted, it is stored in the file system of choice, following the corresponding method.

4.3.4 Test program

The testing program consists in the loading of the JSON file into a Spark dataset, on which a query is performed, to calculate the average of all the blood test values, and the result is then displayed in the terminal.
To provide an easy customization, some program configurations, like the app name, the file location, the number of partitions, are loaded from an xml file,

stored in a GlusterFS volume.
The program is packaged in a jar file and then submitted to Spark via the `./sbin/spark-submit` script.

4.3.5 Custom script

In order to speed up the testing process a custom script has been made. It gives the possibility to start/stop one or more slaves directly from the master node, it can perform multiple sumbits of a job, and allows a fully automation of the process: with a single command we can start the slave(s), launch a job a given amount of times, and automatically stop the slave(s) at the end of the execution.
This script also allows to run a job without relying on a distributed file system: a function is implemented, that takes the submitted files and distributes them to all the slave nodes prior to the start of the Spark job.
This function however is only used in the early stages of the project, to test the correctness of the Spark environment and of the test program, before the configuration of the file systems, and it is not included in the tests.

Chapter 5

Validation Results

The testing process needs to evaluate two main parts of the Fog architecture, the data movement and the computation movement, and how well they can be executed with Spark.
Each combination of Spark configuration and data configuration is tested ten consecutive time, the total times of the computations are fetched from the Spark History Server, and the results are visualized via box plots.
In addition to the two main goals of the tests, we wanted to evaluate how some Spark settings can impact on the performance, regardless of the external factors.
The two settings we considered more relevant to the project, and therefore tested, are the number of cpus allocated to a Spark worker and the number of partitions of the data.
Since these two topics does not concern the Fog architecture and the main goal of the thesis, only a couple of significant configurations have been tested.

5.1 Results aggregation

The results of the tests are aggregated in three main categories, the two key aspects of the Fog architecture, Data movement and Computation Movement, and a third category, which comprises the tests relevant to the config-

uration of Spark, and how it can be tuned to improve the performance.
The first category is the Data Movement, which in this case is realized by having different deployment, and choosing a different data source accordingly to the cluster configuration.
The intuitive result is that having the data closer to the computation brings better performance, and the tests aim to verify this hypothesis and to assess how much the data location is influential among the overall setup.
The results are therefore aggregated by cluster configuration, in order to visualize the differences between all the data sources (chapter 5.2.1).
The Computation Movement aims to adapt the cluster configuration according to the data submitted, realized in our implementation by either moving or adding workers to the Spark cluster.
The tests aim to verify the influence of the location of a Spark slave (the computation node) with respect to the master and to the data source, and how much allocating more resources, i.e. adding nodes to the cluster, can improve the performance
Thus the results are aggregated by data source, hence giving a visual comparison of the computational time of all the cluster configurations.

5.2 Results

In this section we display the final results of the tests performed and discuss them in relation of the scope of the thesis, grouped by the three aforementioned categories: Data Movement, Computation Movement and Spark settings. The results are displayed as box plots, which provide a quick view of the average, maximum, minimum time and the consistency of the performance, as well as a visual comparison between the configurations.

5.2.1 Results - Data Movement

The following graphs show the results of the Data Movement simulations: each graph refers to a configuration of the Spark cluster, while each box plot refers to a different data source.

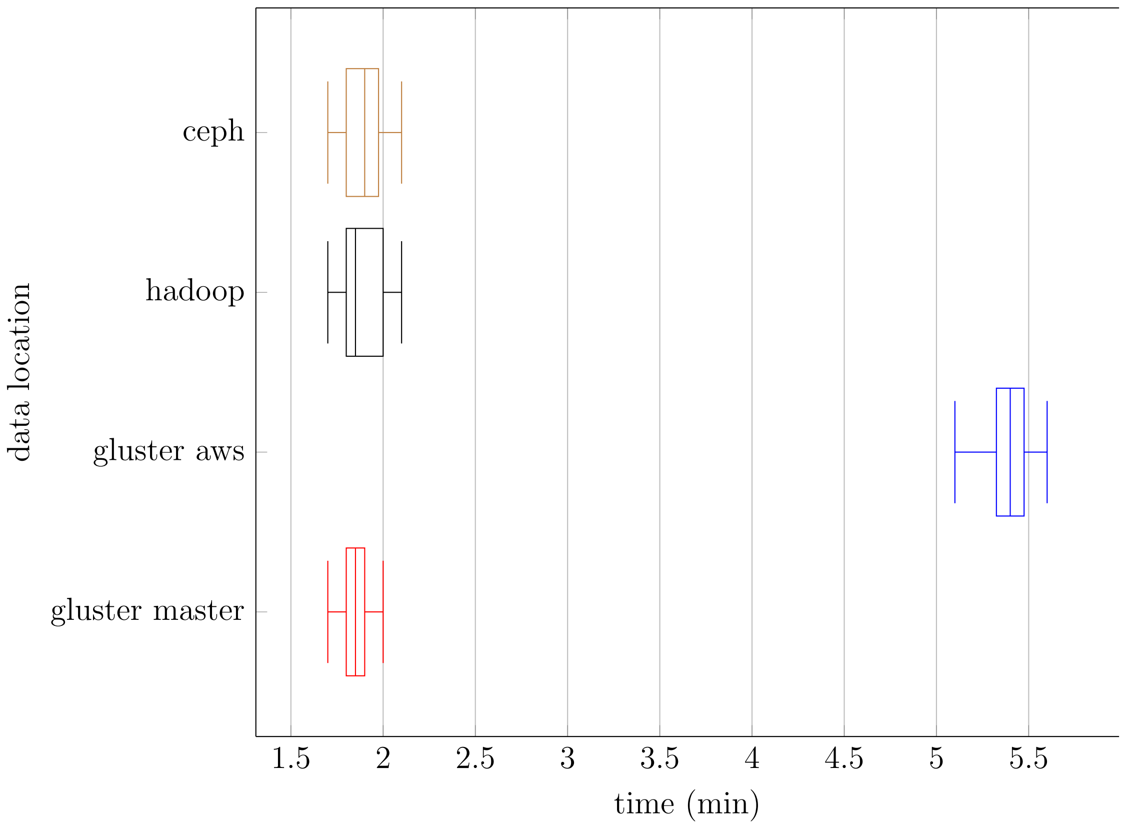

Figure 5.1: Single slave on master node

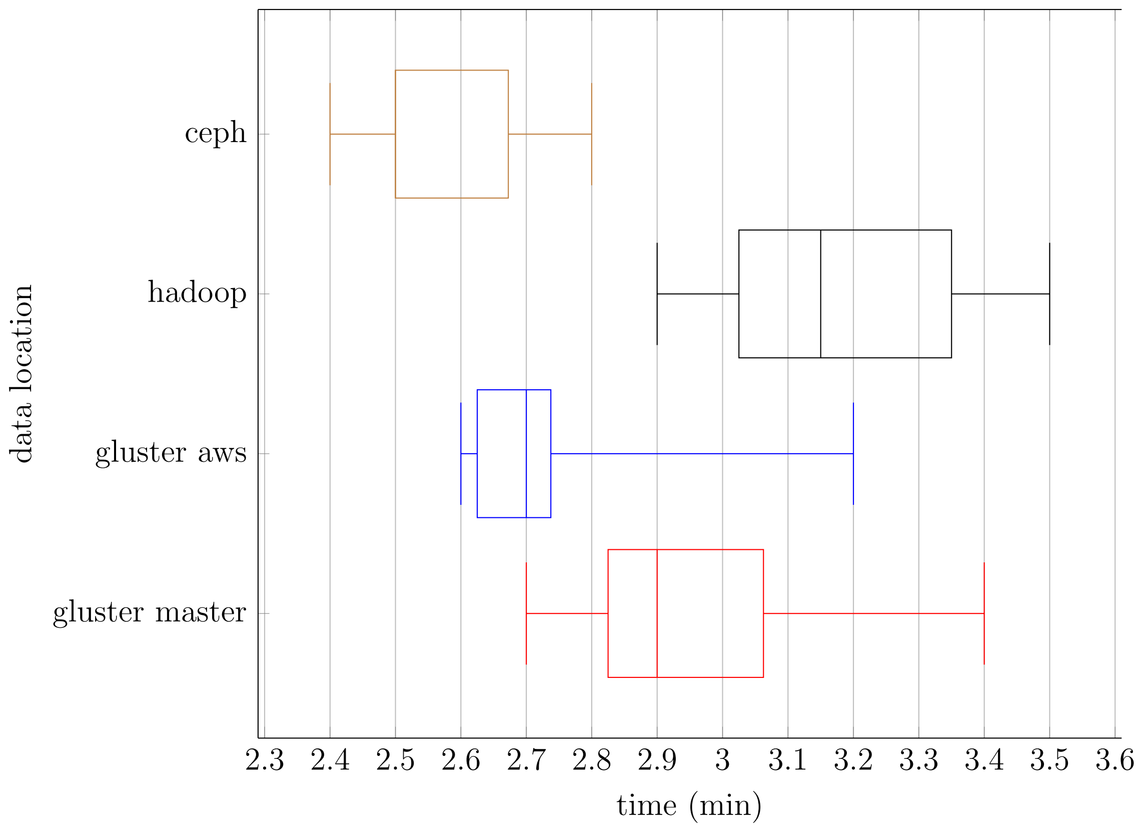

Figure 5.2: Single slave on Frankfurt 1 node

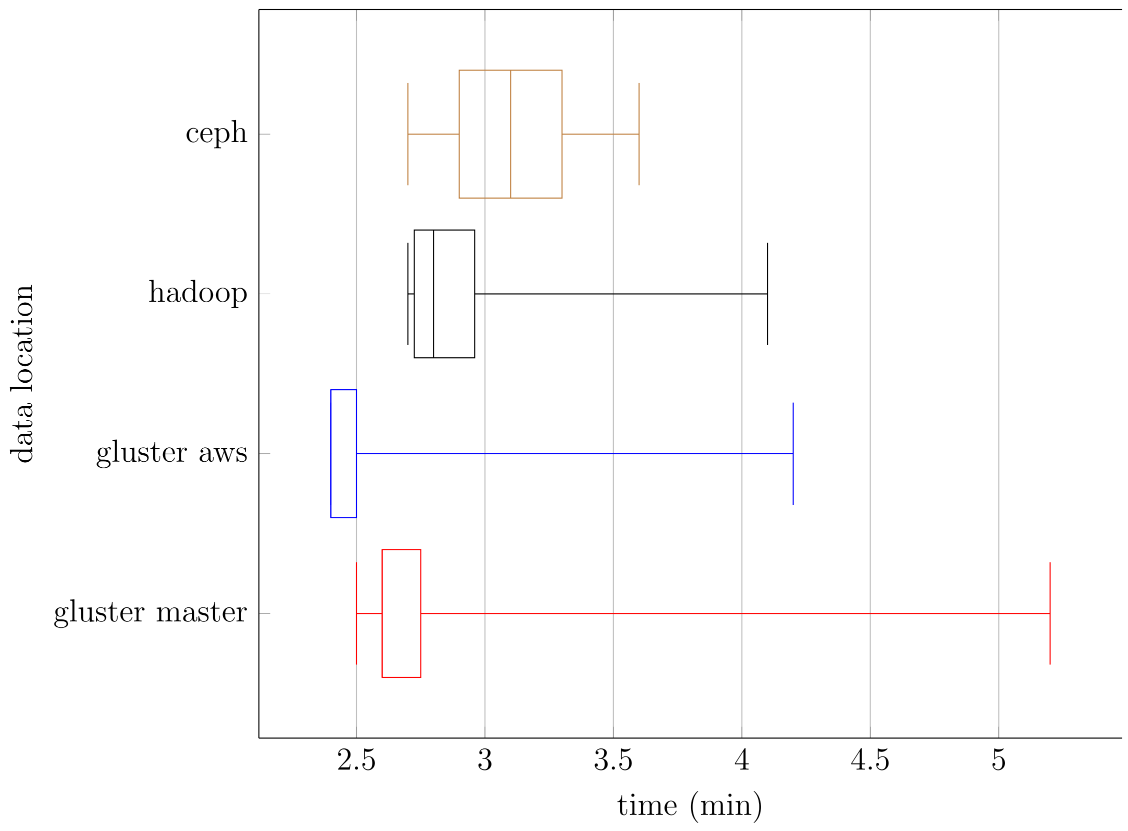

Figure 5.3: Single slave on Miami node

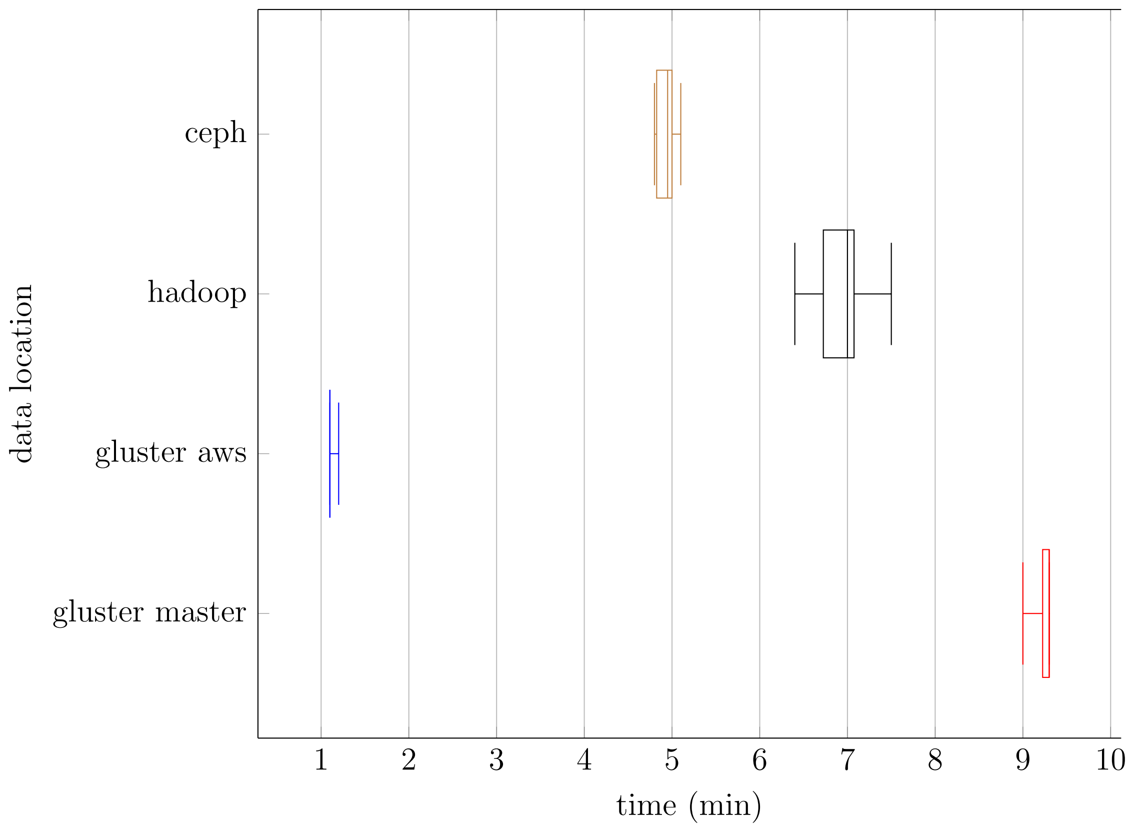

Figure 5.4: Single slave on AWS node

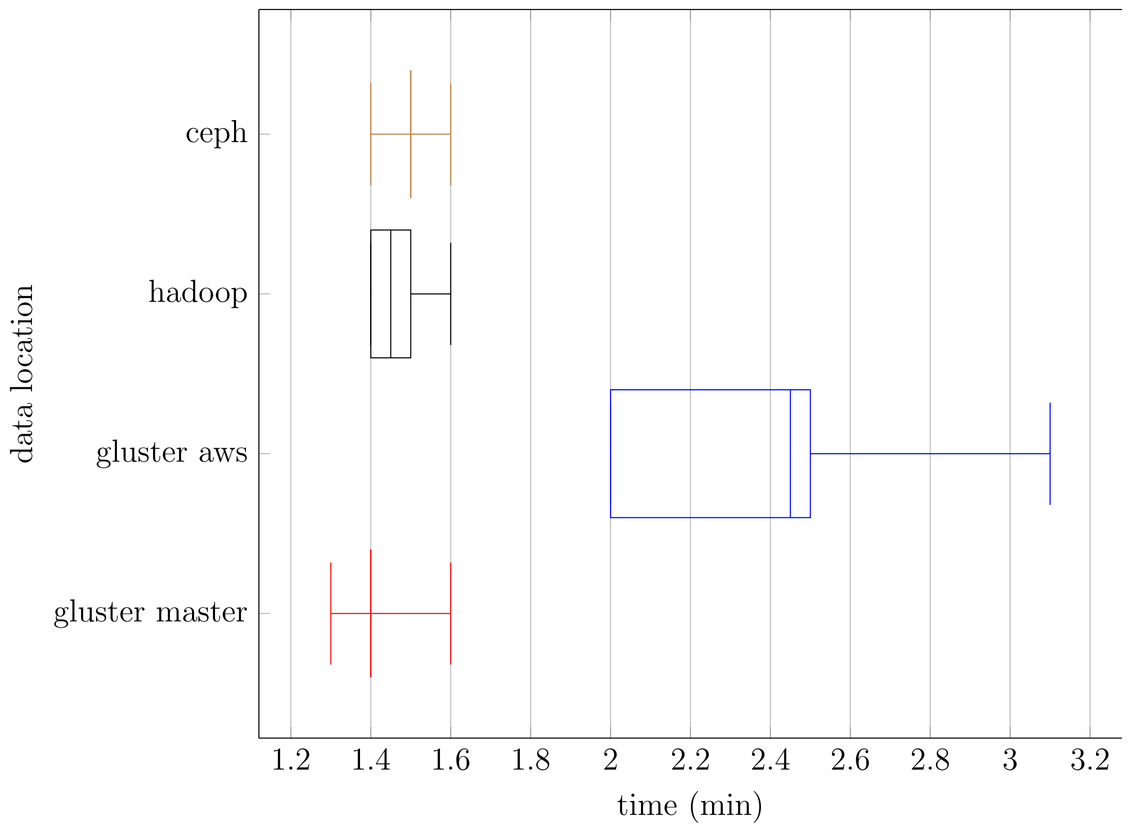

Figure 5.5: Two slaves in Frankfurt

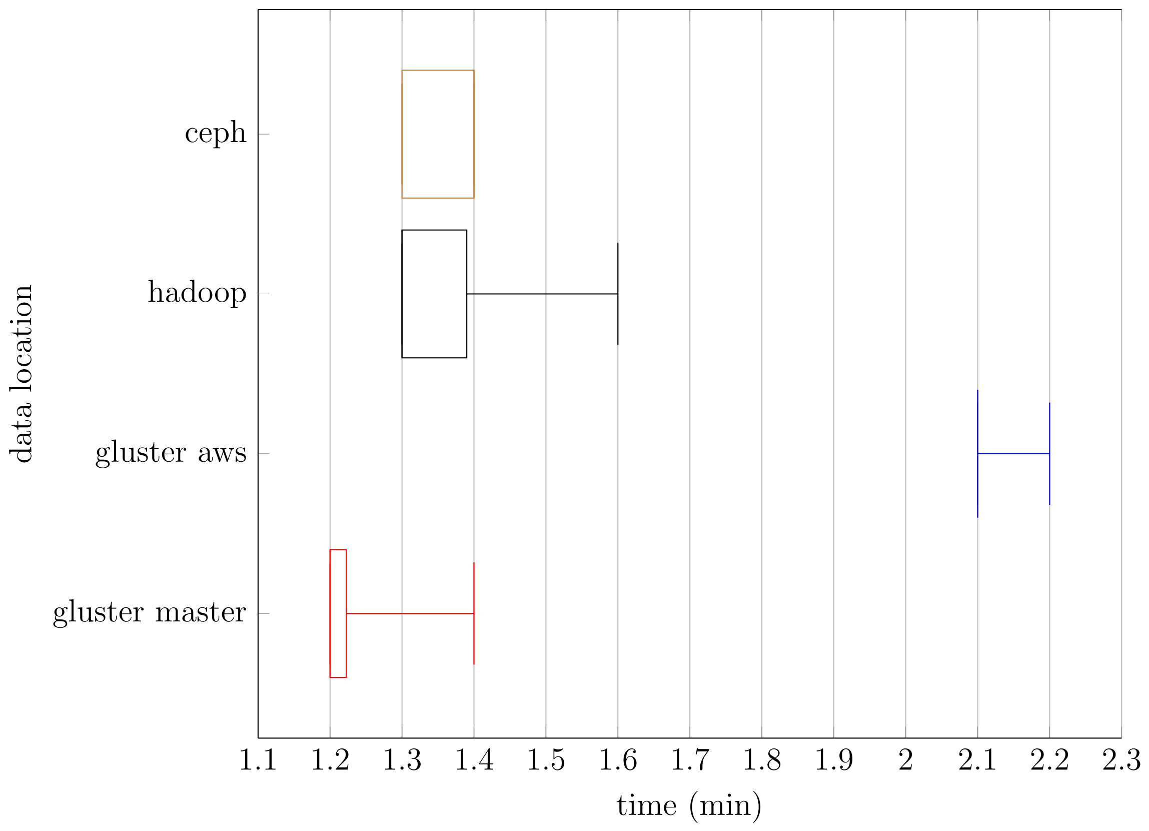

Figure 5.6: Three slaves in Frankfurt

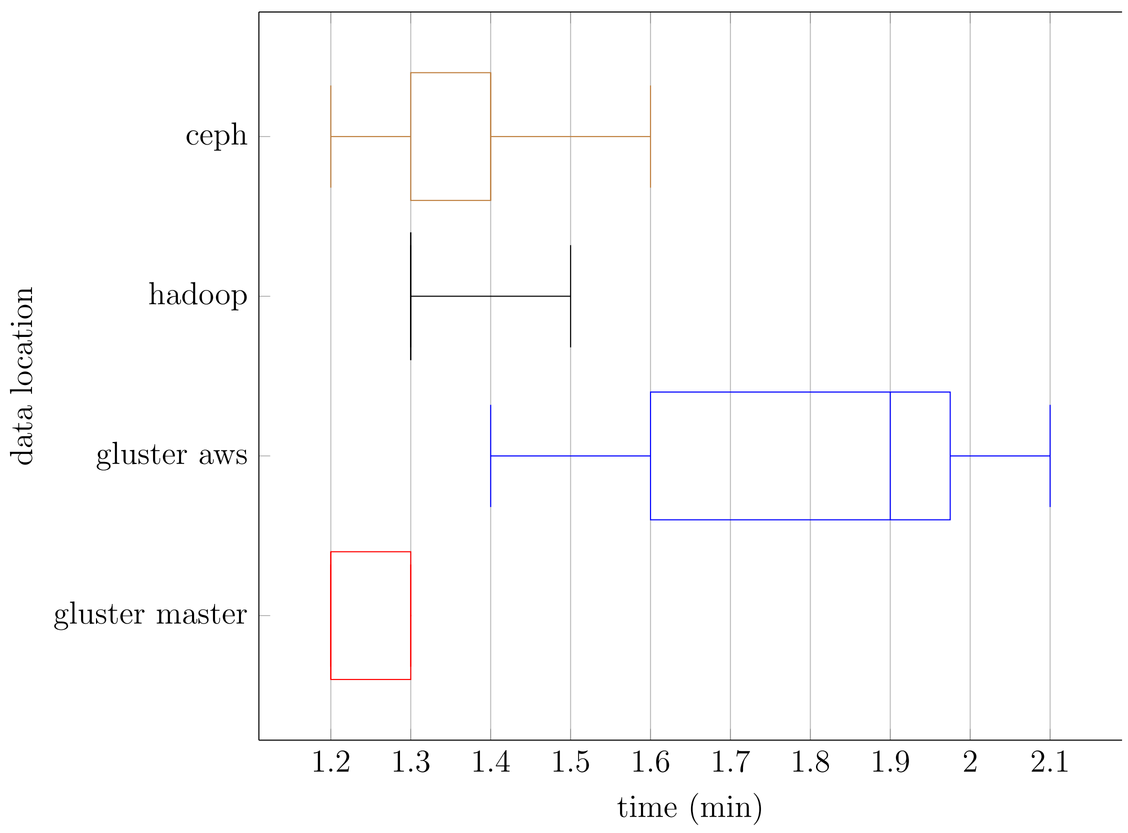

Figure 5.7: 4 slaves

We can see that the data location has a significant impact on the performance of the job. In fact, looking at the two files stored in GlusterFS -the master one located in Frankfurt and the AWS one located in Ohio, USA- it is evident that the closer the slave is to the data source, the quicker the job is. In fact the nodes located in America have better performance when reading from the AWS node, while the slaves in Frankfurt perform better with the Frankfurt data source.
The configuration with a Frankfurt slave (Figure 2), however, shows an anomaly, being faster reading from the AWS file.
Increasing the number of nodes we notice the expected results: with more Frankfurt nodes the performance difference between the data sources remains constant, while with the addition of the Miami node, in the test with 4 slaves,

we have a slight improvement of the computation speed when reading from the file in the AWS node, compared to the Frankfurt file, due to the proximity of the Miami node to the AWS data.
Moving on to the performance of the different filesystems, we notice that GlusterFS is quicker in every configuration (if the data is close to the slave), while CephFS and Hadoop have a slightly worse performance, similar between them, despite having both distributed the file across multiple nodes.
The configuration with the Frankfurt slave is an anomaly even in this case, since it performed better with CephFS.
The anomalies shown by the computations with the Frankfurt slave may be caused by a slow or unstable connection with the master node, or a different configuration of the virtual machine.

5.2.2 Results - Computation Movement

The following box plots summarize the tests relevant to the computation movement: the two graphs represent the tests performed reading from files located in the master node, in Frankfurt, and in the AWS node, in America, both stored in a GlusterFS volume.
Each box plot in the graph represents the test performed in a different spark configuration.

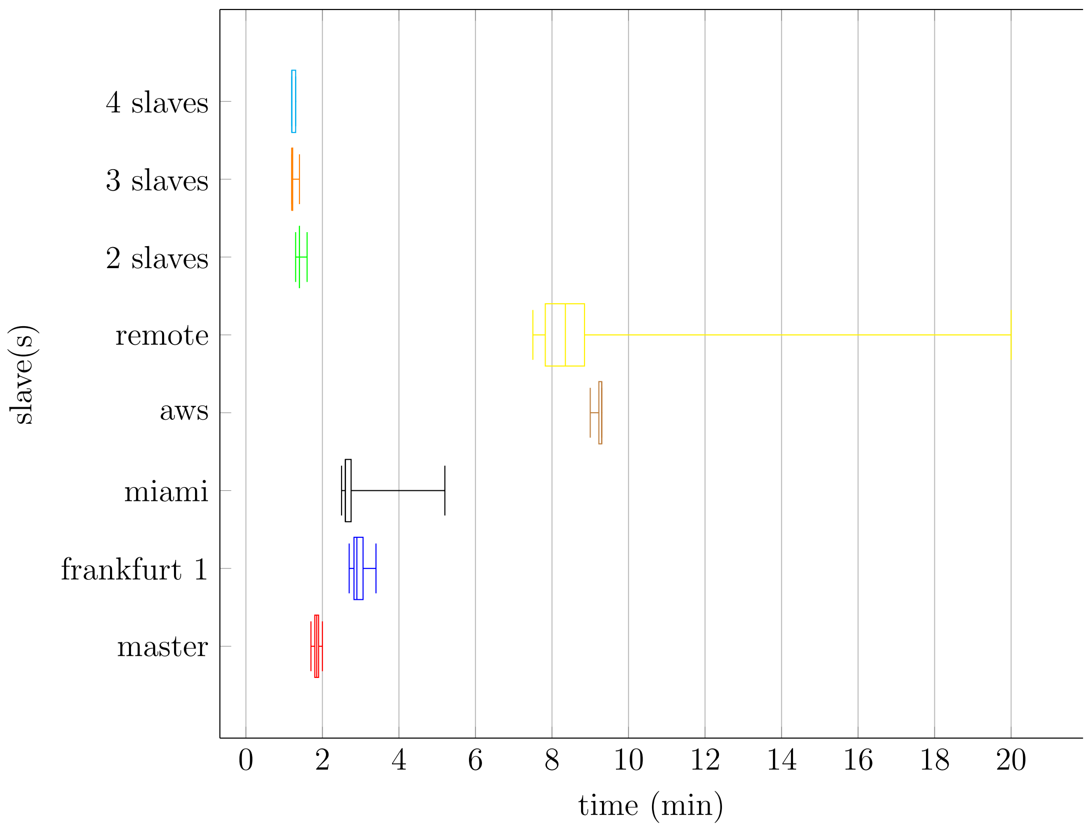

Figure 5.8: File stored in master node, Frankfurt, with GlusterFS

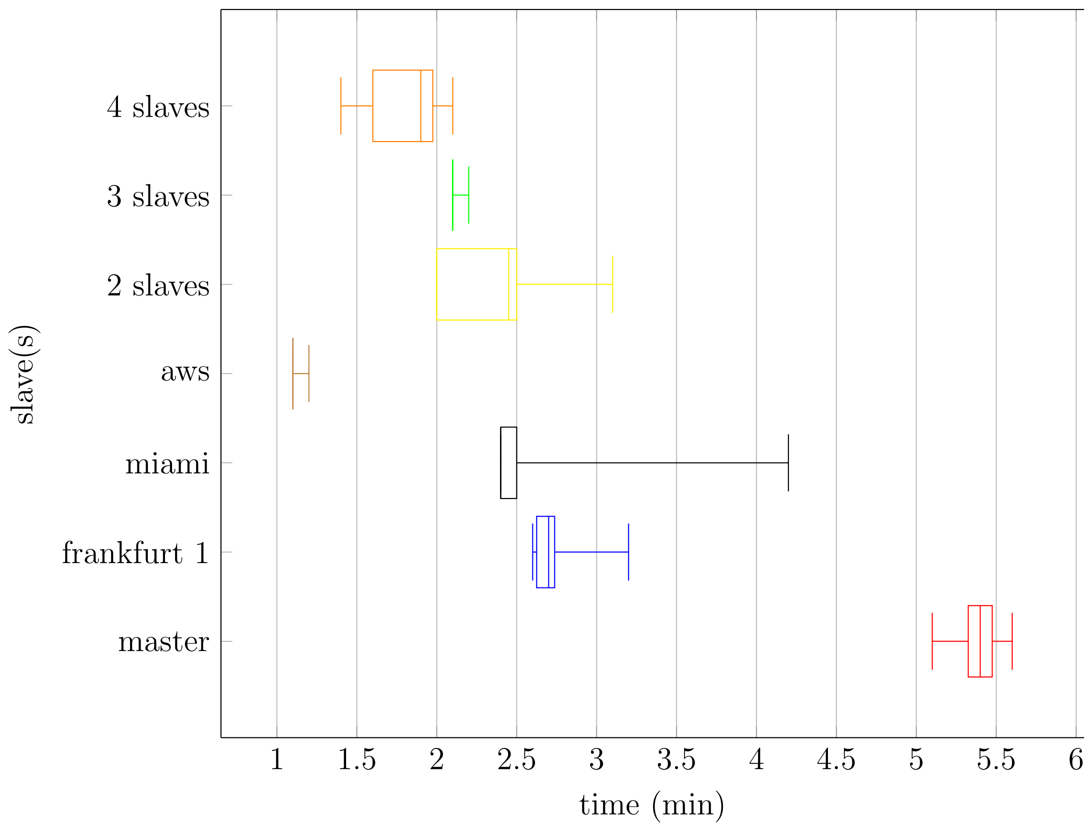

Figure 5.9: File stored in AWS node, USA, with GlusterFS

As already evaluated in the data source testing, we see that the slaves near the data source perform better than the further ones.
With the addition of a second slave, we notice a significant improvement in the computation time. When we add a third and a fourth slave, however, we do not see any more particular benefit, probably because the file size does not take fully advantage of the additional resources.
In the graph 5.9 the computation with 4 slaves is noticeably quicker than the computation with three slaves, but this happens due to the fact that the fourth slave is located in Miami, closer to the data source.

5.2.3 Results - Spark settings

These tests do not aim to evaluate either the Data or the Computation Movement, but are useful to understand how we can tune Spark to achieve better performance.
As mentioned previously in the document, we identified two particular settings relevant to the architecture, so we divided them in the following sections, the data partitioning and the number of CPUs.

Data partitioning

The following graph summarize the tests performed to evaluate how changing the number of partitions of the data affects the performance of a job.
We performed the tests in two scenarios: one with a single worker, located in Miami to emphasize the different given the longer computation time, and one with four slaves, to evaluate if Spark can perform a better distribution of the tasks.

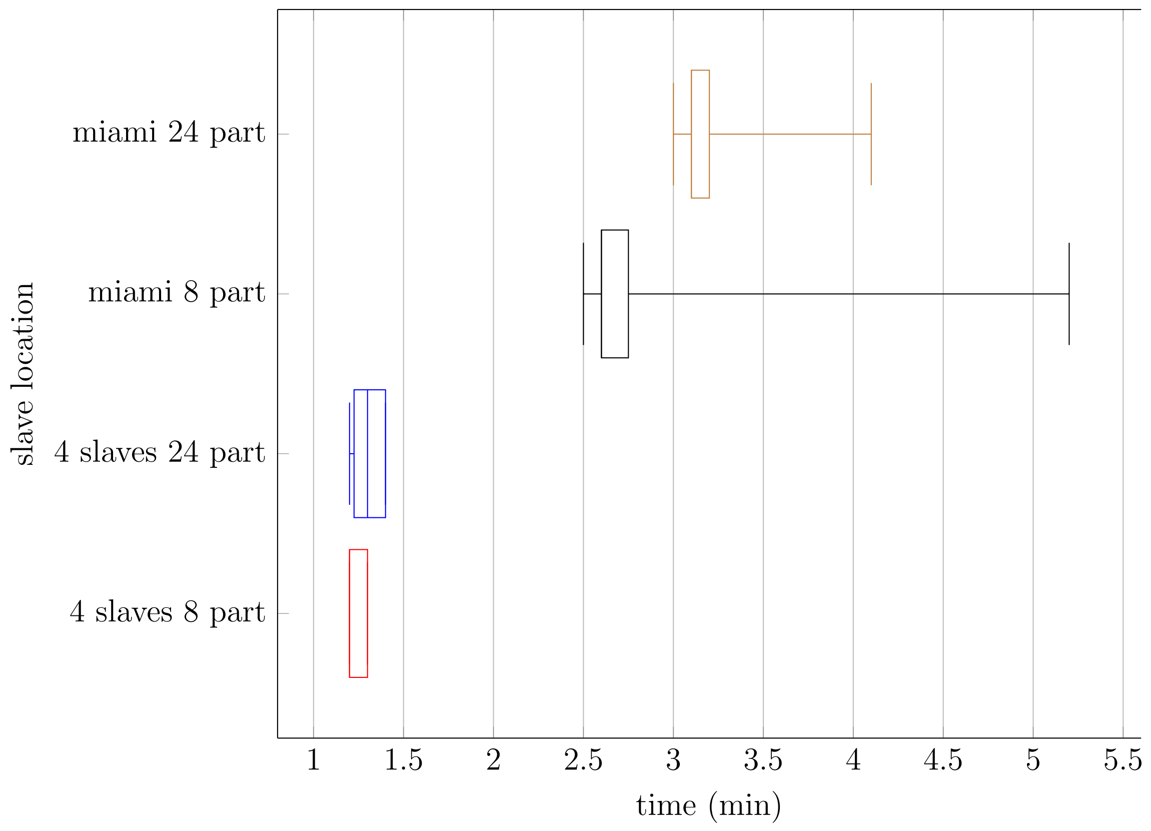

Figure 5.10: File stored in master node, Frankfurt, with GlusterFS

Both the tests with the Miami node and with the 4 nodes highlight that having an excessive number of partition causes a slight loss of performance, probably due to the fact that each slave has a single CPU, which prevents Spark to be able to compute more chunks of data in parallel.
We can then conclude that a number of partitions much higher than the total number of CPUs only adds an overhead, without giving any advantage in term of parallelism.
This results reflect the suggested partitioning for the data in a Spark job [7], which is to partition the data in about two/three times the number of total CPUs in the computation, while trying to maintain a small enough partition size to be stored in memory.

Number of CPUs

This test aim to calculate the influence that the number of CPUs has on the computation time.
Since Spark takes advantage of the parallelization of the tasks, having multiple CPUs per worker should yield a great advantage, similar to the addition of a different machine to the cluster.
The only machine suitable for the test is the remote one, since all the others are equipped with a single core, so we are limited to a single configuration of the cluster.

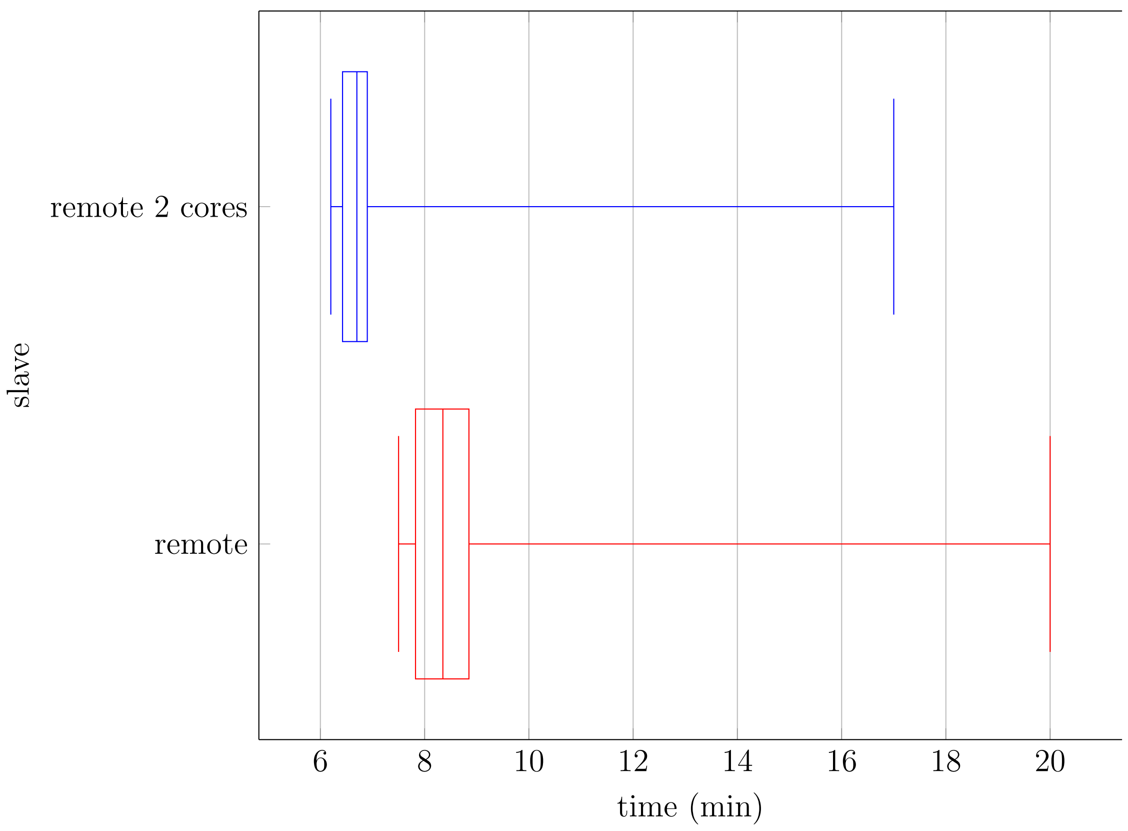

Figure 5.11: File stored in master node, Frankfurt, with GlusterFS

The two tests performed on the remote node show that, as it can be expected, adding more cores to the computation gives a performance boost, thanks to the higher parallelism of the tasks, although having double the

number of CPUs does not make the job two times faster.
The time needed to transfer the data, in this case more influential due to the slow internet connection of the remote node, combined with the initial start time of Spark, might mitigate the benefits of such upgrade.
However we notice that the performance increase is similar to the addition of a second slave (see section 5.2.2), so we can draw the conclusion that a cluster configuration with one two-core worker will perform similar to a configuration with two single-core workers.

5.3 Testing - Conclusions

From all the tests performed we can draw the following conclusions, relative to the Spark configuration and its application in Fog environments.
We can affirm that both the Data Movement and the Computation Movement are very effective in reducing the completion time of a job, in fact the best way to improve the performance is to move the computation near the data source, by choosing a close worker, or vice versa move the data near the computation, i.e. moving the file to the closest GlusterFS volume.
This effectiveness of the movements is encountered in every scenario, with every type of data source.
Adding more computation nodes can help, but only to a certain extent In fact, after a certain number of slaves, the benefits start to decrease.
The reasons may be different: the more scheduling and communication needed from the master can add an overhead greater than the benefits of an additional worker, the network capacity of the data source can act as a bottleneck and limit the performance, or the file is not large enough to fully take advantage of the increased resources.
Finally, to get the best performance, Spark needs to be set up with a couple of partitions per node, bearing in mind that each partition must fit in memory.

5.3.1 Data Movement

From the Data Movement point of view, these tests show that the data locality plays a fundamental aspect in the performance of a job.
The distribution of the data between multiple nodes however does not improve the performance, probably due to the added overhead of the communication between all the machines and to the file system.
Given these results, the Data Movement can be achieved with great performance benefits in this architecture by moving the file in a different GlusterFS volume, located close to the computation nodes.
The Data Duplication, although not tested, would simply consist in copying the file in different GlusterFS volumes, but it would make necessary a manual update of all the duplicates every time one of them is modified, since they are not synchronized.

5.3.2 Computation Movement

Reasoning from a Computation Movement perspective,it consists in a reconfiguration of the cluster, with the addition of the desired slaves prior to the computation.
The greatest improvement can be obtained by choosing the workers closer to the data source, brings the same benefits as the Data Movement.
The Computation Movement might be even more effective than the Data Movement for a single job: even if the worker is placed far from the master, the amount of data transferred between the two is typically far less than the total raw data, and thus the total time of the computation should be shorter than the time of the data transfer plus the computation after the Data Movement.
In case of multiple computation this argument does not hold anymore, since the initial Data Movement time is recouped in the following computations.
On the other hand, the Computation Duplication has to be accurately evaluated depending on the use case, since it emerged that the performance does

not increase linearly with the number of slaves, but it may depend on the data size, the network capacity and the data and computation distribution.

Chapter 6

Concluding Remarks

In this document we introduced the concept of Fog Computing and studied how to realize an implementation using Apache Spark, focusing on its two main paradigms: the Data Movement and the Computation Movement.
We performed different tests to evaluate the best way to configure the architecture and provide the desired Quality of Service.
Spark proved to be a suitable tool for our purposes, being easy and fast to set up, highly customizable and scalable, useful to implement the Computation Movement, and performed well with GlusterFS, which in turn was a great solution for the implementation of the Data Movement.
In order to implement the Data Movement we relied on GlusterFS, deploying volumes in different nodes and moving the file as needed. It emerges from the results that the data locality is highly relevant in the computation time of a job. In fact the best action to perform in order to speed up a job is to move the data near the workers of the cluster, while the distribution of the file among multiple nodes does not yield any significant improvement.
The Computation Movement is instead managed by the Spark cluster. Before the execution of a job, the needed nodes are added to the cluster. The best solution is to choose a worker node as near as possible to the data -which is the same principle of the data movement-, while a computation duplication seems to reach a limit after a while: adding more and more workers yields

to a progressively smaller advantage.
All this information will be useful in future works, which now have insight on the right tools to utilize, how to configure them, on what aspects it is advised to focus on, and what are the areas to be improved.
The next step is to design an architecture which will implement the Data Movement and the Computation Movement, in order to provide a decision system to the client, based on the job submitted.
Its goal will be to recognize the location of the data and the client, analyze the required resources and then perform the right movements to ensure the best quality of service.
To realize this system using Spark, it will be necessary to dive into its mechanics to deeply understand how the cluster manager assigns the tasks to the slaves, and exploit and manipulate this scheduling so that it is possible to move the computation at will.
Before the actual implementation however it can be useful to perform further testing in a more real-world scenario: it might be interesting to evaluate how this solution scales, with an higher number of machines, possibly with more resources, and how the architecture will handle a larger amount of data.
Other solutions can also be tested. Although Spark is the most popular framework for cluster computing, in this project we did not explore other possible candidates that may fit the requirements.
The way the data is provided to the cluster can be another topic to dive into. We researched only three different file systems, and we did not explore all their features, so there might be better solutions, or different configurations in the already tested ones.
In addition, the current data management solution, the GlusterFS volumes, does not allow to perform a Data Duplication that guarantees consistency between the different copies, so it is necessary to study a method to implement the synchronization between the volumes, or a completely new solution.
In this thesis only the Movement actions have been taken into consideration, but the goal-based model [10] takes into account also Data Transformations,

such as encryption and aggregation, so there is a need to find a way to provide such functionalities.

Bibliography

[1] Tariq Rahim Soomro Abdul Ghaffar Shoro. Big data analysis: Apache spark perspective. *Global Journal of Computer Science and Technology*, 2015.

[2] OpenFog Consortium. *OpenFog Reference Architecture for Fog Computing*, 2017.

[3] Apache Spark Documentation. Json files. `https://spark.apache.org/docs/latest/sql-data-sources-json.html`, 2019.

[4] Ceph Documentation. Installation (ceph-deploy) — ceph documentation. `http://docs.ceph.com/docs/mimic/start/`, 2016.

[5] Apache Foundation. Apache hadoop 2.9.2 – hadoop: Setting up a single node cluster. `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SingleCluster.html`, 2018.

[6] Sanjay Ghemawat Jeffrey Dean. Mapreduce: Simplified data processing on large clusters. Technical report, Google, Inc., 2004.

[7] Norbert Kozlowski. Partitioning in apache spark - parrot prediction - medium. `https://medium.com/parrot-prediction/partitioning-in-apache-spark-8134ad840b0`, 11 2017.

[8] N.Goren et al. M. Iorga. Fog computing conceptual model. Technical report, National Institute of Standards and Technology, 2018.

[9] Rahul Nayak. Set up apache spark on a multi-node cluster - y media labs innovation - medium. `https://medium.com/ymedialabs-innovation/apache-spark-on-a-multi-node-cluster-b75967c8cb2b`, 03 2018.

[10] M. Vitali P. Plebani, M. salnitri. Fog computing and data as a service: a goal-based modeling approach to enable effective data movements. Technical report, Politecnico di Milano, 2017.

[11] Jack Wallen. How to set up high availability storage with glusterfs on ubuntu 18.04 - techrepublic. `https://www.techrepublic.com/article/how-to-set-up-high-availability-storage-with-glusterfs-on-ubuntu-18-04/`, 08 2018.

[12] Ian Ward. Json lines. `http://jsonlines.org/`.

[13] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, 2012. USENIX Association.