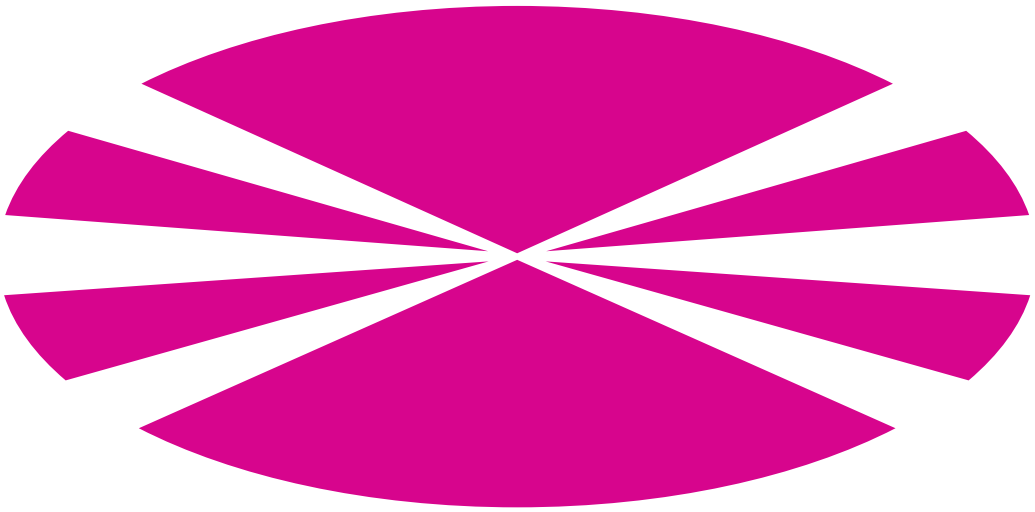

UNIVERSIDADE DA CORUÑA

SCHOOL OF INFORMATICS

Department of Electronics and Systems

Assessment, Design and Implementation of a Private Cloud for MapReduce Applications

Author: Marcos Salgueiro Balsa

Patricia González Gómez

Tutoring: Tomás Fernández Pena

José Carlos Cabaleiro Domínguez

Date: A Coruña, June 2013

Give a man a fish, and you'll feed him for a day.
Teach a man to fish, and you'll feed him for a lifetime.
Anne Isabella Thackeray Ritchie

Great spirits have always encountered violent opposition from mediocre minds.
Albert Einsten

The supreme art of war is to subdue the enemy without fighting.
Sun Tzu, *The Art of War*

[...] It takes these very simple-minded instructions – "Go fetch a number, add it to this number, put the result there, perceive if it's greater than this other number" – but executes them at a rate of, let's say, 1,000,000 per second. At 1,000,000 per second, the results appear to be magic.
Steven Paul Jobs

Summary

The history of computation has seen how the technology's unending evolution has promoted changes in its ways and means. Today, *tablets* and *smartphones*, quantitatively inferiors managing and memorizing numbers, camp freely in a global market saturated with options. The tendency is clear: users will get to use more than one device to access the Internet and will like to have all of their data synchronized and at hand, all the time.

But that is only a part in the equation. At the other side of every service request there lays a server that must deal with an ever increasingly troubling traffic volume, while it maintains response delivery at outstanding delay times — low latency "may" have helped the infant Google rise above the competition. If we also added that the idea of surrounding every implementation effort with energetic efficiency is a transcendental requisite and not simply a good practice, we would have a perfect environment for the proliferation of new distributed paradigms as the *Cloud*. The Cloud is not an intrinsically new idea but an old concept abstraction: *virtualization*. The clouds' cornerstone is flexibility.

Another technology that is constantly making it to the headlines is *MapReduce*. If the Cloud centers around easing infrastructure exploitation, MapReduce's core strength lies in its speeding up driving large masses of unstructured data; with makes them an extraordinary computational tandem. This project puts forth a solution that allows for drawing on computational resources available exploiting both technologies together. Special emphasis has been placed in flexibility of access, being a web browser the only application

required to use the service; in simplifying the virtual cluster configuration, by including a self-managed minimum deployment; and in transparency and extensibility, by freeing source code and documentation as *OSS*, favoring its usage as starting point for larger installations.

Keywords

Distributed Computing, Virtualization, Cloud Computing, MapReduce, OpenStack, Hadoop.

Contents

1 Abstract **1**

1.1 Goals 3

1.2 Arrangement of the Document 4

2 Background **5**

2.1 Cloud Computing 5

2.1.1 Architecture 7

2.1.2 Virtualization Techniques 10

2.1.3 Cloud IaaS frameworks 12

2.2 MapReduce Paradigm 13

2.2.1 Programming Model 13

2.2.2 Applicability of the Model 16

2.2.3 Processing Model 17

2.2.4 Fault Tolerance 19

2.2.5 Aditional Characteristics 20

2.2.6 Other MapReduce Implementations 22

Bibliography **30**

List of Figures

1.1 Demand in exponential growth. Source: *Cloudera Inc.* 2
1.2 Energy savings. Source: [1] 3

2.1 Layers in a cloud in production 6
2.2 Cloud Controller and Cloud Node 7
2.3 Cloud Controller in detail 9
2.4 Map function example (functional version) 14
2.5 Map function signature (MapReduce version) 14
2.6 Fold function example 15
2.7 Reduce function signature 15
2.8 MapReduce wordcount pseudocode. Source: [2] 16
2.9 MapReduce execution diagram. Source: [2] 18

Chapter 1

Abstract

Over the last years there has been a continuous increase in the quantity of information generated with the Internet as the main driver. Furthermore, this information has reshaped from structured — and thus, susceptible to being expressed following a relational schema — to heterogeneous, which has kick-started the necessity to alter the way it is stored and transformed. As the figure 1.1 shows, those that were the undisputed back-end queens — relational database systems mostly — are seeing how their role is fading away due to their incapability to efficiently save unrelated heterogeneity.

As another related dimension, in the year 2000 many .com companies started upgrading their data centers to accommodate the inexorable demand peak that was going to follow. But it never came; and the bubble burst. What happened then was general underutilization — only `10%` of Amazon's global computational resources were in use — that pushed the search for alternative means to export the surplus as a product. Amazon's own initiative unfolded in 2006 with the *AWS* (*Amazon Web Services*) appearance. AWS, among others, implements a public API for flexible on-demand infrastructure provisioning.

Since then, similar projects have proliferated generalizing how private clusters' unused computational capacity is to be serviced, trying to stay API-compatible with the AWS to facilitate interoperability and thus avoid client's

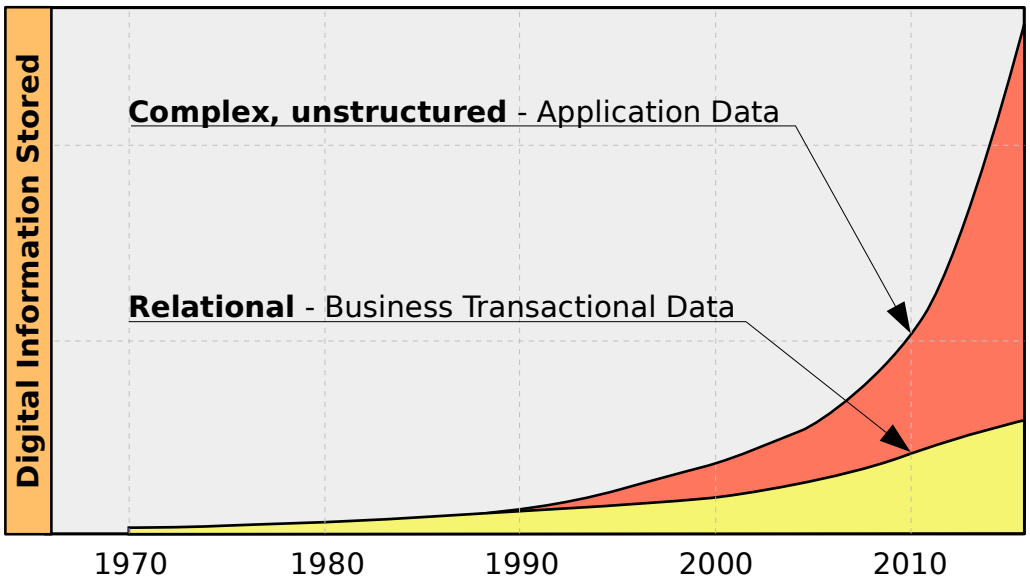

Figure 1.1: Demand in exponential growth. Source: *Cloudera Inc.*

swapping to more flexible providers.

Meanwhile, Google was also in the search for new mechanisms to exploit, with high performance and securely, their own private infrastructure to evolve the capability of their services. MapReduce, as a way to massively execute thousands transformations on input data, became a reality to thrust the generation of Google's humongous inverted index of the Internet [2]. Forthcoming contributions from Nutch's developers — by that time an Internet search engine prototype — to the MapReduce paradigm at *Yahoo!*, would traduce into the appearance of today's *de facto* standard in the field: Hadoop. Nowadays Hadoop is used in a myriad of backgrounds, ranging from travel booking sites to storing and servicing mobile data, ecommerce, image processing applications or searching for new forms of energy.

So, by stacking a MapReduce implementation atop elastic infrastructure an optimal exploitation of computational resources would be attainable, rapidly expanding or shrinking them on-demand, while simultaneously reducing the overall energy consumption required to accomplish processings

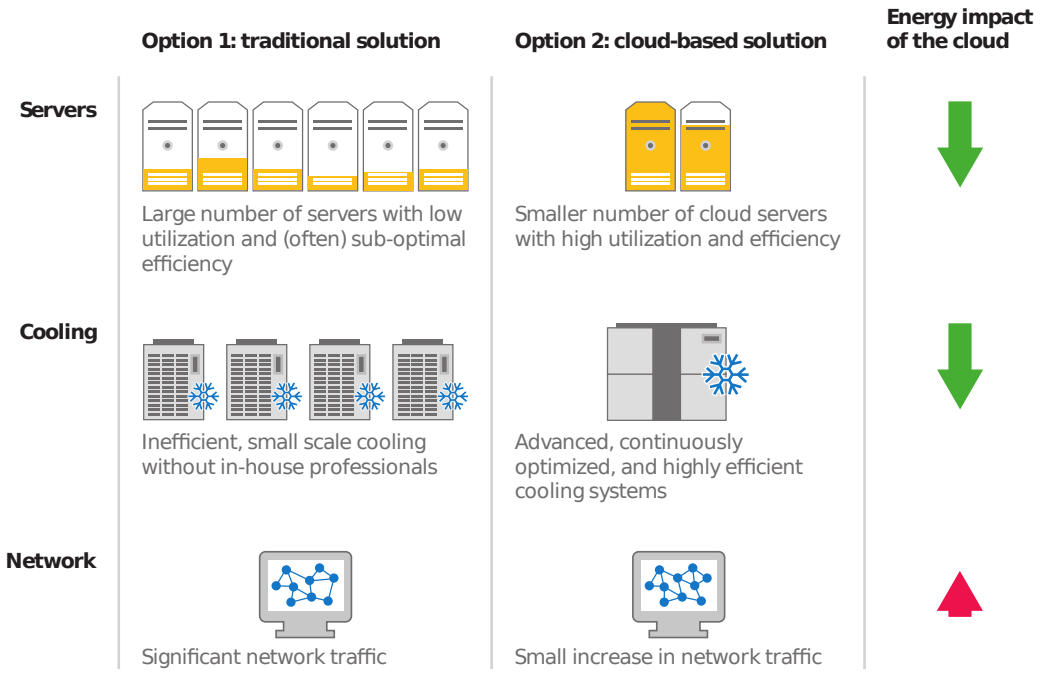

Figure 1.2: Energy savings. Source: [1]

(Figure 1.2).

1.1 Goals

The main goal with this project is to study the feasibility to develop a solution for a Cloud to drive MapReduce applications, with no need to know the particular Cloud structure and/or Hadoop configuration parameters.

In order to achieve such a simple execution model without compromising performance or applicability, a thorough analysis on different *IaaS Frameworks* will be carried out. Their features will be evaluated inside a virtual testing environment to finally narrow the selection to only one. Once an IaaS Framework had been chosen, the attention will be put towards choosing a MapReduce implementation to install over our virtual infrastructure.

Nonetheless, a mechanism to forward MapReduce execution requests will

be devised and implemented trying to focus on simplicity and universal access to this human-cloud-mapreduce interface. Yet, this transparency mustn't become an obstacle in exploiting the application or in fetching processed results. Privacy and security in communications and storage will be conveniently defined; we shan't forget it will be developed as a scaled-down model which could be infinitely scaled out.

1.2 Arrangement of the Document

The contents within this document are distributed as stated next. This first chapter introduces development guidelines in the abstract. Chapter 2 puts the reader closer to the fundamental Cloud Computing concepts — like its general architecture or virtualization —, along with the ones from the MapReduce paradigm. Chapter **??** describes an empirical evaluation of four private IaaS Cloud frameworks. Chapter **??** explores OpenStack Folsom's modular structure and particular inner workings. Analogically, chapter **??** unveils Hadoop's peculiarities as a MapReduce implementation.

The subsequent chapters center on detailing the project from diverse vantage points. Chapter **??** contains a series of design decisions and their accompanying UML diagrams. Chapter **??** gathers an analysis on performance in a real testing cluster. Chapter cap:conclusiones analyzes related papers highlighting how they compare to this solution. Finally, the main contributions of this project are discussed in addition to proposing future improvements to the implementation.

Two annexes have also been included. Annex **??** guides the reader throughout a quick single node installation. Annex **??** covers the definition of some of the concepts and technologies referred to in this text.

Chapter 2

Background

This second chapter tries to acquaint the reader with the key concepts that define Cloud Computing as well as the MapReduce archetype. Later successive elaborations to the project will lay on top of them.

2.1 Cloud Computing

In essence, Cloud Computing, or Cloud for short, is a distributed computing model that attempts to ease the consumption on-demand of that distributed infrastructure, by exporting it as virtual computational resources, platforms or services. However it may seem, the Cloud is no new technology but it introduces a new manner to exploit idle computing capacity. What it intends is to make orchestration of enormous data centers more flexible, so as to allow a user to start or destroy virtual machines as required — Infrastructure as a Service (*IaaS*) —, leverage a testing environment over a particular Operating System or software platform — Platform as a Service (*PaaS*) — or use a specific service like remote backup — Software as a Service (*SaaS*). Figure 2.1 shows the corresponding high level layer diagram of a generic Cloud.

Different IaaS frameworks will cover the functionality that is required to drive the cloud-defining *physical* infrastructure. Nonetheless, an effort to analyze, design, configure, install and maintain the intended service will

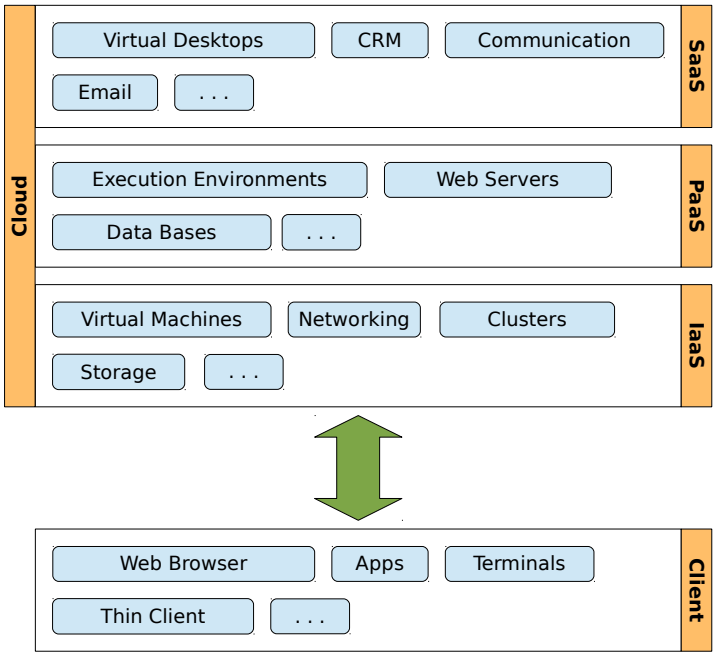

Figure 2.1: Layers in a cloud in production

be needed, bearing in mind that the degree of elaboration grows from IaaS services to SaaS ones. In effect, PaaS and SaaS layers are lied supported by those immediately under — software is implemented over a particular platform which, in turn, is also build upon a physical layer. Every Cloud Framework focuses on giving the option to configure a stable environment in which to run virtual machines defined by four variables: Virtual CPU count, virtual RAM, virtual persistent memory and virtual networking devices. Such an environment makes it possible to deploy virtual clusters upon which to install platforms or services to be subsequently consumed by users, bringing up the software layers that give form PaaS and SaaS paradigms respectively.

No less important cuestions like access control, execution permissions, quota or persistent or safe storage will also be present in all of the frameworks.

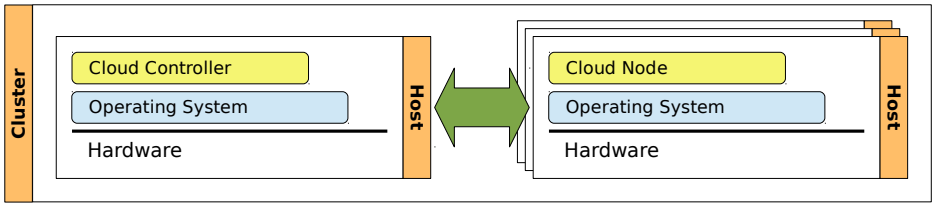

Figure 2.2: Cloud Controller and Cloud Node

2.1.1 Architecture

Figure 2.1 showed possible layers that could be found in a cloud deployment. Depending on the layers that are implemented, the particular framework and the role played by the cluster node, different particular modules will appear to make possible the consumption of configured services. These modules may be though of as Cloud subsystems that connect each one of the parts that are required to execute virtual machines. Those virtual machines' capabilities are defined by the four variables previously discussed — VCPUS, RAM, HDD and networking. As there is no methodology dictating how those subsystems should be in terms of size and responsibility, and thus, each framework makes its own modular partition regarding infrastructure management.

Setting modularity apart, one common feature among different clouds is the separation of responsibility in two main roles: *Cloud Controller* and *Cloud Node*. Figure 2.2 shows a generic Cloud deployment in a cluster with both roles defined. The guidelines followed for having this two roles lies close to *Master-Slave* architectures' approach. In those, in the abstract, there's a set of computers labeled as coordinators which are expected to control execution, and another set made up with those machines that are to carry out the actual processing.

Within this general role distribution in a cluster, host computers or cluster nodes — labeled as Cloud Controllers or Cloud Nodes — cooperate in a synchronized fashion through *NTP* (*Network Time Protocol*) and communicate via message passing supported by asynchronous queues. To store

services' metadata and status they typically draw upon a *DBMS* (*Data Base Management System*) implementation, which is regularly kept running in a dedicated cluster node set sharded (distributed) between the members of the set.

Although there is no practical restriction to configuring both Cloud Controller and Cloud Node within a single computer in a cluster, this approach should be limited to development environments due to the considerable impact in performance that it would carry.

Cloud Controller

The fundamental task for a Controller is to maintain all of the cloud's constituent modules working together by coordinating their cooperation. As an example, it is a Controller's duty to:

- Authentication and authorization control.
- Available infrastructure resources recount.
- Quota management.
- Usage balance.
- User and project inventory.
- API exposure for service consumption.
- Real time cloud monitoring.

Being an essential part of a cloud as it is, the Controller node (not to be mistaken for the Cloud Node) is usually replicated in physically distinct computers. Figure 2.3 shows a Cloud Controller's architecture from a high level perspective.

As a general rule, clients will interact with clouds through a Web Service API — mostly *RESTful* APIs (*REpresentational State Transfer*). Those APIs vary slightly from company to vendor as usual, which forces clients to

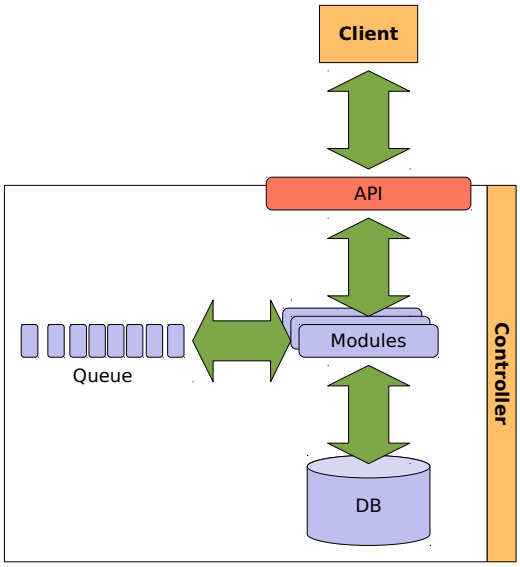

Figure 2.3: Cloud Controller in detail

be partially coupled to clouds. That is why there has been an increasing trend for unifying and standardizing those APIs in order to guarantee compatibility inter-framework. Of special mention is the cloud standard proposed by the *Open Grid Forum*: *OCCI* (*Open Cloud Computing Interface* [6]).

The cloud's conforming modules support its functional requirements. Each one of them will have a well-defined responsibility, and so appear networking modules, access and security control modules, storage modules, etc. Many of them existed before the advent of Cloud Computing but they worked only locally. Inter-module communication is handled by means of an asynchronous message queue that guarantees an equally efficient broadcasting system outside of the Cloud Controller, i.e. the rest of the cluster nodes participating in the cloud.

To store and expose configuration data to the cluster in a single place while managing concurrent requests to update these data, every IaaS Framework evaluated resorts to a DBMS whose profiling must be properly tailored.

Hardware requirements on the cluster nodes vary from each particular framework implementation and the *QoS* expected, but, in the abstract, they normally need something around 10 GB of RAM, quad core CPU, Gigabit Ethernet and one TB of storage.

Cloud Node

If the Cloud Controller is entrusted the cloud's correct functioning acting like a glue for its parts, the actual task processing is performed in the Cloud Nodes; that is, the VCPU, VRAM, VHDD are going to be mapped from the corresponding CPU, RAM and HDD from the real nodes of the cluster.

Cloud Nodes may be heterogeneous according to their hardware characteristics. They will configure a resource set that, seen from the outside of the cluster, will appear to be a homogeneous whole where the summation of capacities of every participating node is the cloud's dimension. Further, this homogeneous space could be provisioned, as discussed above, on demand. It is the Cloud Controller's responsibility single out the optimal distribution of virtual servers throughout the cluster, attending to the physical aspects of both the virtual machine and the computer in which the former will run.

The most important subsystem in a Cloud Controller is the *hypervisor* or *VMM* (*Virtual Machine Monitor*). The hypervisor is responsible for making possible the execution of virtual servers — or virtual instances following the AWS nomenclature — by creating the virtual architecture needed and a *virtual execution domain* managed with the help of the operating system kernel. To generate this architecture there fundamentally exist three techniques: *Emulation*, *Paravirtualization* and *Hardware Virtualization* or *Full Virtualization*. Different hypervisors will support them in a different degree, but most will cover only one of them.

2.1.2 Virtualization Techniques

What follows is a brief review of the main methods to create virtual infrastructure.

Emulation

Emulation is the most general virtualization method, in a sense that it does not call for anything special be present in the underlying hardware. However, it also carries the highest penalization in terms of performance. With emulation, every structure sustaining the virtual machine operation is created as a functional software copy of its hardware counterpart; i. e., every machine instruction to be executed in the virtual hardware must be run software-wise first, and then be translated on the fly into another machine instruction runnable in the physical domain — the cluster node. The interpreter implementation and the divergence between emulated and real hardware will directly impact the translation overhead. This fact hinders the emulation from being widely employed in performance-critical deployments. Nonetheless, thanks to its operating flexibility it's generally used as a mechanism to support legacy systems. Besides, the kernel in the guest operating system — the kernel in the virtual machines's — operating system needs no alteration whatsoever, and the cluster node's kernel need only load a module.

Hardware Virtualization

Hardware Virtualization, on the contrary, allows host's processes to run directly atop the physical hardware layer, with no interpretation. Logically, this provides a considerable speedup from emulation, though imposes a special treatment to be given to its virtual processes. Regarding CPUs, both AMD's and Intel's support virtual process execution — which is the capacity to run processes belonging to the virtual domain with little performance overhead — as far as the convenient hardware extensions are present (*SVM* and *VT-x* respectively [7]). Just as what happened with emulation, an unaltered host's kernel may be used. This fact is of relative importance as if wasn't so it would limit the myriad of OSs that could be installed as guests. Lastly, it should be pointed out that the hardware architecture is exposed to the VM as it is, i. e. with no software middleware.

Paravirtualization

Paravirtualization uses a different approach. To begin with, it is indispensable that the guest's kernel be modified to make it capable of interacting with a paravirtualized environment. When the guest runs, the hypervisor will separate those regions of instructions that have to be executed in kernel mode in the CPU, from those in user mode which will be executed as regular host processes. Subsequently, the hypervisor will manage an on-contract execution between host and guest allowing the latter to run kernel mode sections as if pertaining to the real execution domain — as if they were processes running in the host, not in the guest — with almost no performance slowdown. Paravirtualization, in turn, does not require an special hardware extension be present.

2.1.3 Cloud IaaS frameworks

Cloud IaaS frameworks are those software systems managing the abstraction of complexity associated with on demand provisioning and administering failure-prone generic infrastructure. In spite of being almost all of them open sourced — which fosters reusability and collaboration —, they have evolved in different frames. This fact has raised a condition of lacking outwards interoperability, maturing non-standard APIs; though today those divergences are fading away. These frameworks and APIs are product of the efforts to improve and ease controlling the underlying particular clusters on which they germinated. Thus, it is no surprising their advances had originated parallelly with the infrastructure they drove, leaving compatibility in the background.

Slowly but steadily these managing systems became larger in reach and responsibility boosted by an increasingly interest in the sector. In the end, it happened that software and systems engineering made them more abstract, so they finally overlapped functionally. AWS appearance finished forging the latent standardization need, and thus, as of today, most frameworks offer APIs closer and closer to Amazon's — nowadays the de-facto standard —

and OCCI's [6].

2.2 MapReduce Paradigm

The origin of the paradigm centers around a paper publication of two Google employees [2]. In this paper they explained a method implementation devised to abstract the common parts present in distributed computing that rendered simple but large problems much more complex to solve when paralleling their execution on massive clusters. A concise definition states that MapReduce is "*a data processing model and execution environment that runs on large clusters of commodity computers*" [8].

2.2.1 Programming Model

The MapReduce programming model requires the developer express his problem as a partition of two well-defined pieces. A first part deals with the reading of input data and with producing a set of intermediate results that will be scattered over the cluster nodes. These intermediate transformations will be grouped according to an intermediate key value. A second phase begins with that grouping of intermediate results and concludes when every *reduce* operation on the groupings succeeds. Seen from another vantage point, the first phase corresponds, broadly speaking, to the behavior of the functional *map* and the second to the functional *fold*.

In terms of the MapReduce model, these functional paradigm concepts give rise to *Map* and *Reduce* functions. Both Map and Reduce have to be supplied by the developer, which may force a deviation in breaking the original problem down. As counterpart, the MapReduce model will deal with parallelizing the computation, distributing input data across the cluster, handling exceptions that could raise and recovering output results; everything transparent to the programmer.

$$\begin{array}{|l|}\hline \textbf{map} : (\alpha \rightarrow \beta) \ \rightarrow \ \alpha\, list \ \rightarrow \ \beta\, list \\ \textbf{map} \ (\textbf{pow}\ 2) \ [1,2,3] \ \Rightarrow \ [1,4,9] \\ \hline \end{array}$$

Figure 2.4: Map function example (functional version)

$$\begin{array}{|l|}\hline \textbf{map} : (k1, v1) \ \rightarrow \ (k2, v2)\, list \\ \textbf{k} : clave \\ \textbf{v} : valor \\ (\textbf{kn}, \textbf{vn}) : par \ (clave, valor) \ en\ un\ dominio\ n \\ \hline \end{array}$$

Figure 2.5: Map function signature (MapReduce version)

Función Map

The typical functional map takes any function F and a list of elements L or, in general, any recursive data structure, to return a list resulting from applying F to each element of L. Figure 2.4 shows its signature and an example.

In its MapReduce realization, map function receives a tuple as input and produces another tuple *(key, value)* as intermediate output. It is the MapReduce library who is responsible for feeding the map function by mutating the data contained in input files into *(key, value)* pairs. Then, it deals with grouping those intermediate tuples by key before passing them in as input to the reduce function. Input and output data types correspond to those shown in the function signature figure 2.5.

Reduce function

The typical functional fold expects any function G, a list L, or generally any type of recursive data structure, and any initial element I, subtype of L's elements. Fold returns the value in I resulting from building up the intermediate values generated after applying G to each element in L. Figure 2.6 presents fold signature as well as an example.

$$\textbf{fold} : (\alpha \to \beta \to \alpha) \ \to \ \alpha \ \to \ \beta \, list \ \to \ \alpha$$
$$\textbf{fold} \ (+) \ 0 \ [1, 2, 3] \ \Rightarrow \ 6$$

Figure 2.6: Fold function example

$$\textbf{reduce} : (k2, v2 \, list) \ \to \ v2 \, list$$
$$\textbf{k} : clave$$
$$\textbf{v} : valor$$
$$(\textbf{kn}, \textbf{vn}) : par \ \ (clave, valor) \ \ en \ un \ dominio \ n$$

Figure 2.7: Reduce function signature

Contrary to map, reduce expects the intermediate groups as input to produce a smaller set of values for each group as output, because reduce will iteratively *fold* the groupings into values. Those reduced intermediate values will be passed in again to the reduce function if more values with the same key appeared from subsequent maps. Reduce signature is shown on figure 2.7. Just as happens with map, MapReduce handles the transmission of intermediate results out from map into reduce. The model also describes the possibility to define a *Combiner* function that would act after map partially reducing the values within the same grouping to lower network traffic — the combiner usually runs in the same machine as the map.

A word counter in MapReduce

As an example, figure 2.8 shows the pseudocode of a MapReduce application to count the number of words in a document set.

In a wordcount execution flow the following is going to happen: map is going to be presented with a set of names containing all of the documents in plain text whose words will be counted. Map will subsequently iterate over each document in the set emitting the tuple *(<word>, "1")* for each word found. Thus, an explosion of intermediate pairs will be generated as

```
Map (String key, String value):
// key: documento name
// value: document contents
for each word w in value:
EmitIntermediate (w, "1");

Reduce (String key, Iterator values):
// key: a word
// values: an Iterable over intermediate counts of the word key
int result = 0;
for each v in values:
Emit (AsString (result));
```

Figure 2.8: MapReduce wordcount pseudocode. Source: [2]

output of map, will be distributed over the network and progressively folded in the reduce phase. Reduce is going to be input every pair generated by map but under a different form. Reduce will accept on each invocation the pair *(<word>, list("1"))*. The list of *"1"*s, or generically an `Iterable` over *"1"*s, will contain as many elements as instances of the word *<word>* there were in the document set — this supposing that the map phase were over before starting the reduce phase and that every word *<word>* were submitted to the same reducer in the cluster — a cluster node executing the reduce function.

Once the flow had been completed, MapReduce would return a listing with every word in the documents and the number of times it appeared.

2.2.2 Applicability of the Model

The myriad of problems that could be expressed following the MapReduce programming parading is clearly reflected in [2], a subset of them being:

- Distributed grep: Map emits every line matching the regular expres-

sion. Reduce only forwards its input to its output acting as identity function.

- Count of URL access frequency: Like wordcount.

- Reverse web-link graph: For each URL contained in a web document, map generates the pair *(<target_URL>, <source_URL>)*. Reduce will emit the pair *(target, list(source))*.

- Inverted index: Map parses each document and emits a series of tuples in the form *(<word>, <document_id>)*. All of them are passed as input to reduce that generates the sequence of pairs *(<word>, list(document_id))*.

2.2.3 Processing Model

Besides defining the structure that the applications willing to leverage the MapReduce capabilities will have to follow — so that they need not code their own distribution mechanisms —, with [2] an implementation of the model was introduced which allowed Google to stay protocol, architecture and system agnostic while keeping their commodity clusters on full utilization. This agnosticism allows for deploying vendor-lock-free distributed systems.

The MapReduce model works by receiving self-contained processing requests called *job*s. Each job is a *partition* of smaller duties called *task*s. A job won't be completed until no task is pending for finishing execution. The processing model main intent is to distribute the tasks throughout the cluster in a way that reduced job latency. In general, it can be stated that task processing on each phase is done in parallel and phases execute in sequence; yet, it is not needed for reduce to wait until map is complete.

Figure 2.9 shows a summary of a typical execution flow. It is interesting enough to deepen in its details as many other MapReduce implementations will present similar approaches.

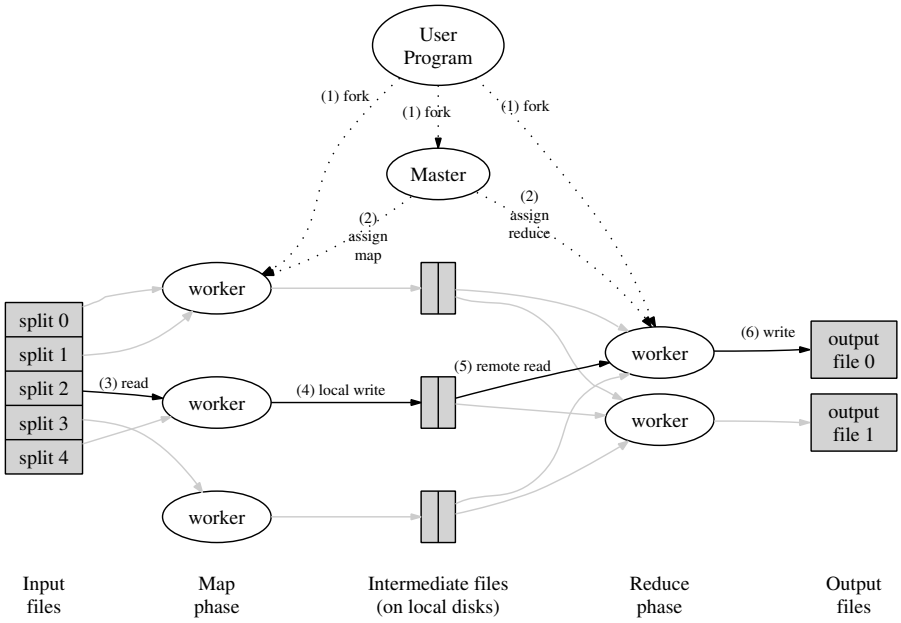

Figure 2.9: MapReduce execution diagram. Source: [2]

1. MapReduce divides input files in M parts, the size of which is parameterized, and distributes as many copies of the MapReduce user algorithm as nodes participate in the computation.

2. From this moment each program copy resides in a cluster node. A random copy is chosen among them and labeled as the *Master Replica*, effectively assigning the *Master Role* to the node holding the replica; every other node in the cluster is designated with the *Worker Role*. Those worker nodes will receive the actual MapReduce tasks and their execution will be driven from the master node. There will be M map tasks and R reduce tasks.

3. Workers assigned with map tasks read their corresponding portions of the input files and parse the contents generating tuples *(key, value)* that will be submitted to the map function for processing. Map outputs are stored in memory as a cache.

4. Periodically, those pairs in memory are dumped to a local disk — dumped to a drive of the node that is executing the map function — and partitioned into R regions. Their path on disk is then sent back to the master, responsible for forwarding these paths to *reduce workers* or *reducer*s.

5. Now, when a reducer is notified that it should start processing, the path to the data of the reduction is send along and the reducer will fetch them directly from the mapper via *RPC* (*Remote Procedure Call*). Before actually invoking the reduce function, the node itself will sort the intermediate pairs by key.

6. Lastly, the reducer iterates over the key-sorted pairs submitting to the user-defined reduce function the key and the `Iterable` of values associated to the key. The output fo the reduce function for the reducer partition is appended to a file stored over the distributed file system.

When every map and reduce tasks had succeeded, the partitioned output space — the file set within each partition — would be returned back to the client application that had made the MapReduce invocation.

This processing model is abstract enough as to be employed to the resolution of indeterminatedly large problems running on huge clusters.

2.2.4 Fault Tolerance

The idea of providing an environment to execute jobs long enough to require large sets of computing machines to keep the latency within reasonable timings, calls for the definition of a policy able to assure a degree of tolerance to failure. If unattended, those failures would lead to errors; some would cause finished tasks to get lost, others would put intermediate data offline. Consequently, if no measures were taken to prevent or deal with failure, job throughput would humble as some would have to be rescheduled all along.

The MapReduce model describes a policy foreseeing a series within an execution flow and duly implements a series of actions against them.

Worker Failure

The least taxing of the problems. To control that every worker is up, the master node pings them periodically. If a worker did not reply to pings repeatedly, it would be marked as failed.

A worker marked failed will neither be scheduled new tasks nor will be remotely accessed by reducers to load intermediate map results that it may had; a fact that could prevent the workflow from succeeding. If so were the case, the access to these data would be resolved by the master labeling the results of the failing tasks as *idle*, so that they could be rescheduled a later time to store the results in an active worker.

Master Failure

Failure of a master node is more troublesome. The proposed approach consists in making the master periodically create a snapshot from which to restore to a previous state if it went unexpectedly down. It is a harder problem than a worker failure mainly because there can only be one master per cluster, and the time it would take another node to take over the master role would leave the scheduling pipeline stalled. The master being in a single machine has also the benefit of lowering the probability of failure, precisely why in the original paper [2] it had been put forward that the entire job be canceled. Still, as there is no good design to leave a *single point of failure*, subsequent MapReduce implementations have proposed to replicate the master in other nodes in the same cluster.

2.2.5 Aditional Characteristics

What follows is a summary of additional features of the original MapReduce implementation.

Locality

The typical bottleneck in a modern deployment is network bandwidth. In MapReduce executions, the information flows into the cluster from the external client. As already discussed, each node in a MapReduce cluster holds a certain amount of the input data and shares its processing capacity to be used for particular MapReduce tasks over those data. Each stage in the MapReduce executing pipeline requires a lot of traffic to be handled by the network which would reduce throughput if no wide enough channel were deployed nor a locality exploiting strategy were implemented.

In fact, MapReduce explores a method to use locality as an additional resource. The idea is for the distributed file system to place data as close as possible to where they will transformed — it will try to store data in the mappers' and reducers' local drives —, effectively diminishing the transport over the net.

Complexity

A priori, variables M and R, the number of partitions of the input space and of the intermediate space respectively, may be configured to take any value whatsoever. Yet, there exist certain practical limits to their values. For every running job the master will have to make $O(M + R)$ scheduling decisions — if no error forced the master to reschedule tasks —, as each partition of the input space will have to be submitted to a mapper and each intermediate partition will have to be transmitted to a reducer, coming to $O(M + R)$ as the expression of *temporal complexity*. Regarding *spatial complexity*, the master will have to maintain $O(M \cdot R)$ as piece of state in memory as the intermediate results of a map task may be propagated to every piece R of the reduce space.

Backup Tasks

A situation could arise in which a cluster node be executing map or reduce tasks much slower than it theoretically could. Such a circumstance may arise with a damaged drive which would cause read and write operations to slow down. And since jobs complete when all of its composing tasks had been finished, the faulted node (*the straggler*) would be curbing the global throughput. To alleviate this handicap, when few tasks are left incomplete for a particular job, *Backup Task*s are created an submitted to additional workers, making a single task be executed twice concurrently. By the time one copy of the task succeeds it will be labeled completed, duly reducing the impact of stragglers at the cost of wasting computational resources.

Combiner Function

Many times it happens that there exists a good number of repeated intermediate pairs. Taking wordcount as an example, it can be easily seen that every mapper will generate as many tuples *("a", "1")* as *a*'s there are in the input documents. A mechanism to lower the tuples that will have to be emitted to reducers is to allow for the definition of a *Combiner Function* to group outputs from the map function — and in the same mapper node — before sending them out over the network, effectively cutting down traffic.

In fact, it is usual for both combiner and reduce functions to share the same implementation, even though the former writes its output to local disk while the latter writes directly to the distributed file system.

2.2.6 Other MapReduce Implementations

Since 2004 multiple frameworks that implement the ideas exposed in the paper [2] have been coming out. The next listing clearly shows the impact MapReduce has created.

Hadoop [8] One of the first implementations to cover the MapReduce processing model and framework of reference to other MapReduce codifi-

cations. It is by far the most widely deployed, tested, configured and profiled today.

GridGain [9] Commercial and centered around in-memory processing to speedup execution: lower data access latency at the expense of smaller I/O space.

Twister [10] Developed as a research project of the University of Indiana, tries to cut out and abstract the common parts required to run MapReduce workflows in order to keep them longer in the cluster's distributed memory. With such an approach, the time taken to configure mappers and reducers in multiple executions is lowered by doing the set up only once. Thus, *Twister* really shines in executing *iterative* MapRecude jobs — those jobs where maps and reduces do not happen in sequence once, but need instead a multitude of complete map-reduce cycles to succeed.

GATK [11] Used for genetic research to sequence and evaluate DNA fragments from multiple species.

Qizmt [12] Written in C# and deployed in MySpace.

misco [13] Written 100% Python and based on previous work at Nokia it is posed as a MapReduce implementation capable of running in mobile devices.

Peregrine [14] By optimizing how intermediate results are transformed and by passing every I/O operation throughout an asynchronous queue, its developers claim to have formidably accelerated task execution rate.

Mars [15] Implemented in *NVIDIA CUDA*, it revolves around extracting higher performance by moving the map and reduce operations into the graphic card. It is supposed to improve processing throughput by over an order of magnitude.

Hadoop is undoubtedly the MapReduce implementation more used nowadays. Its open source nature and its flexibility, both for processing and storing, have constantly reported back an increasing interest from the IT industry. This has brought out many pluggable extensions that enhance Hadoop's applicability.

Bibliography

[1] Google Apps: Energy Efficiency in the Cloud. `http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/en/us/green/pdf/google-apps.pdf`. Accedido: junio 2013.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *Common. ACM*, 51(1):107–113, 2008.

[3] Pierre Riteau, Ancuta Iordache, and Christine Morin. Resilin: Elastic MapReduce for Private and Community Clouds. Research Report RR-7767, INRIA, October 2011.

[4] Huan Liu and Dan Orban. Cloud MapReduce: A MapReduce Implementation on Top of a Cloud Operating System. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 464–474, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Steve Loughran, Jose Maria Alcaraz Calero, Andrew Farrell, Johannes Kirschnick, and Julio Guijarro. Dynamic Cloud Deployment of a MapReduce Architecture. *IEEE Internet Computing*, 16(6):40–50, November 2012.

[6] Andy Edmonds, Thijs Metsch, Alexander Papaspyrou, and Alexis Richardson. Toward an Open Cloud Standard. *IEEE Internet Computing*, 16(4):15–25, July 2012.

[7] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, May 2005.

[8] Tom White. *Hadoop, the Definitive Guide*. O' Reilly and Yahoo! Press, 2012.

[9] Nikita Ivanov. GridGain and Hadoop: Differences and Sinergies. `http://www.gridgain.com/blog/gridgain-hadoop-differences-synergies/`. Accedido: junio 2013.

[10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *The First International Workshop on MapReduce and its Applications*, 2010.

[11] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. The Genome Analysis Toolkit: A MapReduce Framework for Analyzing Next-generation DNA Sequencing Data. 2010.

[12] Quizmt project web page. `http://qizmt.myspace.com/`. Accedido: junio 2013.

[13] Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, and Ville H. Tuulos. Misco: a Mapreduce Framework for Mobile Systems. In *Proceedings of the 3rd International Conference on Pervasive Technologies Related to Assistive Environments*, PETRA '10, pages 32:1–32:8, New York, NY, USA, 2010. ACM.

[14] Peregrine project web page. `http://peregrine_mapreduce.bitbucket.org/`. Accedido: junio 2013.

[15] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[16] *Eucalyptus — 3.1.1 Installation Guide*, 2012.

[17] *CloudStack Basic Installation Guide for CloudStack Version 3.0.0 — 3.0.2*, 2012.

[18] Citrix Unveils Next Phase of Cloudstack Strategy. `http://www.citrix.com/news/announcements/apr-2012/citrix-unveils-next-phase-of-cloudstack-strategy.html`, 2012. Accedido: junio 2013.

[19] How to Use CloudStack without Hardware Virtualization. `http://support.citrix.com/article/CTX132015`, 2012. Accedido: junio 2013.

[20] *Apache CloudStack 4.0.0 — Incubating CloudStack Installation Guide*, 2012.

[21] *CloudStack Advanced Installation Guide for CloudStack Version 3.0.0 — 3.0.2*, 2012.

[22] *Citrix XenServer 6.0 Installation Guide*, 2012.

[23] OpenNebula 3.8.1 QuickStart. `http://wiki.centos.org/Cloud/OpenNebula/QuickStart`, 2012. Accedido: junio 2013.

[24] Getting Started with Openstack on Fedora 17. `http://fedoraproject.org/wiki/Getting_started_with_OpenStack_on_Fedora_17`, 2012. Accedido: junio 2013.

[25] *OpenStack Install and Deploy Manual — Red Hat — Folsom*, 2012.

[26] *OpenStack Network (Quantum) Administration Guide — Folsom*, 2012.

[27] *OpenStack Object Storage Administration Manual — Folsom*, 2012.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[29] *Amazon Elastic Compute Cloud — User Guide — API Version 2012-12-01*, 2013.

[30] QEMU Internals. `http://qemu.weilnetz.de/qemu-tech.html`, 2012. Accedido: junio 2013.

[31] Jaromír Hradílek, Douglas Silas, Martin Prpič, Stephen Wadeley, Eliška Slobodová, Tomáš Čapek, Petr Kovář, John Ha, David O'Brien, Michael Hideo, and Don Domingo. *Fedora 17 System Administrators' Guide. Deployment, Configuration and Administration of Fedora 17.* Red Hat Inc., 2012.

[32] Christopher Curran and Jan Mark Holzer. *Red Hat Enterprise Linux 5.2 – Virtualization Guide.* Red Hat Inc., 2008.

[33] Michael Hideo Smith. *Red Hat Enterprise Linux 5.2 — Deployment Guide.* Red Hat Inc., 2008.

[34] Johan De Gelas. Hardware Virtualization: the Nuts and Bolts. `http://www.anandtech.com/show/2480`, 2012. Accedido: junio 2013.

[35] *OpenStack Install and Deploy Manual — Red Hat — Essex*, 2012.

[36] *OpenStack Compute Administration Manual — Essex*, 2012.

[37] *OpenStack Compute Administration Manual — Folsom*, 2012.

[38] Jacek Artymiak. *Programming OpenStack Compute API.* Rackspace US Inc., 2012.

[39] *OpenStack API Reference*, 2013.

[40] OpenNebula OCCI Specification 3.8. `http://opennebula.org/documentation:archives:rel3.8:occidd`, 2012. Accedido: junio 2013.

[41] DEISA. Deisa glossary. `http://www.deisa.eu/references`. Accedido: junio 2013.

[42] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`, 2011. Accedido: junio 2013.

[43] *CloudStack API Documentation (v3.0)*, 2012.

[44] Simon Kelley. dnsmasq — A Lightweight DHCP and Caching DNS Server — ManPage. `http://www.thekelleys.org.uk/dnsmasq/docs/dnsmasq-man.html`, 2012. Accedido: junio 2013.

[45] HDFS Users' Guide. `http://hadoop.apache.org/docs/r1.0.4/single_node_setup.html`, 2010. Accedido: junio 2013.

[46] Single Node Setup. `http://hadoop.apache.org/docs/r1.0.4/single_node_setup.html`, 2008. Accedido: junio 2013.

[47] MapReduce Tutorial. `http://hadoop.apache.org/docs/r1.0.4/mapred_tutorial.html`, 2008. Accedido: junio 2013.

[48] Cluster setup. `http://hadoop.apache.org/docs/r1.0.4/hdfs_user_guide.html`, 2008. Accedido: junio 2013.

[49] Dhruba Borthakur. HDFS Architecture Guide. `http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html`, 2012. Accedido: junio 2013.

[50] Jimmy Lin and Chris Dyer. *Data–Intensive Text Processing with MapReduce*. 2010.

[51] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A Framework for Large Scale Data Processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM.

[52] Ahmed Metwally and Christos Faloutsos. V-SMART-join: A Scalable MapReduce Framework for All-pair Similarity Joins of Multisets and Vectors. *Proc. VLDB Endow.*, 5(8):704–715, April 2012.

[53] Amr Awadallah. Apache Hadoop in the Enterprise — Keynote. Cloudera Inc., 2011.

[54] Mendel Cooper. Advanced Bash-Scripting Guide. An In-depth Exploration of the Art of Shell Scripting. `http://tldp.org/LDP/abs/html/`, 2012. Accedido: junio 2013.

[55] Bruce Barnett. Sed — An Introduction and Tutorial. `http://www.grymoire.com/Unix/Sed.html`, 2012. Accedido: junio 2013.

[56] MapReduce Wikipedia entry. `http://en.wikipedia.org/wiki/MapReduce`, 2012. Accedido: junio 2013.