ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación:

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

Título del Proyecto:

CLOUD COMPUTING KEEPS FINANCIAL METRICS COMPUTATION SIMPLE

Alumna: Ainhoa Azqueta Alzúaz

Tutores: José Enrique Armendáriz Íñigo
Pablo Murta Baião Albino

Pamplona, 28 Julio 2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays the volume of data, process and users of Internet have increased rapidly. All people would like to store any kind of files like photos, films or files with a determinate structure among other. The problem appeared when we do not have enough space to store these files due to the fact that we can not have a large number of hard disks in our property. So the solution to this problem is a paradigm that is in continuous change called *Cloud Computing* [16]. This paradigm allows the clients pay to the company that offers that service depending of the resources they used; this is also known as the “pay-as-you-go” model. Thus you can contract depending of the necessities you have.

We are talking about hundreds of terabytes to store that could be used for different purposes. Let us see this with an example, we can use stored data to compute some statistics about a feature(i.e., the mean value of a certain measure). This made several companies to develop their own systems to share their. Due to that this companies started to develop their own systems to handle all their data but they were so expensive to buy and hold. Furthermore, the tools that we had to make search of information in the databases not provide optimum performance to that kind of applications. We are talking about operations on a large scale search, aggregation and binding of multidimensional data schemas. Because of that the highest internet companies (Amazon [6], Google [8], Microsoft [15], Yahoo! [5], etc.) started to redesign massive data storage and in order to build scalable systems. Thus, lead to the definitions of *Cloud Computing* paradigm.

The first company that started to deal with this problem was Google with designed and implemented the Google File System [8], a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients. Google File System shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. To process and generate all these large data sets that are stored in the Google File System, Google designed MapReduce [9] that is a programming model and an associated implementation to help us to manage all these data. The user has to define a map and a reduce function, the first one processes a key/value pair to generate a set of intermediate key/value pairs and the second one merges all intermediate values associated with the same intermediate key. We talk about this process later in Chapter 2.

However, there are other private companies that develop this project on their own like Microsoft with Microsoft Azure, Amazon with EC2 or Yahoo! with PNUTS as we previously said. It is worth noting that there are open source software projects, like Apache Software Foundation [3], that develop equivalent architectures like Hadoop [4, 20], it is a collection of related subprojects that fall under the umbrella of infrastructure for distributed computing. But Hadoop is best known for MapReduce and Hadoop Distributed File System (HDFS) which are inspired in MapReduce of Google and Google File system respectively. To understand properly what Hadoop is we have to explain all the subprojects in which it is divided [20]:

Common The common utilities that support the other Hadoop subprojects, before called core.

Avro is a data serialization system that provides rich data structures; a compact, fast, binary data format; a container file, to store persistent data; remote procedure call; and simple integration with dynamic languages.

Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The salient property of Pig programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets. At present time. Pig's infrastructure layer consists of a compiler that produces sequences of MapReduce programs, for which large-scale parallel implementations already exist (e.g., the Hadoop subproject).

HBase is the Hadoop database. Is an open-source, distributed, versioned, column-oriented store modeled after Google' Bigtable: A Distributed Storage System for Structured by Chang et al. Just as Bigtable leverages the distributed data storage provided by the Google File System, HBase provides Bigtable like capabilities on top of Hadoop.

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time there are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

Hive is a data warehouse system for Hadoop that facilitates easy data summarization,adhoc queries, and the analysis of large datasets stored in Hadoop compatible file systems. Hive provides a mechanism to project structure onto this data and query the data using a SQL-like language called HiveQL. At the same time this language also allows traditional MapReduce programmers to plug in their custom mappers and reducers wher it is inconvenient or inefficient to express this logic in HiveQL.

Chukwa is an open source data collection system for monitoring large distributed systems. Chukwa is built on top of the HDFS and MapReduce framework and inherits hadoop's scalability and robustness. Chuckwa also includes a flexible and powerful toolkit for displaying, monitoring and analyzing results to make the best use of the collect data.

Cassandra develops a highly scalable second-generation distributed database, bringing together Dynamo's fully distributed design and Bigtable's ColumnFamily-based data model.

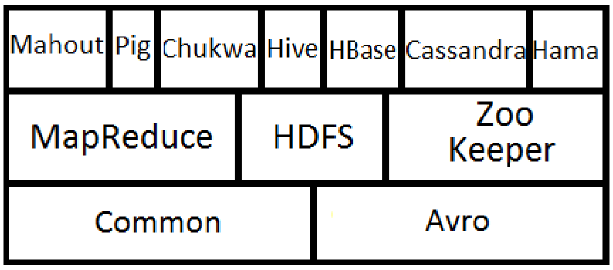
Hama is a distributed computing framework based on BSP (Bulk Synchronous Parallel) computing techniques for massive scientific computations, Currently being incubated as one of the incubator project by the Apache Software Foundation.

Mahout its goal is to build scalable machine learning libraries. With scalable they mean:

scalable to reasonably large data sets, scalable to support your business case and scalable community.

To have a best point of view about all these subprojects we can use the next Figure 1.1 in which we can watch all of them and in with appeared the HDFS in the middle and MapReduce on the left hand of it due to that HDFS is the pillar of all these project:

Figure 1.1: Hadoop Structure



Source: Own.

The main goal of this work is to apply the use of a dynamic distributed system tool specialized in processing huge informations data sets. We are going to use MapReduce to perform some mathematical computations on a data storage that contains some accounting information. In particular, we decided to use a Spanish and Portuguese database called *Sistema de Análisis de Balances Ibéricos* (SABI), which we will discuss later in Chapter 3.

This work focuses on the financial database SABI [19], which is considered as a basic research tool by many Spanish universities [11, 1, 2] and is largely used by private companies to perform market analysis. Although this repository constitutes the main financial information source in Spain, many companies do not properly fill in all the field, leading to an incomplete data panel. In some works, this issue is roughly solved by excluding those rows that belong to companies with missing values or inconsistent data [11]. Nonetheless, this data repository is targeted to engage researchers in analyzing companies' efficiency [13, 17, 10], computing ratios like indebtedness, availability of idle resources, or capital costs [14], among others.

As we think that is tool could be so interested due to we realized some accounting computations on a large database like(SABI) and it could be so useful as we explain previously we are going to explain the conclusion that we obtain with this project and who we could continuous improving this technique in Chapter 5.

Chapter 2

System Model

In this chapter we are going to explain the structure of HDFS, MapReduce and finally we are going to show who MapReduce Programs works with an example using a weather dataset.

2.1 The Hadoop Distributed FileSystem (HDFS)

To understand what HDFS is, we have to know that the distributed filesystems are filesystems that manage the storage across a network of machines. HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters on commodity hardware.

2.1.1 Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. The HDFS structure is based on a set of blocks that are large compared to disk blocks due to minimize the cost of seeks. The main difference between a disk block and a HDFS block is the size, the disk block size is about 512 bytes and the default HDFS block size is 64 Megabytes although some HDFS installation use 128 Megabytes. The files in HDFS are broken into block-sized chunks, which are stored as independent units.

Having a block abstraction for a distributed filesystem brings several benefits. The first is that a file can be larger than any single disk in the network, and the second is that making the unit of abstraction a block rather than a file simplifies the storage subsystem.

Furthermore, blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client. A block that is no longer available due to corruption or machine failure can be replicated from their alternative locations to other live machines to bring the replication factor back to the normal level [20].

2.1.2 Namenodes and Datanodes

An HDFS cluster has two types of node operating in a master-worker pattern: a namenode (the master) and a number of datanodes (workers, slaves).

The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

Datanodes are the work horses of the filesystem. They store an retrieve blocks when they are told to, and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the fileystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there datanodes. For this reason, it is important to make the namenode resilient to namenode resilient to failure, and Hadoop provides two mechanisms for this.

The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.

It is also possible to run a secondary namenode, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary data, loss is almost guaranteed. The usual course of action in this case is to copy the namenodes metadata files that are on NFS to the secondary and run it as the new primary [20].

2.1.3 Interfaces

Hadoop is written in Java, and all Hadoop filesystem interactions are mediated through the Java API. Hadoop has several interfaces but there are two interfaces that are specific to HDFS:

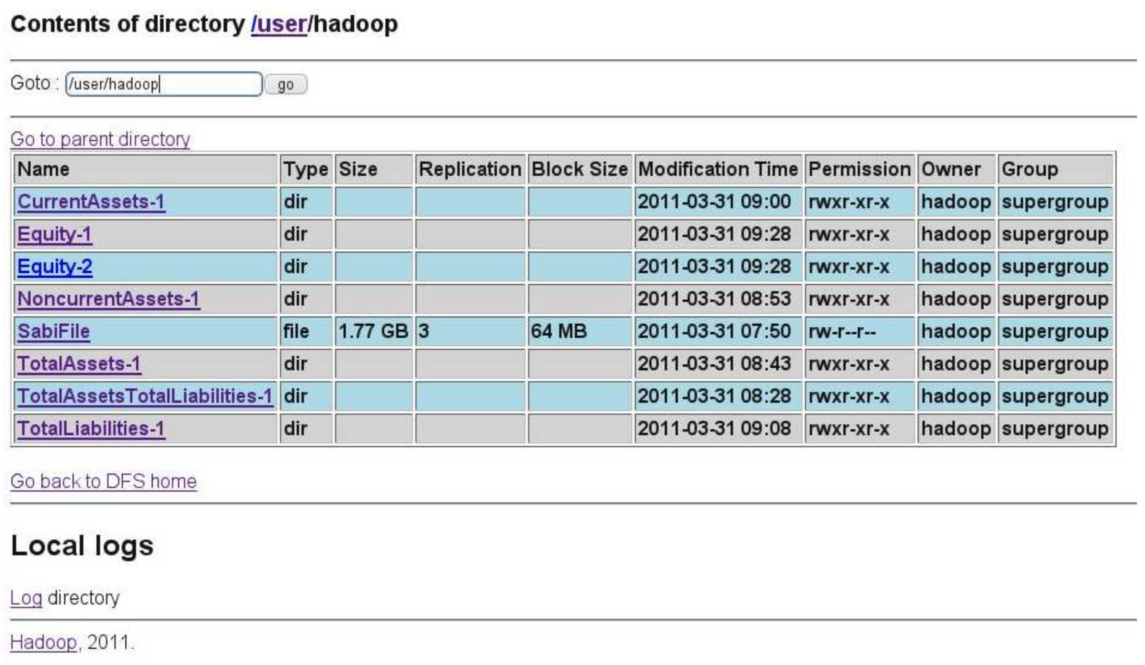
HTTP This is a read-only interface for retrieving directory listings and data. Directory listings are served by the namenode's embedded web server (which runs on port 50070) in XML format, while file data is streamed from datanodes by their web servers (running on port 50075). We can show an example in Figure 2.1:

FTP This interface permits the use of the FTP protocol to interact with HDFS. This is a convenient way to transfer data into an out of HDFS using existing FTP clients.

2.1.4 Data Flow

Anatomy of a File Read

Figure 2.1: HTTP Interface from HDFS



Contents of directory /user/hadoop

Goto : /user/hadoop go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
CurrentAssets-1	dir				2011-03-31 09:00	rwxr-xr-x	hadoop	supergroup
Equity-1	dir				2011-03-31 09:28	rwxr-xr-x	hadoop	supergroup
Equity-2	dir				2011-03-31 09:28	rwxr-xr-x	hadoop	supergroup
NoncurrentAssets-1	dir				2011-03-31 08:53	rwxr-xr-x	hadoop	supergroup
SabiFile	file	1.77 GB	3	64 MB	2011-03-31 07:50	rw-r--r--	hadoop	supergroup
TotalAssets-1	dir				2011-03-31 08:43	rwxr-xr-x	hadoop	supergroup
TotalAssetsTotalLiabilities-1	dir				2011-03-31 08:28	rwxr-xr-x	hadoop	supergroup
TotalLiabilities-1	dir				2011-03-31 09:08	rwxr-xr-x	hadoop	supergroup

Go back to DFS home

Local logs

Log directory

Hadoop, 2011.

To learn how data flows between the client interacting with HDFS we are going to consider Figure 2.2 in which the namenode and the datanode shows the main sequence of events when reading a file.

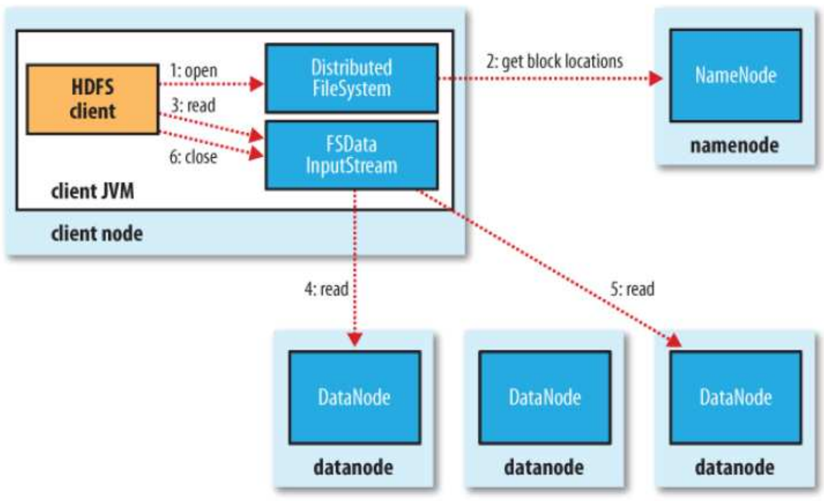
The client opens the file it wishes to read by calling `open()` on the FileSystem object, which for HDFS is an instance of DistributedFileSystem (step 1 in Figure 2.2). DistributedFileSystem calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client (according to the topology of the clusters network). If the client is itself a datanode (in the case of a MapReduce task, for instance), then it will read from the local datanode.

The DistributedFileSystem returns a FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

The client then calls `read()` on the stream (step 3). DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls `read()` repeatedly on the stream (step 4). When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream.

Blocks are read in order with the DFSInputStream opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls

Figure 2.2: A client reading data from HDFS



`close()` on the FSDataInputStream (step 6).

During reading, if the client encounters an error while communicating with a datanode, then it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesnt needlessly retry them for later blocks. The client also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, it is reported to the namenode, before the client attempts to read a replica of the block from another datanode [20].

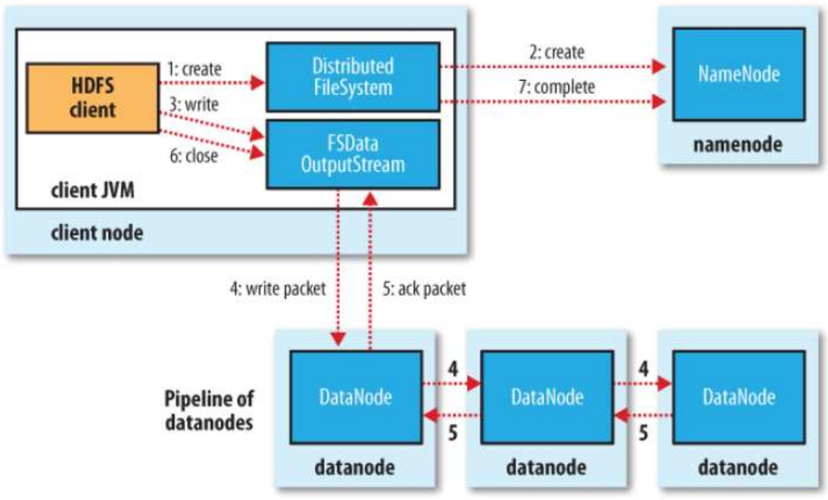
Anatomy of a File Write

The client creates the file by calling `create()` on DistributedFileSystem (step 1 in 2.3). DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystems namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesnt already exist, and that the client has the right permissions to create the file. If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException. The DistributedFileSystem returns a FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutput Stream, which handles communication with the datanodes and namenode.

As the client writes data (step 3), DFSOutputStream splits it into packets, which it writes to an internal queue, called the data queue. The data queue is consumed by the Data Streamer, whose responsibility it is to ask the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipelinewell assume the replication level is 3, so there are three nodes in the pipeline. The DataStreamer streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).

DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the ack queue. A packet is removed from the ack queue

Figure 2.3: A client writing data to HDFS



only when it has been acknowledged by all the datanodes in the pipeline (step 5).

If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on. The failed datanode is removed from the pipeline and the remainder of the blocks data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.

Its possible, but unlikely, that multiple datanodes fail while a block is being written. As long as dfs.replication.min replicas (default one) are written the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (dfs.replication, which defaults to three).

When the client has finished writing data it calls `close()` on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up of (via Data Streamer asking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully [20].

2.2 The MapReduce procedure

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pair, and a reduce function that merges all intermediate values associated with the same intermediate key.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication [9].

2.2.1 Map and Reduce Functions

The user has to implement two functions to express the computation: Map and Reduce functions.

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function. The number of partitions and the partitioning function are specified by the user.

Later in this chapter, Section 2.3, we will show how the Map and Reduce Functions works with an example.

2.2.2 Interface

As we said in the previous section there are a HTTP interface for the MapReduce jobs that is ran by the master and exports a set of status pages show the progress of the computation, such as how may task have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. Moreover, we can show which workers have filed, and which map and reduce tasks they were processing when they filed. Lets see an example of this interface at Figure 2.4.

2.2.3 Fault Tolerance

The MapReduce library tolerate machine failures in case that one of the workers failed. The master pings every worker periodically. If no response is revived from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data form worker A will read the data from worker B.

Due to this MapReduce is resilient to large-scale worker failures [9].

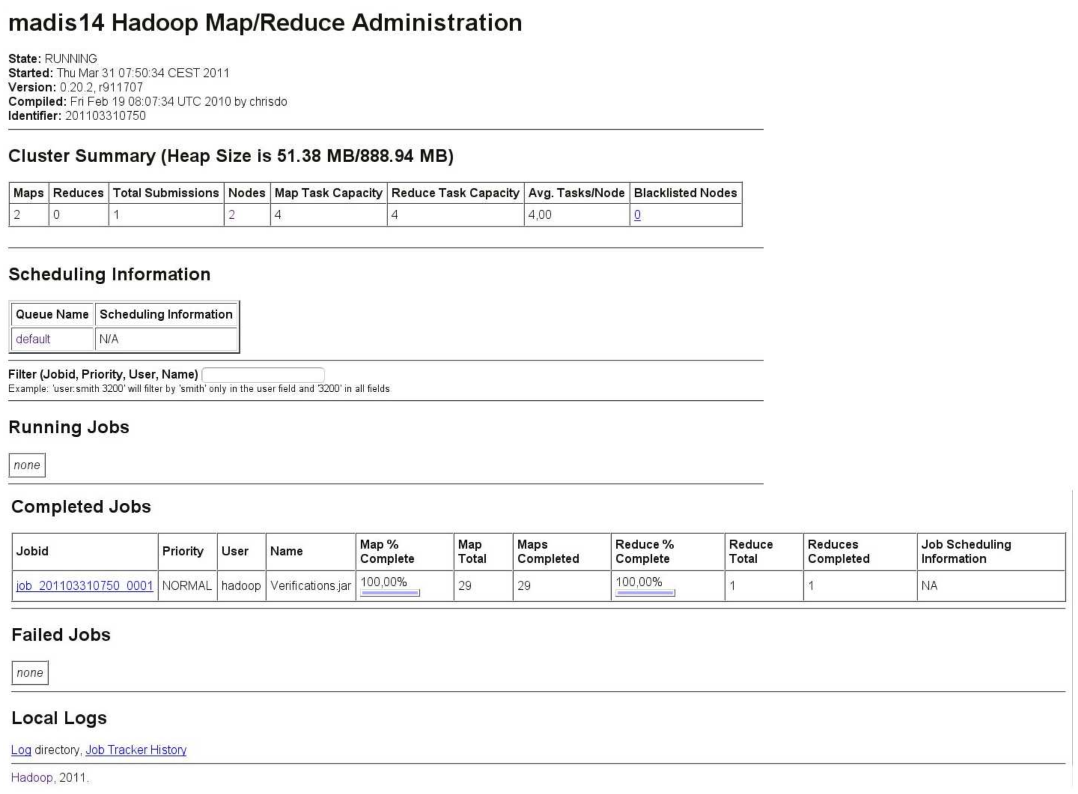


Figure 2.4: HTTP Interface from MapReduce

madis14 Hadoop Map/Reduce Administration

State: RUNNING
Started: Thu Mar 31 07:50:34 CEST 2011
Version: 0.20.2, r911707
Compiled: Fri Feb 19 08:07:34 UTC 2010 by chrisdo
Identifier: 201103310750

Cluster Summary (Heap Size is 51.38 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
2	0	1	2	4	4	4,00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_201103310750_0001	NORMAL	hadoop	Verifications.jar	100,00%	29	29	100,00%	1	1	NA

Failed Jobs

none

Local Logs

Log directory, Job Tracker History

Hadoop, 2011.

2.3 A Weather Dataset

To understand how MapReduce works we are going to explain the example that appeared in [20]. They manage a weather dataset through a MapReduce program. These data are temperature measures of different part of the world and are store in a file in which each row represent a measure.

We shows a sample of the weather repository to comprehend how the map and reduce functions works:

$$0067011990999991950051507004\ldots 9999999N9 + 00001 + 99999999999\ldots$$
$$0043011990999991950051512004\ldots 9999999N9 + 00221 + 99999999999\ldots$$
$$0043011990999991950051518004\ldots 9999999N9 - 00111 + 99999999999\ldots$$
$$0043012650999991949032412004\ldots 0500001N9 + 01111 + 99999999999\ldots$$
$$0043012650999991949032418004\ldots 0500001N9 + 00781 + 99999999999\ldots$$

These lines are presented to the map function as the key-value pairs:

$$(0, 006701199099999\mathbf{1950}051507004\ldots 9999999N9\mathbf{+0000}1 + 99999999999\ldots)$$
$$(106, 004301199099999\mathbf{1950}051512004\ldots 9999999N9\mathbf{+0022}1 + 99999999999\ldots)$$
$$(212, 004301199099999\mathbf{1950}051518004\ldots 9999999N9\mathbf{-0011}1 + 99999999999\ldots)$$
$$(318, 004301265099999\mathbf{1949}032412004\ldots 0500001N9\mathbf{+0111}1 + 99999999999\ldots)$$
$$(424, 004301265099999\mathbf{1949}032418004\ldots 0500001N9\mathbf{+0078}1 + 99999999999\ldots)$$

The keys are the line offsets within the file, which we ignore in our map function. The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output. (The temperature values have been interpreted as integers.)

$(1950, 0)$
$(1950, 22)$
$(1950, -11)$
$(1949, 111)$
$(1949, 78)$

The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key. So, continuing the example, our reduce function sees the following input:

$(1949, [111, 78])$
$(1950, [0, 22, -11])$

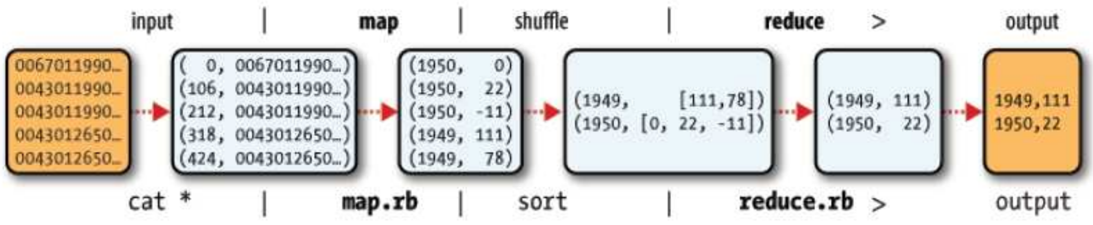
Each year appears with a list of all its air temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

$(1949, 111)$
$(1950, 22)$

This is the final output: the maximum global temperature recorded in each year.

The whole data flow is illustrated in Figure 2.5.

Figure 2.5: MapReduce logical data flow



To sum up, the Map function takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function. The Reduce function, accepts an intermediate key, and set of values for that key. It merges together these values to form a possibly smaller set of values. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory. And this Reduce function produce key/value pairs whit the results that we wish to obtain.

Chapter 3

Experimental Setup

3.1 Case of study: SABI

This section presents a case of study where map/reduce functions are applied to deal with large amounts of data and perform financial computation. Especially, we focus on the repository *Sistema de Análisis de Balances Ibéricos* (SABI) which provides data of 1.222.198 Spanish and Portuguese companies from 1994 to 2009 it is approximately of 20 GB size and is used by 99% of the Spanish Universities for several research studies in the fields of Economics and Bussiness Administration. Both high dimensionality and popularity, couple with some critiques about the consistency of this database. We are interested in proving the data correctness with a distributed software processing tool [17].

3.1.1 Data availability and format

Sabi is available at [19]. However, there is a big constraint since only up to 500 companies can be retrieved per query issued, due to the downloading format provided by the web site. Therefore, due to the volume of data we are dealing with, we opted to obtain the data using the DVD that SABI offers to their customers. Data can be obtained either as a text or an Excel file, we chose the former as the limitation in the number of columns and rows of the latter makes it hard to afford.

3.1.2 Verification: Database Correctness

SABI stores financial information from Spanish and Portuguese firms. All data are manually inserted which leads to (1) human errors and (2) missing values. In order to detect and correct such incompleteness and possible inconsistencies, we propose some verifications. In the following, we describe the main economic indicators used and present some description information about the data set .

We proceed to process the information associated with the balance accounts of the companies that SABI has in its database. These balance accounts correspond to those presented by the company in the Business Registry and should meet specific accounting criteria. For the sake of clarity, in the following we describe the balance account definitions we need to process with the data verification.

Total Assets are possessions, rights and other resources financially controlled by the company; i.e., result of past circumstances from the company which will probably derive economic benefits in the future [7].

Additionally, the conceptual framework of accounting requires that the asset is rated with viability and simultaneously for its recognition; i.e., each entry in the asset must have an entry in the liability as well to have a balance between both accounts. This is called Method of Double Lines or Conceptual Framework Accounting.

Noncurrent Assets are those things a company owns that are not expected to be converted into or used as cash within one year, e.g. equipment, furniture and fixtures, real estate, patents, trademarks, and long-term investments [18].

Current Assets are those things a company owns that are expected to be turned into (or used as) cash within one year from the date they are listed on the balance sheet, e.g. current assets include cash, short-term investments, promissory note (IOUs) from customers, prepaid expenses, and inventory [18].

Total Liabilities are those elements that compose the balance sheet accounts: net assets, noncurrent liabilities and current liabilities. Total liabilities are simply monies that a company owes. A business might owe money to the Internal Revenue Service in the form of taxes, to employees in the form of accrued payroll, to vendors in the form of accounts payable, or to banks for credits cards, mortgages, and other loans. The company takes into account all these and expect extinction outflow of resources that can produce profits or future economic performance [7, 18].

Equity are the share of owners in the financing of the company. This can vary depending on the contribution made by the owners or partners as well as on the distribution company results. This is one of the accounts that is part of the net assets.

The verifications we carried out according to the following account are:

- $Total\,Assets\; = \;Total\,Liabilities$
- $Total\,Assets\; = \;Uncalled\,shareholders + Immobilized + Expenses\,deferred + Current\,Assets$
- $Noncurrent\,Assets\; = \;Up\,cost + Intangible\,assets + Tangible\,assets + Financial\,assets + Long - term\,treasury\,stock\; + \;Due\,on\,long - term\,traffic$
- $Current\,Assets\; = \;Expenditure\,required\,by\,shareholders + Stocks + Debtors + Short\,term\,investments + Short - term\,treasury\,shares + Treasury + Accrual$
- $Total\,Liabilities\; = \;Equity + Revenue\,deferred + Provisions\,for\,liabilities\,and\,charges + Long - term\,creditors\; + \;Short - term\,creditors$
- $Equity\; = \;Subscribed\,capital + premium + reservations\,and\,results\,for\,previous\,exercises\,income\; + \;interim\,dividend\,paid\,during\,the\,year + share\,for\,capital\,reduction$

3.1.3 Computation: Return On Investments (ROI)

We cannot only verify the accuracy of data inserted into the SABI repository but we can also perform some computations. We computed the companies ranking according to their financial profitability, ROI. It is a performance measure used to evaluate the efficiency of an investment or to compare the efficiency of a number of different investments. ROI is measured as the ratio between the Result Before Taxes (RBT) and the Equity. We have used Hadoop MapReduce to perform this computations and hence we show that it is also an attractive to perform data computation over the SABI repository. We show below how ROI is calculated and in Section 4.3 how we proceed with this computations with Hadoop MapReduce.

$$ROI = \frac{Profit\ on\ ordinary\ activities + Extraordinary\ positive\ results}{Equity + Revenue\ deferred - Uncalled\ shareholders - Longterm\ treasury\ shares}$$

3.1.4 Validity of returned results

We compare each balance account of the companies and the ROI. To this end, we created columns with dichotomic results (0 and 1) controlled by the name of the company and the year, where 0 means that there is a wrong entry in the data of the company that the company inserted and 1 otherwise.

We have run our MapReduce computation over a reduced data sample in order to verify its correctness against its equivalent computation with Microsoft Excel.

In order to give a description of the data inserted in the repository we ran a MapReduce computation that produces the results shown in Table 3.1. Moreover, we have to say that all the years have the same number of companies (266417), although some companies have disappeared, SABI decided to put the label (*EXTINGUIDA*) and without putting any information about the companies with this label.

We can see from Table 3.1 that the different activity sectors obtained according to the CNAE classification [12] this information is located in the 11th field, and we have used the first two digits of this field to classified the companies as follows: from 01 to 04 Agriculture; and, 05-34 Industry; and, 35-39 Energies; and 41-44 Construction and Dwelling; and, 45-49 Services. As some companies did not insert this field but they introduce the rest of information we decided to create another column called “Unclassified”. It is important to note that if we would want to make a research based on this distribution, we would have to eliminate all these companies.

Finally, we want to highlight, from the information derived from Table 3.1, only the 1,26% of our repository have introduced the data and that most of them belong the services sector. Other significant aspect is that SABI has more data between 2002 and 2006 that indicates that we have more information in the central period analyzed.

3.2 Treatment file

The data repository that we obtained that from the SABI DVD had a structure that could not be directly applied to Hadoop. Due to that we had to develop some programs to modify this structure. In Figure 3.1 we show the first five fields of a SABI file. The different fields are separated by blank spaces, such that the pattern is that between character of one field and the first character of the next field are at must 31 characters.Another characteristic of the file is that the companies with large names has its name separated in several lines.

Table 3.1: Distribution of the repository with regard to the activity sector and the year

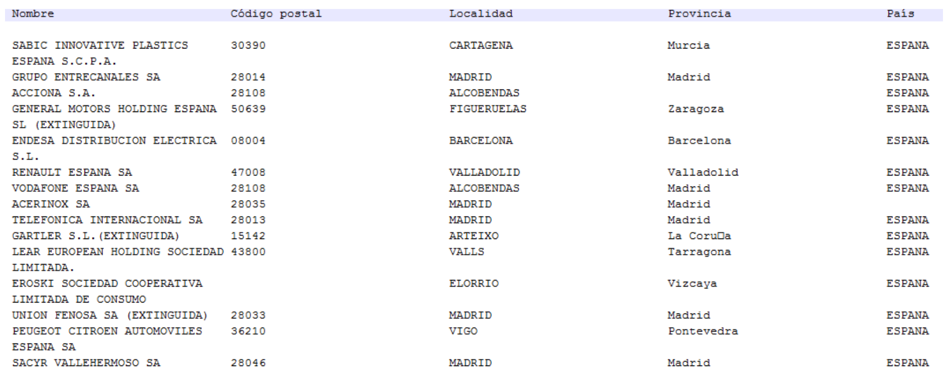
	Agriculture	Industry	Energies	Construction & Dwellings	Services	Unclassified	Total Sample
1994	61	1539	51	528	2652	0	4831
1995	113	2244	77	942	4187	0	7563
1996	265	6115	203	3137	11970	0	21690
1997	298	6444	228	3412	13057	1	23440
1998	303	6644	267	3840	14119	1	25174
1999	421	8895	387	5282	18508	1	33494
2000	439	9110	426	5693	19327	3	34998
2001	465	9250	467	6188	20294	5	36669
2002	468	9281	508	6749	21044	6	38056
2003	492	9274	525	7101	21539	5	38936
2004	501	9271	516	7386	21812	2	39488
2005	496	9266	555	7632	22170	2	40121
2006	501	9204	588	7636	22311	2	40242
2007	462	8616	553	6861	20776	6	37274
2008	151	2648	202	1697	5598	1	10297
2009	0	0	0	0	0	0	0
TOTAL	5436	107801	5553	74084	239364	35	432273

Figure 3.2 presents another five fields of the file that refers to specific accounting information of the companies introduced in Figure 3.1. We can see form Figure 3.2 that some of the fields are empty, due to that we need to fill this blank fields with a character that does not change the meaning, so we decided to put a “*” character.

To solve these problems we prepared several scripts that fit the file according to the Hadoop supported structure. The first point was to correct the companies names and put them in one single line per company. To do so, we create a file with the name of the companies followed by 1 if the line has the first part of the company name or 0 otherwise. Thanks to this file structure we can easily build the company name into a single row name (see Figure 3.3 for the source code sample and Figure 3.4 for the resulting file):

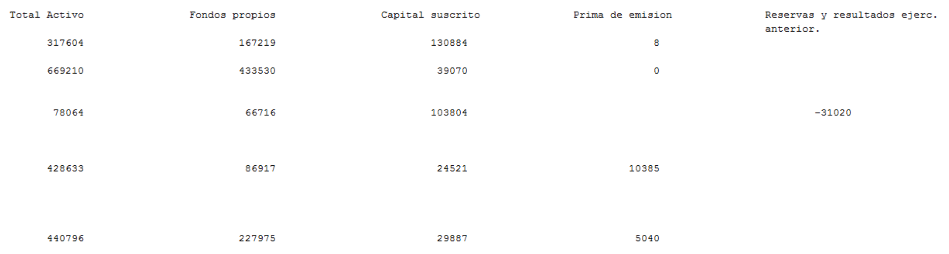
The second part of our file treatment consisted in replacing the empty fields with a special character “*”. We produced a program with this feature, the program reads in blocks of 31 characters and if the next one is a blank space it is replaced by “*”; otherwise it continues the reading process. (see Figure 3.5 for the source code sample and Figures 3.6, 3.7 for the result file). Later in the process we use the character “$\backslash t$” to define the different fields. The next stage consisted in replaceing the wrong layout of the companies’ names to fix them to campanies’ names the correct one that we obtained previously and adding the year at the beginning of the line ready to be used by Hadoop and adapted for our specific needs of computation. We show a sample of this file in Figure 3.8.

Figure 3.1: The first 5 fields of a sample SABI file



Nombre	Código postal	Localidad	Provincia	País
SABIC INNOVATIVE PLASTICS ESPANA S.C.P.A.	30390	CARTAGENA	Murcia	ESPANA
GRUPO ENTRECANALES SA	28014	MADRID	Madrid	ESPANA
ACCIONA S.A.	28108	ALCOBENDAS		ESPANA
GENERAL MOTORS HOLDING ESPANA SL (EXTINGUIDA)	50639	FIGUERUELAS	Zaragoza	ESPANA
ENDESA DISTRIBUCION ELECTRICA S.L.	08004	BARCELONA	Barcelona	ESPANA
RENAULT ESPANA SA	47008	VALLADOLID	Valladolid	ESPANA
VODAFONE ESPANA SA	28108	ALCOBENDAS	Madrid	ESPANA
ACERINOX SA	28035	MADRID	Madrid	
TELEFONICA INTERNACIONAL SA	28013	MADRID	Madrid	ESPANA
GARTLER S.L.(EXTINGUIDA)	15142	ARTEIXO	La Coru□a	ESPANA
LEAR EUROPEAN HOLDING SOCIEDAD LIMITADA.	43800	VALLS	Tarragona	ESPANA
EROSKI SOCIEDAD COOPERATIVA LIMITADA DE CONSUMO		ELORRIO	Vizcaya	ESPANA
UNION FENOSA SA (EXTINGUIDA)	28033	MADRID	Madrid	ESPANA
PEUGEOT CITROEN AUTOMOVILES ESPANA SA	36210	VIGO	Pontevedra	ESPANA
SACYR VALLEHERMOSO SA	28046	MADRID	Madrid	ESPANA

Figure 3.2: A sample of a SABI file with some accounting info



Total Activo	Fondos propios	Capital suscrito	Prima de emision	Reservas y resultados ejerc. anterior.
317604	167219	130884	8	
669210	433530	39070	0	
78064	66716	103804		-31020
428633	86917	24521	10385	
440796	227975	29887	5040	

3.3 Sabi Data Repository Explanation

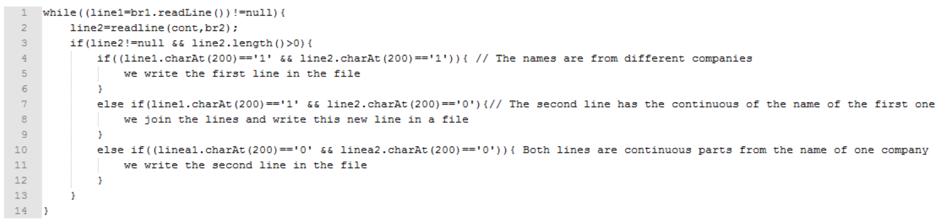
In this section we are going to explain how our system works. To understand it properly we are going to use Figure 3.9. It shows us an example of comparison between the Total Assets and the sum of Uncalled shareholders, Immobilized, Expenses deferred, Assets:

The input file of our system corresponds to the following format; i.e. “`Year \t Company \t...\t uncalled shareholders \t immobilized \t...\t Expenses deferred \t Assets \t...\t total assets \t...`”. For example, we have: “`2006 \t Repsol Petróleos S.A. \t...\t 2242904 \t...\t 48258 \t 3452272 \t...\t 5743434 \t...`”.

The master (JobTracker in Figure 3.9) assigns to several workers (TaskTrakers in Figure 3.9) the different blocks in which the files are split into to do their map tasks. These tasks follow the same structure for all the comparisons that we have specified per year; all of these accounts are separated using “`\t`” too. So the result of the map task will be in the case: “`2006 \t Repsol Petróleos S.A. \t 5.743.434 \t 0 \t 2.242.904 \t 48.258 \t 3.452.272`”.

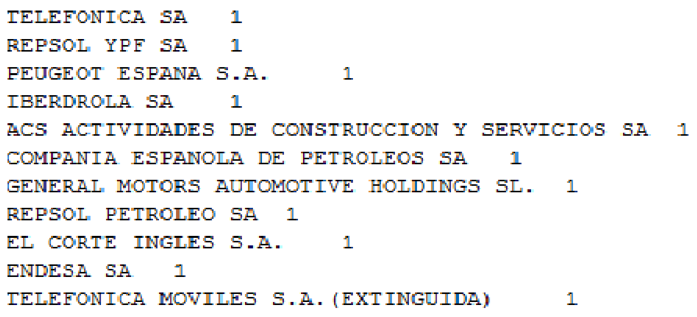
These map tasks will eventually finish and populate the intermediate files. Previously, the master will assign to other workers the reduce task to be executed over all the intermediate files. It will first obtain the different accounts. The second value corresponds to the total assets and the rest corresponds to the values whose addition must be equal to the former. In our case, we have that: total assets = 5.743.434, uncalled shareholders = 0, immobilized =

Figure 3.3: code companies name



```
while((line1=br1.readLine())!=null){
    line2=readline(cont,br2);
    if(line2!=null && line2.length()>0){
        if((line1.charAt(200)=='1' && line2.charAt(200)=='1')){ // The names are from different companies
            we write the first line in the file
        }
        else if(line1.charAt(200)=='1' && line2.charAt(200)=='0'){// The second line has the continuous of the name of the first one
            we join the lines and write this new line in a file
        }
        else if((linea1.charAt(200)=='0' && linea2.charAt(200)=='0')){ Both lines are continuous parts from the name of one company
            we write the second line in the file
        }
    }
}
```

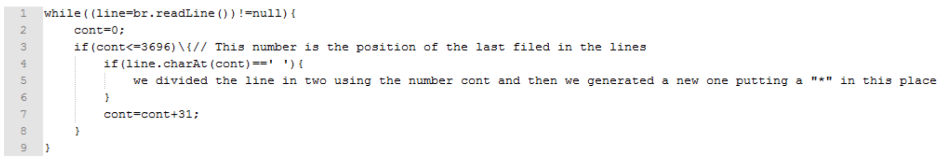
Figure 3.4: SABI file company's name



```
TELEFONICA SA    1
REPSOL YPF SA    1
PEUGEOT ESPANA S.A.       1
IBERDROLA SA     1
ACS ACTIVIDADES DE CONSTRUCCION Y SERVICIOS SA   1
COMPANIA ESPANOLA DE PETROLEOS SA    1
GENERAL MOTORS AUTOMOTIVE HOLDINGS SL.   1
REPSOL PETROLEO SA   1
EL CORTE INGLES S.A.     1
ENDESA SA    1
TELEFONICA MOVILES S.A.(EXTINGUIDA)      1
```

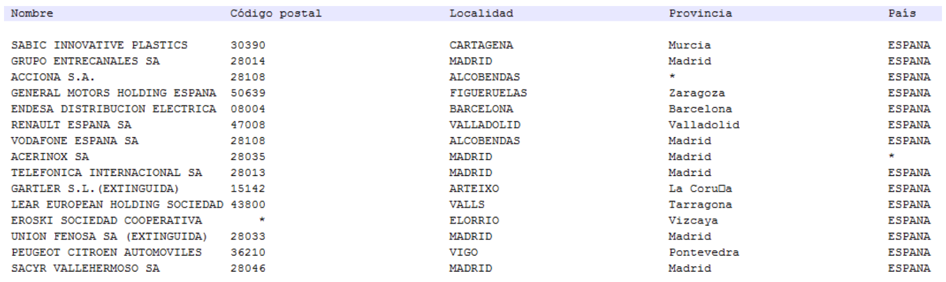
2.242.904, expenses deferred = 48.258, current assets = 3.452.272. In this case it is satisfied and it will return the following tuple: "`Repsol Petróleos S.A.` $\backslash t$ `2006` $\backslash t$ `1`"; otherwise, it will be "`Repsol Petrleos S.A.` $\backslash t$ `2006` $\backslash t$ `0`". At the end of the reduce task we will have a file composed of the following tuples (formatted as text lines with "$\backslash t$" as the field separator for each tuple): "`Cemex España S.A.` $\backslash t$ `2003` $\backslash t$ `0`" . . . "`Endesa Distribución Eléctrica` $\backslash t$ `2004` $\backslash t$ `1`" . . . "`Ford España S.L.` $\backslash t$ `2000` $\backslash t$ `0`" . . . "`Repsol Petróleos S.A.` $\backslash t$ `2006` $\backslash t$ `1`" . . . .

Figure 3.5: code prepearing fields



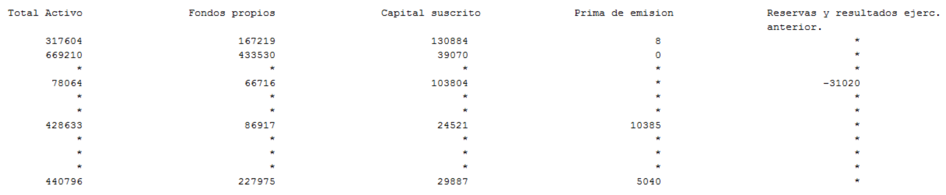
```
while((line=br.readLine())!=null){
    cont=0;
    if(cont<=3696)\{// This number is the position of the last filed in the lines
        if(line.charAt(cont)==' '){
            we divided the line in two using the number cont and then we generated a new one putting a "*" in this place
        }
        cont=cont+31;
    }
}
```

Figure 3.6: SABI file replaced fields in head part



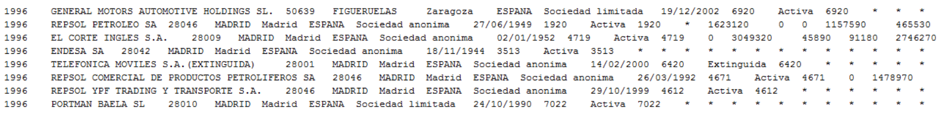
Nombre	Código postal	Localidad	Provincia	País
SABIC INNOVATIVE PLASTICS	30390	CARTAGENA	Murcia	ESPANA
GRUPO ENTRECANALES SA	28014	MADRID	Madrid	ESPANA
ACCIONA S.A.	28108	ALCOBENDAS	*	ESPANA
GENERAL MOTORS HOLDING ESPANA	50639	FIGUERUELAS	Zaragoza	ESPANA
ENDESA DISTRIBUCION ELECTRICA	08004	BARCELONA	Barcelona	ESPANA
RENAULT ESPANA SA	47008	VALLADOLID	Valladolid	ESPANA
VODAFONE ESPANA SA	28108	ALCOBENDAS	Madrid	ESPANA
ACERINOX SA	28035	MADRID	Madrid	*
TELEFONICA INTERNACIONAL SA	28013	MADRID	Madrid	ESPANA
GARTLER S.L.(EXTINGUIDA)	15142	ARTEIXO	La Coru□a	ESPANA
LEAR EUROPEAN HOLDING SOCIEDAD	43800	VALLS	Tarragona	ESPANA
EROSKI SOCIEDAD COOPERATIVA	*	ELORRIO	Vizcaya	ESPANA
UNION FENOSA SA (EXTINGUIDA)	28033	MADRID	Madrid	ESPANA
PEUGEOT CITROEN AUTOMOVILES	36210	VIGO	Pontevedra	ESPANA
SACYR VALLEHERMOSO SA	28046	MADRID	Madrid	ESPANA

Figure 3.7: SABI file replaced fields in tail part



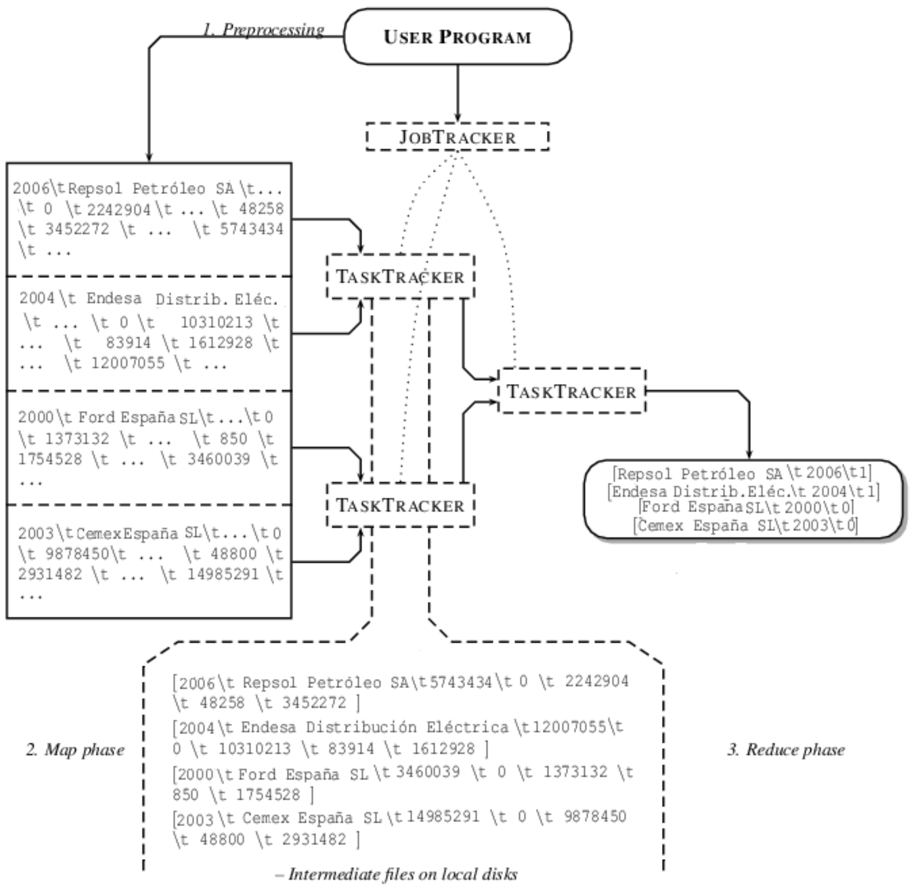
Total Activo	Fondos propios	Capital suscrito	Prima de emision	Reservas y resultados ejerc. anterior.
317604	167219	130884	8	*
669210	433530	39070	0	*
*	*	*	*	*
78064	66716	103804	*	-31020
*	*	*	*	*
*	*	*	*	*
428633	86917	24521	10385	*
*	*	*	*	*
*	*	*	*	*
*	*	*	*	*
440796	227975	29887	5040	*

Figure 3.8: SABI file corrected



```
1996   GENERAL MOTORS AUTOMOTIVE HOLDINGS SL.  50639   FIGUERUELAS    Zaragoza    ESPANA  Sociedad limitada   19/12/2002  6920    Activa  6920    *   *   *
1996   REPSOL PETROLEO SA  28046   MADRID  Madrid  ESPANA  Sociedad anonima    27/06/1949  1920    Activa  1920    *   1623120     0   0   1157590     465530
1996   EL CORTE INGLES S.A.    28009   MADRID  Madrid  ESPANA  Sociedad anonima    02/01/1952  4719    Activa  4719    0   3049320     45890   91180   2746270
1996   ENDESA SA   28042   MADRID  Madrid  ESPANA  Sociedad anonima    18/11/1944  3513    Activa  3513    *   *   *   *   *   *   *   *   *   *   *   *   *
1996   TELEFONICA MOVILES S.A.(EXTINGUIDA)     28001   MADRID  Madrid  ESPANA  Sociedad anonima    14/02/2000  6420    Extinguida  6420    *   *   *   *   *
1996   REPSOL COMERCIAL DE PRODUCTOS PETROLIFEROS SA   28046   MADRID  Madrid  ESPANA  Sociedad anonima    26/03/1992  4671    Activa  4671    0   1478970
1996   REPSOL YPF TRADING Y TRANSPORTE S.A.    28046   MADRID  Madrid  ESPANA  Sociedad anonima    29/10/1999  4612    Activa  4612    *   *   *   *   *   *
1996   PORTMAN BAELA SL    28010   MADRID  Madrid  ESPANA  Sociedad limitada   24/10/1990  7022    Activa  7022    *   *   *   *   *   *   *   *   *   *   *
```

Figure 3.9: MapReduce Architecture



1. Preprocessing
USER PROGRAM
JOBTRACKER
2006\t Repsol Petróleo SA \t... \t 0 \t 2242904 \t ... \t 48258 \t 3452272 \t ... \t 5743434 \t ...
2004 \t Endesa Distrib. Eléc. \t ... \t 0 \t 10310213 \t ... \t 83914 \t 1612928 \t ... \t 12007055 \t ...
2000 \t Ford España SL \t ... \t 0 \t 1373132 \t ... \t 850 \t 1754528 \t ... \t 3460039 \t ...
2003 \t Cemex España SL \t ... \t 0 \t 9878450 \t ... \t 48800 \t 2931482 \t ... \t 14985291 \t ...
TASKTRACKER
TASKTRACKER
TASKTRACKER
[Repsol Petróleo SA \t 2006 \t 1]
[Endesa Distrib.Eléc. \t 2004 \t 1]
[Ford España SL \t 2000 \t 0]
[Cemex España SL \t 2003 \t 0]
2. Map phase
[2006 \t Repsol Petróleo SA \t 5743434 \t 0 \t 2242904 \t 48258 \t 3452272]
[2004 \t Endesa Distribución Eléctrica \t 12007055 \t 0 \t 10310213 \t 83914 \t 1612928]
[2000 \t Ford España SL \t 3460039 \t 0 \t 1373132 \t 850 \t 1754528]
[2003 \t Cemex España SL \t 14985291 \t 0 \t 9878450 \t 48800 \t 2931482]
3. Reduce phase
– Intermediate files on local disks

Chapter 4

Discussion

This Chapter presents the execution of our project, which we have described previously. After introducing some aspects of our implementation in Section 4.1 and describing the experimental setting in Section 4.2, Section 4.3 brings some insights about the process, results and behavior obtained from our system with different configurations. Finally, in Section 4.4 analyzes analyze the results obtained using some tables.

4.1 System Implementation

The implementation of the system is has been done using Hadoop Version 0.20.0 [4]. All system parameters can be configured using XML files.

core-site.xml In this configuration file we specify the path of the directory in which there are going to be the temporary files that appeared during the MapReduce execution, moreover we specify the name of the default file system (see Figure 4.1):

hadoop-env.sh In this configuration file we specify the path of the Java implementation to use (see Figure 4.2):

hdfs-site.xml In this confiruation file we specify the number of replications can be specified when the file is created and the block size in bits (see Figure 4.3):

mapred-site.xml In this configuration file we specify the host and port that the MapReduce job tracker runs at (see Figure 4.4):

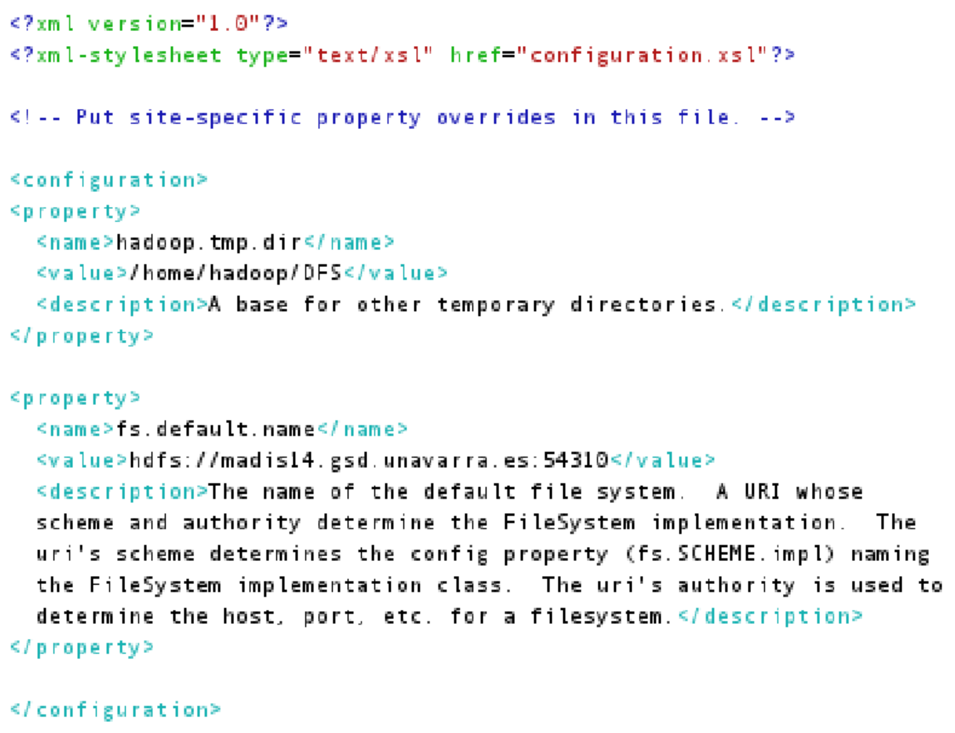
masters In this file we specify the name of the master, in our case (see Figure 4.5):

slaves In this file we specify the names of the slaves, in this case they are (see Figure 4.6):

4.2 Experimental Settings

Our testing configuration consists of 5 computers conected in a 100Mbps switched LAN, where each machine has an Intel Core 2 Quad processor running at 2.66GHz, 3.39GB of RAM and a 120GB hard disk runing Linux (version openSUSE 11.2 (i586)).

Figure 4.1: `core-site.xml`



```xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/home/hadoop/DFS</value>
  <description>A base for other temporary directories.</description>
</property>

<property>
  <name>fs.default.name</name>
  <value>hdfs://madis14.gsd.unavarra.es:54310</value>
  <description>The name of the default file system.  A URI whose
  scheme and authority determine the FileSystem implementation.  The
  uri's scheme determines the config property (fs.SCHEME.impl) naming
  the FileSystem implementation class.  The uri's authority is used to
  determine the host, port, etc. for a filesystem.</description>
</property>

</configuration>
```

4.3 Sabi Experiments

First of all, we have to upload our SABI file in HDFS.However, we have to configure our system with the different XML files that we have explained before. Afterwards, we have to configure our system and then start our system using a start script. When the system is running we upload the file using this command:
$\%hadoop\,fs\,-put\,originDirectory\,fileName.$

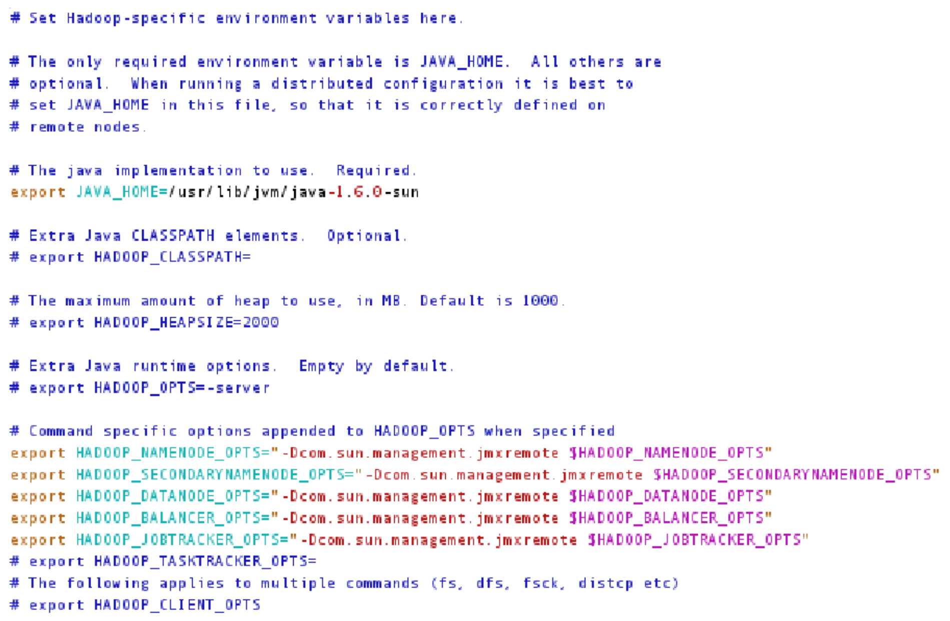
4.3.1 Verifications

We have developed several scripts to adapt the system to different scenarios. We have included in Figures 4.7- 4.9 samples of Map and Reduce functions that we have used for the different verifications.

Let us start with Figure 4.7 that reads each line from the modified SABI file (explained in Section 3.2) and picks up the selected fields separated by "$\backslash t$". For exdample in the case of Total Assets = Total Liabilities we need the 1^{st}, 2^{nd}, 29^{th} and 46^{th} field that correspond with the Year, Name, Total Assets and Total Liabilities respectively. And the output structure in this case is as follows: "`context.write(new Text(name), new Text(year + ''`$\backslash t$`'' + totalAssets + ''`$\backslash t$`'' + totalLiabilities));`" and this is input for the reduce task.

In the reduce task (see Figure 4.8) we receive the output from the map task with this structure: $\langle$ companyName, mapOutput1, mapOutput2, ...$\rangle$. For each line it takes the key companyName and to realize the computation, if it is correct we put 1, and 0 otherwise.

Figure 4.2: `hadoop-env.sh`



```
# Set Hadoop-specific environment variables here.

# The only required environment variable is JAVA_HOME.  All others are
# optional.  When running a distributed configuration it is best to
# set JAVA_HOME in this file, so that it is correctly defined on
# remote nodes.

# The java implementation to use.  Required.
export JAVA_HOME=/usr/lib/jvm/java-1.6.0-sun

# Extra Java CLASSPATH elements.  Optional.
# export HADOOP_CLASSPATH=

# The maximum amount of heap to use, in MB. Default is 1000.
# export HADOOP_HEAPSIZE=2000

# Extra Java runtime options.  Empty by default.
# export HADOOP_OPTS=-server

# Command specific options appended to HADOOP_OPTS when specified
export HADOOP_NAMENODE_OPTS="-Dcom.sun.management.jmxremote $HADOOP_NAMENODE_OPTS"
export HADOOP_SECONDARYNAMENODE_OPTS="-Dcom.sun.management.jmxremote $HADOOP_SECONDARYNAMENODE_OPTS"
export HADOOP_DATANODE_OPTS="-Dcom.sun.management.jmxremote $HADOOP_DATANODE_OPTS"
export HADOOP_BALANCER_OPTS="-Dcom.sun.management.jmxremote $HADOOP_BALANCER_OPTS"
export HADOOP_JOBTRACKER_OPTS="-Dcom.sun.management.jmxremote $HADOOP_JOBTRACKER_OPTS"
# export HADOOP_TASKTRACKER_OPTS=
# The following applies to multiple commands (fs, dfs, fsck, distcp etc)
# export HADOOP_CLIENT_OPTS
```

The output of the reduce task is: $\langle$ `companyName` ``$\backslash t$'' `year1` ``$\backslash t$'' `0 or 1` ``$\backslash t$'' `year2` ``$\backslash t$'' `0 or 1` ``$\backslash t$'' `...`$\rangle$.

The main program specifies the call to MapReduce to execute the scripts. A Job object defines the specification of the job. It gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute in the cluster). Rather than explicitly specify the name of the JAR file, we can pass a class in the `JobConf` constructor, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.

Having constructed a `JobConf` object, we specify the input and output paths. An input path is specified by calling the static `addInputPath()` method on `FileInputFormat`, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern. As the name suggests, `addInputPath()` can be called more than once to use input from multiple paths.

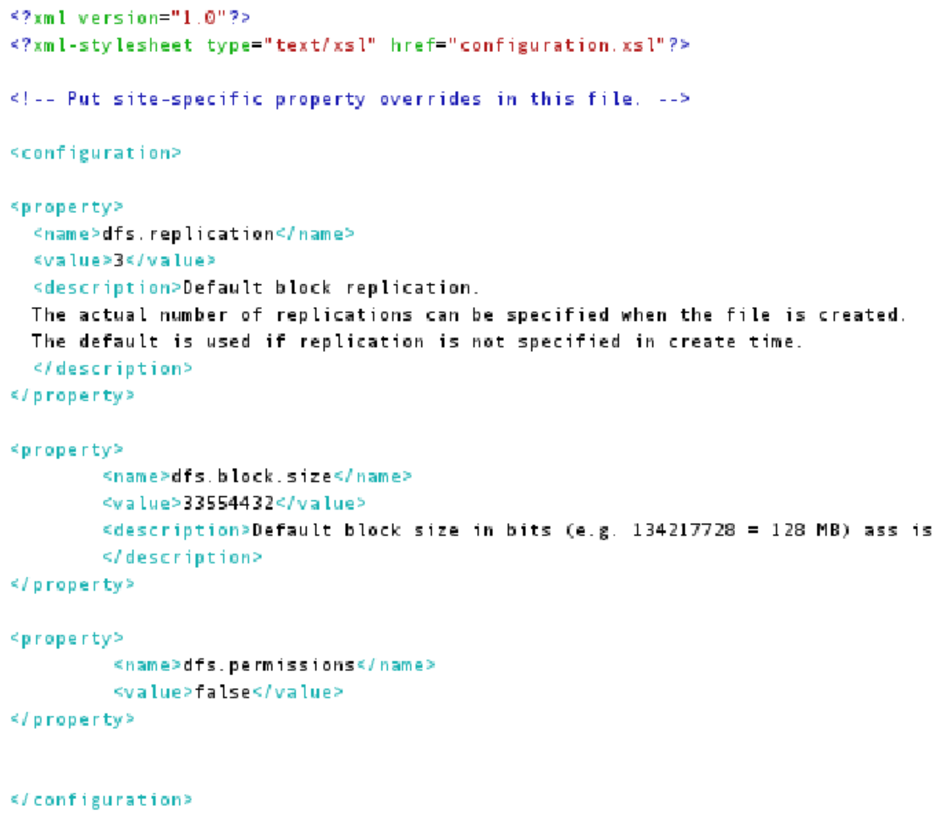
The output path (of which there is only one) is specified by the static `setOutputPath()` method on `FileOutputFormat`. It specifies a directory where the output files from the reducer functions are written. The directory shouldnt exist before running the job, as Hadoop will complain and not run the job. This precaution is to prevent data loss (it can be very annoying to accidentally overwrite the output of a long job with another).

Next, we specify the map and reduce types to use via the `setMapperClass()` and `setReducerClass()` methods.

The `setOutputKeyClass()` and `setOutputValueClass()` methods control the output types for the map and the reduce functions, which are often the same, as they are in our case. If they are different, then the map output types can be set using the methods `setMapOutputKeyClass()` and `setMapOutputValueClass()`.

The input types are controlled via the input format, which we have not explicitly set since

Figure 4.3: `hdfs-site.xml`



```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
  <name>dfs.replication</name>
  <value>3</value>
  <description>Default block replication.
  The actual number of replications can be specified when the file is created.
  The default is used if replication is not specified in create time.
  </description>
</property>

<property>
        <name>dfs.block.size</name>
        <value>33554432</value>
        <description>Default block size in bits (e.g. 134217728 = 128 MB) ass is
        </description>
</property>

<property>
         <name>dfs.permissions</name>
         <value>false</value>
</property>


</configuration>
```

we are using the default `TextInputFormat`.

After setting the classes that define the map and reduce functions, we are ready to run the job. The static `runJob()` method on `JobClient` submits the job and waits for it to finish, writing information about its progress to the console [20].

4.3.2 Return On Investments (ROI)

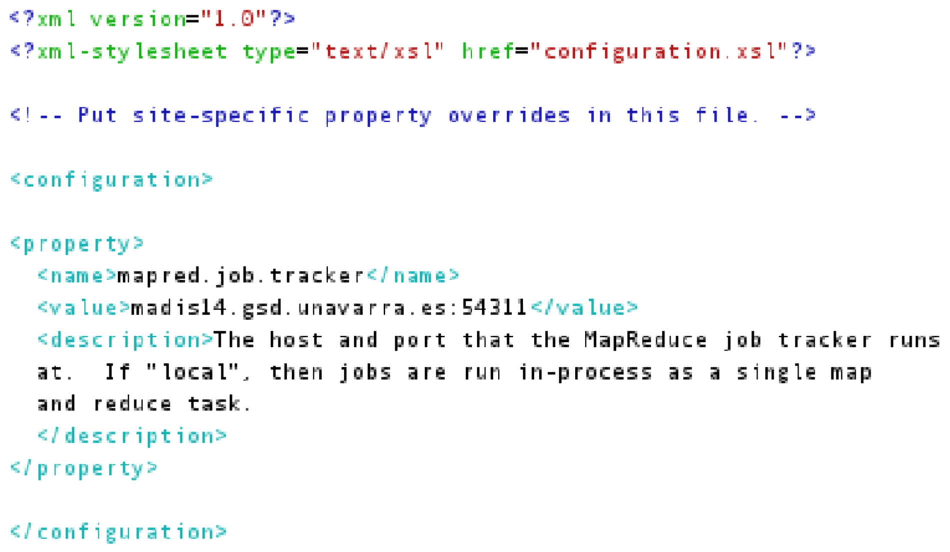
Once we have seen how MapReduce works with the different verifications we have done, we are going to explain how MapReduce can perform simple mathematical operations. In our case we are going to compute the ROI value for all the companies. The ROI parameter measures the efficiency of an investment or to compare the efficiency of a number of different investments.

The Map Task has the same structure in all verifications (see Section 4.3.1). We have to obtain all the data that we need for our operation:

$$ROI = \frac{Profit\ on\ ordinary\ activities + Extraordinary\ positive\ results}{Equity + Revenue\ deferred - Uncalled\ shareholders - Longterm\ treasury\ shares}$$

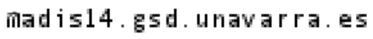
All these fields correspond with the positions 1st, 2nd, 75th, 81th, 30th, 37th, 12th and 18th fields. The output structure in this case is as follows: “`context.write(new Text(name), new Text(year + ‘‘\t’’ + profitOrdActivities + ‘‘\t’’ + ex-`

Figure 4.4: `mapred-site.xml`



```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
  <name>mapred.job.tracker</name>
  <value>madis14.gsd.unavarra.es:54311</value>
  <description>The host and port that the MapReduce job tracker runs
  at.  If "local", then jobs are run in-process as a single map
  and reduce task.
  </description>
</property>

</configuration>
```

Figure 4.5: `masters`



```
madis14.gsd.unavarra.es
```

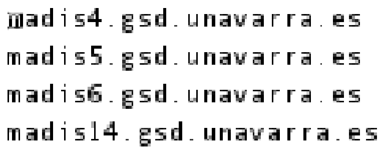
`traPositResults + ‘‘`$\backslash t$`’’ + equity + ‘‘`$\backslash t$`’’ + revenueDeferred + ‘‘`$\backslash t$`’’ + uncalledShareholders + ‘‘`$\backslash t$`’’ + ltTreasuryStocK ));`”.

The different between the previous MapReduce approach is the Reduce Task has to perform an arithmetic computation instead of performing a verification. As we could see in Figure 4.10, first of all we take all the operands with the exception of the company name and the year. We check if all the operands are different from ”* to perform the computation; in such a case the returned result will be ”`ERROR` if one of the operators meets this character, otherwise the operation is performed producing the following output: $\langle$`companyName ‘‘`$\backslash t$`’’ year1 ‘‘`$\backslash t$`’’ result1 ‘‘`$\backslash t$`’’ year2 ‘‘`$\backslash t$`’’ ERROR ‘‘`$\backslash t$`’’ ...`$\rangle$.

4.3.3 Performance analysis

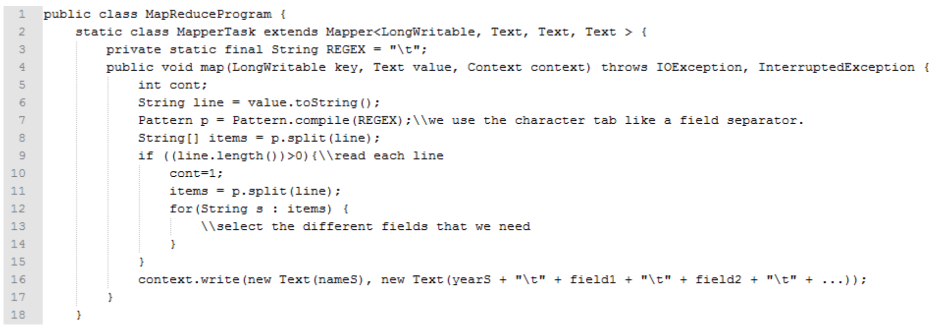
We measure these computations in different scenarios in which we change the number of datanodes (between 3 and 5), the replication depth (between 3 and 4) and the chunk size (32MB, 64MB, 128MB). As we said before the default chunk size is 64MB so we start with this configuration in our experiments; afterward, we changed the parameters mentioned above. With 3 computers as datanodes and a replication number of 3 the execution times were about 120 seconds, so we decided to change our configuration using in this case 4 computers as datanodes and the same replication number and the results were better, we reduce the time execution 20 seconds more or less and the execution times were about 100 seconds. The next step was to check if we change the replication factor to 4, to validate if we can obtain better results. However, the results were similar and we can provide better fault-

Figure 4.6: `slaves`



```
madis4.gsd.unavarra.es
madis5.gsd.unavarra.es
madis6.gsd.unavarra.es
madis14.gsd.unavarra.es
```

Figure 4.7: MapTask



```java
public class MapReduceProgram {
    static class MapperTask extends Mapper<LongWritable, Text, Text, Text > {
        private static final String REGEX = "\t";
        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
            int cont;
            String line = value.toString();
            Pattern p = Pattern.compile(REGEX);\\we use the character tab like a field separator.
            String[] items = p.split(line);
            if ((line.length())>0){\\read each line
                cont=1;
                items = p.split(line);
                for(String s : items) {
                    \\select the different fields that we need
                }
            }
            context.write(new Text(nameS), new Text(yearS + "\t" + field1 + "\t" + field2 + "\t" + ...));
        }
    }
```

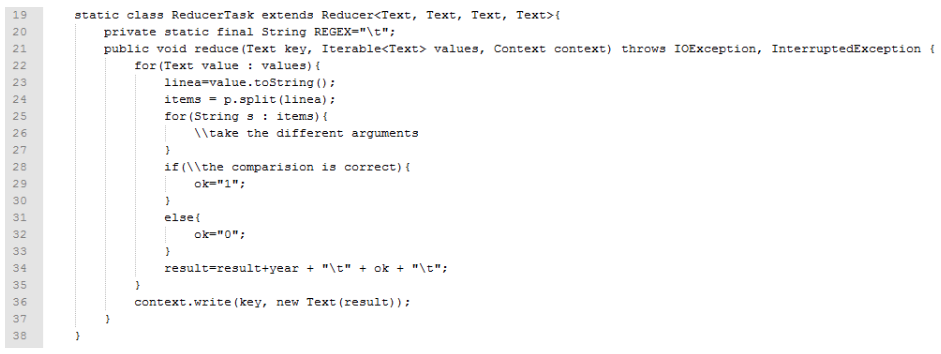
tolerance support without incurring in a greater overhead. However, we continued changing the parameters to 5 datanodes and the different replication factors but the results were worse than the previous cases so we stopped there.

From this situation we decided to change the chunk size from 64MB to 128MB. Thus, as we know that the best configuration so far is 4 computers with a replication factor of 3, we used this configuration with the new chunk size and the result was that the execution time was lower than the same configuration with 64MB of chunk size; we are talking about 95 seconds. Next, we set the configuration of 4 computers and a replication number of 4 again and the result was an increase of 20 seconds. Due to that we decided to change the the chunk size again to 32MB.

The chunk size at this moment was of 32MB and using again the configuration of 4 computers and a replication number of 3, we discover that the execution time increase in a pronounced way; in this case, the completion time was 150 seconds. From these results we notice that the chunk size is so important, if the file is so large we need an appropriate chunk size in order to have the less number of chunks as possible to have less number of Map and Reduce Tasks to be executed, but we have to take notice that if the chunk is so big the Map and Reduce Tasks will take longer. In addition, we reduce the time that the system spend changing between the different chunks during the execution of the map and reduce tasks. The number of computers is so important too, because we have the same number of namenodes as number of computers and this means that we could have at most this number of tasks running concurrently in our system.

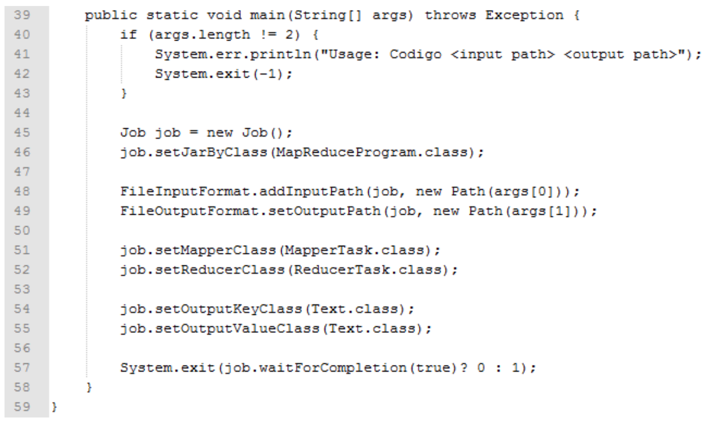
Let us see Table 4.1 to realize a better idea about the different execution times obtained during the experiments and in with we could see that the best configuration is with a chunk size of 128MB using 4 computers with a replication depth of 3 and the worst execution time

Figure 4.8: ReduceTask



```
static class ReducerTask extends Reducer<Text, Text, Text, Text>{
    private static final String REGEX="\t";
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        for(Text value : values){
            linea=value.toString();
            items = p.split(linea);
            for(String s : items){
                \\take the different arguments
            }
            if(\\the comparision is correct){
                ok="1";
            }
            else{
                ok="0";
            }
            result=result+year + "\t" + ok + "\t";
        }
        context.write(key, new Text(result));
    }
}
```

Figure 4.9: MainProgram



```
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: Codigo <input path> <output path>");
            System.exit(-1);
        }

        Job job = new Job();
        job.setJarByClass(MapReduceProgram.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(MapperTask.class);
        job.setReducerClass(ReducerTask.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        System.exit(job.waitForCompletion(true)? 0 : 1);
    }
}
```

in the distributed system is with a chunk size of 32MB. Obviously the centralized system, labeling with “1(1)”, has an execution time higher than the rest of computation (i.e.; the centralized) and can be considered the worst case in the system.

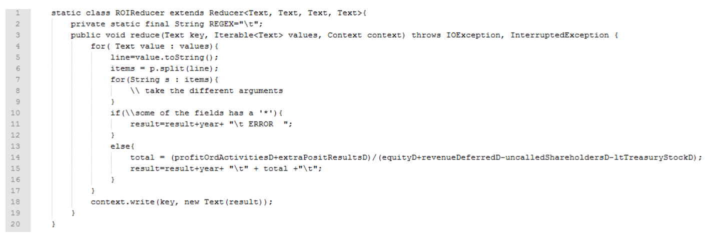
4.3.4 AWK Performance

To compare the execution time between a distributed system and a centralized system we decided to develop an AWK script (see Figure 4.11). The execution time of this AWK script using the same SABI file was 2 minutes 30 seconds. Thanks to this script we can certify that a big data repository is easier and faster with this distributed tool than with a traditional centralized application.

4.4 Evaluation

In this section we are going to evaluate the results obtained with our system. Thanks to the files that our system produces we could generate two tables (see Tables 4.2 and 4.3) to show the ccuracy of the data repository.

Figure 4.10: ROI Reduce Task



```
static class ROIReducer extends Reducer<Text, Text, Text, Text>{
    private static final String REGEX="\t";
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        for( Text value : values){
            line=value.toString();
            items = p.split(line);
            for(String s : items){
                \\ take the different arguments
            }
            if(\\some of the fields has a '*'){
                result=result+year+ "\t ERROR  ";
            }
            else{
                total = (profitOrdActivitiesD+extraPositResultsD)/(equityD+revenueDeferredD-uncalledShareholdersD-ltTreasuryStockD);
                result=result+year+ "\t" + total +"\t";
            }
        }
        context.write(key, new Text(result));
    }
}
```

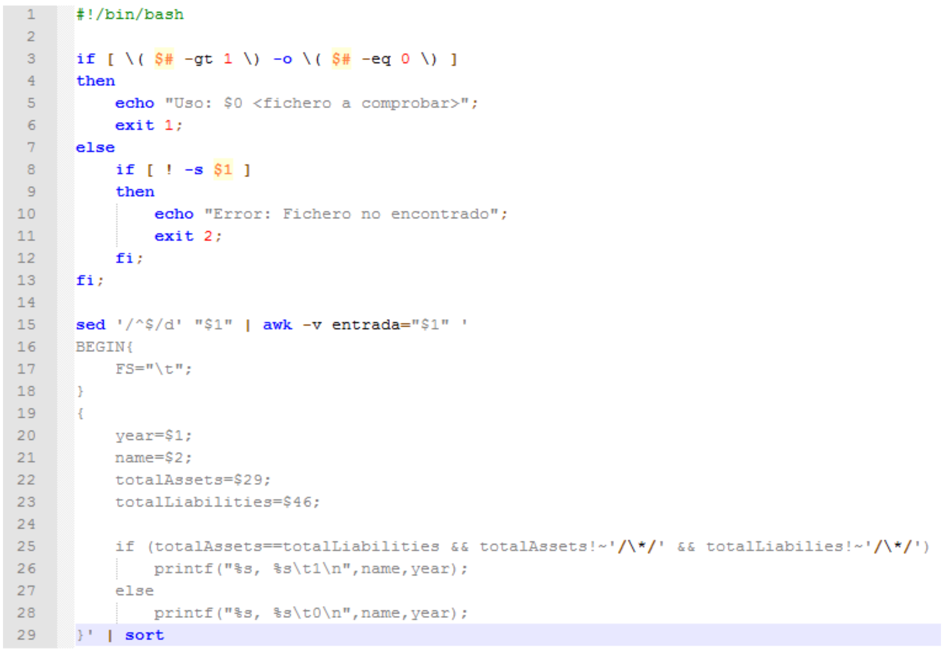
Table 4.1: Execution times in regard with the different configuration system

Chunk Size (MB)	64						128			32
# Rep (Rep Depth)	1(1)	3(3)	4(3)	4(4)	5(3)	5(4)	3(3)	4(3)	4(4)	4(3)
TOTAL ASSETS = TOTAL LIABILITIES	175	119	99	104	105	108	110	95	107	152
TOTAL ASSETS	178	125	99	101	107	109	117	94	97	120
CURRENT ASSETS	175	123	106	105	109	106	116	101	101	125
NONCURRENT ASSETS	174	117	101	104	106	105	114	98	97	124
TOTAL LIABILITIES	179	124	101	104	106	105	114	98	97	124
EQUITY	176	118	97	103	110	107	115	99	99	121
ROI	178	120	99	102	111	113	118	98	102	120

As Table 4.2 shows, the number of companies that have passed the computations in each of the activity sectors. Comparing these results with the TOTAL row from Table 3.1, we could see that most of the companies that had presented data have passed the $Total\,Assets = Total\,Liabilities$ check point. On the other hand, we verify the most hobbled verification was the comparison between the Current Assets and the fields to be considered as the contributors to its associated value. To obtain all the results we have used Hadoop too in the way we have explained in the previous section.

As Table 4.3 shows, we could infer that for each sector the percentage of companies that have data in their registry (N) with respect to the total sample and the percentage of companies that have passed all the check points in reference of the number of companies that have presented data (n). It is worth highlighting, that the companies that belong to the Energy sector have presented 50% of the data correctly and, on the other side, we could see that the companies in the Constructions & Dwellings sector have the lowest percentage with only a 40%. As a find remark, we could state that hardly the companies arrive to the 50% percentage which produce that only a 41% of our total sample have passed the computation.

Figure 4.11: AWK implementation



```
#!/bin/bash

if [ \( $# -gt 1 \) -o \( $# -eq 0 \) ]
then
    echo "Uso: $0 <fichero a comprobar>";
    exit 1;
else
    if [ ! -s $1 ]
    then
        echo "Error: Fichero no encontrado";
        exit 2;
    fi;
fi;

sed '/^$/d' "$1" | awk -v entrada="$1" '
BEGIN{
    FS="\t";
}
{
    year=$1;
    name=$2;
    totalAssets=$29;
    totalLiabilities=$46;

    if (totalAssets==totalLiabilities && totalAssets!~'/\*/' && totalLiabilies!~'/\*/')
        printf("%s, %s\t1\n",name,year);
    else
        printf("%s, %s\t0\n",name,year);
}' | sort
```

Table 4.2: Distribution of the repository with regard the sector and the different check points

	Agriculture	Industry	Energies	Construction & Dwellings	Services	Unclassified	Total Sample
TOTAL ASSETS = TOTAL LIABILITIES	5436	107770	5553	217407	239304	4	575474
TOTAL ASSETS	2814	56993	3367	39034	126991	2	229201
NONCURRENT ASSETS	2769	54822	3188	38169	122218	3	221169
CURRENT ASSETS	2176	44285	2840	29984	99924	0	179209
TOTAL LIABILITIES	2488	48410	2970	34007	112845	0	200720
EQUITY	2634	52249	3238	37272	117659	1	213053

Table 4.3: Description of the percentages of companies that have presented data and the ones that have passed the computation

	agriculture		Industry		Energies		Construction & Dwellings		Services		Unclassified		Total Sample	
	N	n	N	n	N	n	N	n	N	n	N	n	N	n
1994	1%	38%	32%	35%	1%	24%	11%	39%	55%	37%	0%	0%	1%	36%
1995	1%	34%	30%	35%	1%	26%	12%	38%	55%	37%	0%	0%	2%	37%
1996	1%	39%	28%	37%	1%	37%	14%	41%	55%	40%	0%	0%	5%	39%
1997	1%	41%	27%	38%	1%	37%	15%	42%	56%	40%	0%	0%	5%	40%
1998	1%	38%	26%	37%	1%	39%	15%	40%	56%	39%	0%	0%	6%	39%
1999	1%	38%	27%	38%	1%	39%	16%	40%	55%	40%	0%	0%	8%	39%
2000	1%	39%	26%	37%	1%	45%	16%	40%	55%	40%	0%	0%	8%	40%
2001	1%	39%	25%	42%	1%	56%	17%	41%	55%	42%	0%	0%	8%	42%
2002	1%	38%	24%	45%	1%	57%	18%	41%	55%	44%	0%	0%	9%	44%
2003	1%	42%	24%	44%	1%	56%	18%	43%	55%	43%	0%	0%	9%	44%
2004	1%	41%	23%	43%	1%	55%	19%	41%	55%	43%	0%	0%	9%	43%
2005	1%	43%	23%	42%	1%	56%	19%	40%	55%	43%	0%	0%	9%	42%
2006	1%	40%	23%	43%	1%	55%	19%	40%	55%	43%	0%	0%	9%	42%
2007	1%	43%	23%	44%	1%	56%	18%	40%	56%	42%	0%	0%	9%	42%
2008	1%	36%	26%	40%	2%	43%	16%	32%	54%	43%	0%	0%	2%	41%
2009	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
TOTAL	1%	40%	25%	41%	1%	50%	17%	40%	55%	42%	0%	0%	100%	41%

Chapter 5

Conclusions

5.1 Summary

Data driven applications are becoming more popular nowadays and the requirements needed to manage them are very stringent; huge volumes of data do not fit well in traditional database management systems. Cloud computing provides us the proper tools and infrastructure to manage data in a scalable and efficient way. In this paper, we have proposed a method to deal, not just storing but also computing, with large data repositories in the financial research field. This method consists in using the HDFS and MapReduce facilities to detect possible errors and recalculate values of the Spanish/Portuguese data repository (SABI) and to ease the computation of certain financial metrics.

This work has presented a more daily application of MapReduce which embraces economics calculation. However, there is still a long way until this usage becomes familiar to practitioners due to the difficulties of decomposing the problem in operations of mapping and reducing required to apply the MapReduce distributed computing paradigm. We hope our sketch encourages researchers to work on this direction and provide new insight into the field.

5.2 Future Work

Finally, our future research lines are two-fold: (1) to apply the same idea with upper layer Hadoop products such as HBase or Hive and compare which option is the best in terms of coding complexity and (2) to make a performance comparison analysing statistical tools such as R and SPSS.

Chapter 6

Related Publications

The work presented in this document has given rise to the following contribution:

- J. Navarro,A. Azqueta Alzúaz, P. Murta, J.E. Armendáriz Ínigo. Cloud Computing Keeps Financialmetrics computation simple.The 6th International Conference on Software and Data Technologies (ICSOFT 2011), July 18-21, Sevilla, Spain. Conference Proceedings (accepted for publication).

Bibliography

[1] Pablo Murta Baião Albino. Eficiencia y productividad de las cooperativas de crédito españolas frente al desafío de la desintermediación financiera. In *INTERNATIONAL, C. E. A. C. (Ed.) innovation and Management: Answers to the great challenges of public, social economy and cooperative enterprises*, 2008.

[2] Pablo Murta Baião Albino, Katrin Simón-Elorz, and Francisco Javier Arcelus-Ulibarrena. Posicionamiento de la industria hortofrutícola en la industria agroalimentaria española: un análisis shift-share. In *HORTÍCOLAS, S. E. D. C. (Ed.) VI Congreso Ibérico de Ciencias Hortícolas*, 2009.

[3] Apache. The apache software foundation, 2011. Disponible en: `http://www.apache.org/`.

[4] Apache. Hadoop, 2011. Disponible en: `http://hadoop.apache.org/`.

[5] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 205–220. ACM, 2007.

[7] Pablo Archel Domench, Fermín Lizarraga Dallo, Santiago Sánchez Alegría, and Manuel Cano Rodríguez. *Estados Contables:Elaboracin, anlisi e interpretacin(3th Edition)*. Pirámide, 2010.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 29–43. ACM, 2003.

[9] Google. Google app engine, May 2011. Disponible en: `http://code.google.com/intl/en/appengine/`.

[10] Isidoro Guzmán, Narciso Arcas, Rino Ghelfi, and Sergio Rivaroli. Technical efficiency in the fresh fruit and vegetable sector: a comparison study of italian and spanish firms. *Fruits*, 64(4):243–252, July-August 2009.

[11] G. Hernández-Cánovas and P. Martínez-Solano. Relationship lending and sme financing in the continental european bank-based system. *Small Business Economics*, 34(4):465–482, 2010.

[12] Instituto Nacional de Estadística. Cnae 2009. clasificación nacional de actividades económicas, May 2011. Disponible en: `http://www.ine.es/jaxi/menu.do?type=pcaxis\&path=/t40/clasrev\&file=inebase`.

[13] Magdalena Kapelko and Josep Rialp-Criado. Efficiency of the textile and clothing industry in poland and spain. *Fibres & Textiles in Eastern Europe*, 17(3):7–10, 2009.

[14] A. Martínez-Campillo and R. Fernández Gago. What factors determine the decision to diversify? the case of spanish firms (1997-2001). *Investigaciones Europeas de Dirección y Economía de la Empresa*, 15(1):15–28, 2009.

[15] Microsoft. Windows azure platform, May 2011. Disponible en: `http://www.microsoft.com/windowsazure/`.

[16] N. Monserrat, T. Palanca, M. Deppe, and B. Hartman. Replication Server: A component of SYBASE System 10. Technical report, SYBASE Inc., April 1993.

[17] Jose Luis Retolaza and Leire San-Jose. Efficiency in work insertion social enterprises: a dea analysis. In *Universidad, Sociedad y Mercados Globales*, pages 55–64, 2008.

[18] Jr. Rick J. Makoujy. *How to Read a Balance Sheet*. McGraw-Hill, 2010.

[19] Bureau van Dijk. sabi, 2011. Disponible en: `http://sabi.bvdep.com`.

[20] Tom White. *Hadoop:The definitive guide(1th Edition)*. OReilly Media, Inc., 2009.