# Building a distributed search system with Apache Hadoop and Lucene

Mirko Calvaresi

*a Barbara, Leonardo e Vittoria*

# Index

Preface ..... 5
1 Introduction: the Big Data Problem ..... 6
1.1 Big data: handling the Petabyte scenario ..... 6
1.2 Real Time versus Pre calculated Data ..... 7
2 The Map and Reduce Paradigm ..... 11
2.1 Common algorithms problems handled via Map And Reduce ..... 13
2.1.1 Distributed "Grep" ..... 14
2.1.2 Count of URL Access Frequency ..... 15
2.1.3 Reverse Web-Link Graph ..... 15
2.1.4 Term-Vector per Host ..... 15
2.1.5 Common links in a social tree: an advanced example. ..... 16
3 Apache Hadoop: a brief introduction ..... 19
3.1 Distributed Virtual File System: HDFS ..... 20
3.2 HDFS Client ..... 22
3.3 HDFS High-Availability ..... 23
3.4 Hadoop Job ..... 24
3.5 Admin console ..... 27
4 Implementation: The Distributing Indexing Approach ..... 29
4.1 Apache Lucene: Architecture ..... 30
4.1.1 Analyzer ..... 32
4.1.2 Document ..... 34
4.1.3 Index ..... 34
4.2 Query Parser ..... 37
4.3 Retrieving and scoring results ..... 39
4.4 The Lucene query system ..... 39
4.5 Distributing Lucene Indexes using HDFS ..... 42
4.6 The Cluster Set up ..... 49
4.6.1 Using Map-Reduce to populate the index ..... 51
4.6.2 The Jobs architecture ..... 52
5 Building up an administration console ..... 54
5.1 The architecture component ..... 54
5.2 The front end architecture ..... 54
5.3 Front-end design and architecture ..... 56

5.4 User experience ..... 57
5.5 The service level ..... 59
5.6 Information architecture and presentation ..... 61
6 Testing and comparing results ..... 63
6.1 Simulations ..... 67
7 Alternatives based on real time processing ..... 70
8 Bibliography ..... 74
8.1 Web Resources ..... 75

# Preface

This work analyses the problem coming from the so called *Big Data scenario,* which can be defined as the technological challenge to manage and administer quantity of information with global dimension in the order of Terabyte ($10^{12}$ *bytes*) or Petabyte ($10^{15}$ *bytes*) and with an exponential growth rate.
We'll explore a technological and algorithmic approach to handle and calculate theses amounts of data that exceed the limit of computation of a traditional architecture based on real-time request processing: in particular we'll analyze a singular *open source* technology, called *Apache Hadoop*, which implements the approach described as *Map and Reduce*.
We'll describe also how to distribute a cluster of common server to create a *Virtual File System* and use this environment to populate a centralized search index (realized using another *open source* technology, called Apache Lucene*)*.
The practical implementation will be a *web based application* which offers to the user a unified searching interface against a collection of technical papers.
The scope is to demonstrate that a performant search system can be obtained pre-processing the data using the *Map and Reduce paradigm,* in order to obtain a real time response, which is independent to the underlying amount of data.
Finally we'll compare this solutions to different approaches based on clusterization or No SQL solutions, with the scope to describe the characteristics of concrete scenarios, which suggest the adoption of those technologies.

# 1 Introduction: the Big Data Problem

## 1.1 Big data: handling the Petabyte scenario

According to the study "The Diverse and Exploding Digital Universe"[i] , the digital universe was in 2007 at 2.25 x 1021 bits (281 exabytes or 281 billion gigabytes) and in 2011 was expected to be 10 times bigger.
One of the more interesting aspects of the study is that the evolution of data in the digital universe evolves much faster then More's Law and so explosion of data produced principally by the users opens the way to a new paradigm of software component architectures.
In 2012, Gartner updated its definition as follows: "Big data are high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization[1].

Big Data is a common term used to describe projects involving an amount of data, which can't be stored or threated using a conventional machine or even a cluster and queried used a DBMS approach.

To give an example of what can be described as *Big Data,* is worth mentioning the actual dimensions of the most popular Social Networks:

Facebook, as example, declares to process 2,5 billion pieces of content and more than 500 terabyte of data each day, and up to 300 millions photos per day[2].

1 Douglas, Laney. "*The Importance of 'Big Data': A Definition*". Gartner. Retrieved 21 June 2012.
2 http://techcrunch.com/2012/08/22/how-big-is-facebooks-data-2-5-billion-pieces-of-content-and-500-terabytes-ingested-every-day/

Youtube, the popular platform for sharing and storing video, declares to have more than 72 hours of video uploaded per minutes[3].
Walmart handles more than 1 million customer transactions every hour, which is imported into databases estimated to contain more than 2.5 petabytes (2560 terabytes) of data.
Twitter from 2006 (when it was launched) to 2011 was up to 50 million tweets per day[4].
As can been easily understood, it's impossible to describe this kind of projects with a traditional architecture based on a standard multitier application, with a presentation layer, a data layer and a service layer.
Even the normal approach based on a clusterization of components like database or storage systems cannot be applied when the amount of data is so massive, because after a given dimension indexes or keys can't work properly (and so functions like sorting or grouping).

What has to be re-thought is the global algorithmic approach. In other words when scenario evolves to a Big data scenario the architecture has to be radically different.

## 1.2 Real Time versus Pre calculated Data

The traditional DBMS approach has been the central component, and still remains for the majority of the current web projects, of the so-called "*Multitier Architecture*"[5]. The important and the advantage of the database can be summarized using keywords *integrity, interactive access, structured.*
Databases, in fact, maintain structured data and offer consistency across the allowed operations, handle authorization of the users, backup, views and etc.

3 http://www.youtube.com/yt/press/statistics.html
4 https://blog.twitter.com/2011/numbers
5 http://en.wikipedia.org/wiki/Multitier_architecture

As widely adopted, the access to the database is managed by *SQL* (*Structured Query Language*) and therefore interoperability between different vendors is guaranteed.

Plus these days it's common to have software tools that actually make the SQL query even transparent to the application layer, converting database schema into an Object oriented mapping.

These tools are called *ORM* (*Object Relation Mapping*) and act as software proxies against the DBMS, adapting the query to the specific vendor etc.

Moreover database can be clustered and organized in a way to manage redundancy and fail over and handle a growing number of requests.

Again, though, this scenario is good to manage a linear data growth, but not an exponential one, and not the case when the data exceeds the physical limit of the DBMS.

There's another important aspect that has to be taken into consideration, and is the balance between *read* and *write* operations.

It's easy to demonstrate that most of the web activities of the users, as example, are "read only" transactions against the database, more than "write" ones, so they don't affect the data stored in the DBMS or present on the system. If we look at the list of the most accessed portals and website of the world, we can easily see that they offer the large majority of their contents in read mode, and just a small subset involves the data coming from the user.

As example let's take into consideration a simple home page of a major portal like BBC.com. It is very rich of contents coming presumably from a *Content Management System* with its own database where the *content types* like pages, news, links are stored.

Of course each time a visitor comes to the portal one or many transactions have to be opened against the CMS Database to retrieve all the data, which can involve different tables and schemas.

And this operation has to be theoretically repeated for each user generating a lot of traffic from the *presentation layer* to the *database*.

To avoid these problems, most of the current systems use a strong *cache approach*, which consists of loading data in memory and serving them directly from the server memory instead of invoking the database layer.
Sometimes the cache is distributed to a cluster of frontend servers, which act as first responders to the client, and at given intervals, ask the database layer to refresh their data.
This approach actually is based on simple *hash table* that links a resource key to its contents. If the entry for content it's not present in memory, a new request is made against the data layer.
We'll see that this is very close to what it's the Map and Reduce algorithm, but it's used just when the data is already stored and calculated.
What if this is not possible? For example there are scenario where simply database it's not applicable: each database as a physical limit, which is declared by the vendor[6].
Also the performance of a database depends strongly on the number of rows stored: a *join* between 2 tables with a very big number or records takes a longer time than one made against a small record set, etc.
All these aspects become crucial when the amount of data hits the "terabyte" limit, or exceeds the capacity of a single computer.
On the other side the users expect to have an immediate feedback from a system, no matter how complex and potentially articulated the request is.
In that case the only solution is to have architectures, which first calculate the data, and then serve them to the clients.
To fulfill this scope, as explained, it's required and algorithm and an infrastructure approach: the first to analyze the data and the second to distribute them and retrieve efficiently from a cluster of nodes.
The approach we're going to introduce is *Map and Reduce* and has its perfect application when data are unstructured, and that's because it is designed to interpret data at a processing time. In other words the input for this approach

6 For example according to the Oracle documentation the DB size can't be bigger of 4 Gb
http://docs.oracle.com/cd/B19306_01/server.102/b14237/limits002.htm#i287915

are not the intrinsic properties of the data, but the ones chosen by the implementation.

# 2 The Map and Reduce Paradigm

Map and reduce approach was originally proposed by Google [7] in order to describe *"a programming model and an associated implementation for processing and generating large data sets*."

The core steps of the algorithm are the Map and the Reduce function.

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs.

Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs.

The Reduce function, also written by the user, accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values.

The map function takes a value and outputs *key, value* pairs while the output functions merge the key into a list.

Conceptually they can be defined in the following formal way:

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

$$reduce\big(k_2, list(v_2)\big) \rightarrow list(v_2)$$

For instance, if we define a map function that takes a string and outputs the length of the word as the key and the word itself as the value then Map phase would produce a list of entry for each word with their respective length:

"$Hello$" $\rightarrow map(\text{hello}, 5)$

"$Cruel$" $\rightarrow map(\text{cruel}, 4)$

"$World$" $\rightarrow map(world, 5)$

7 http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/it//archive/mapreduce-osdi04.pdf

A common example of *Map and Reduce* approach is a simple "counting problem". Assuming we need to count occurrences of words in different sources we can use the map function to count each word and the reduce one to aggregate the results per key (which is the term itself).

In a distributed environment each *mapper* counts the occurrence e a single word in his resources emitting a $(word_k, 1)$ association.

The reduce function aggregates the reports creating the final report and therefore the $(word_k, \sum i\, frequence_i)$.

The implementation will the roughly the following:

```
function map(String name,Resource s):
 for each word w in Resource:
   emit (w, 1)

function reduce(String word, Iterator pCounts):
   counts   sum = 0
 for each pc in pCounts:
 sum += ParseInt(pc)
 emit (word, sum)
```

It easy to note that the *Map* function is stateless and therefore parallelizable and that's one of the most important characteristics that's makes this approach the perfect candidate for large and distributed systems.

The same stands for the *Reduce* function. In fact we can multiply maps and reducers which works in parallel as a distributed processing network. This is the core approach for parallelization implemented in Apache Hadoop as shown in the following picture.

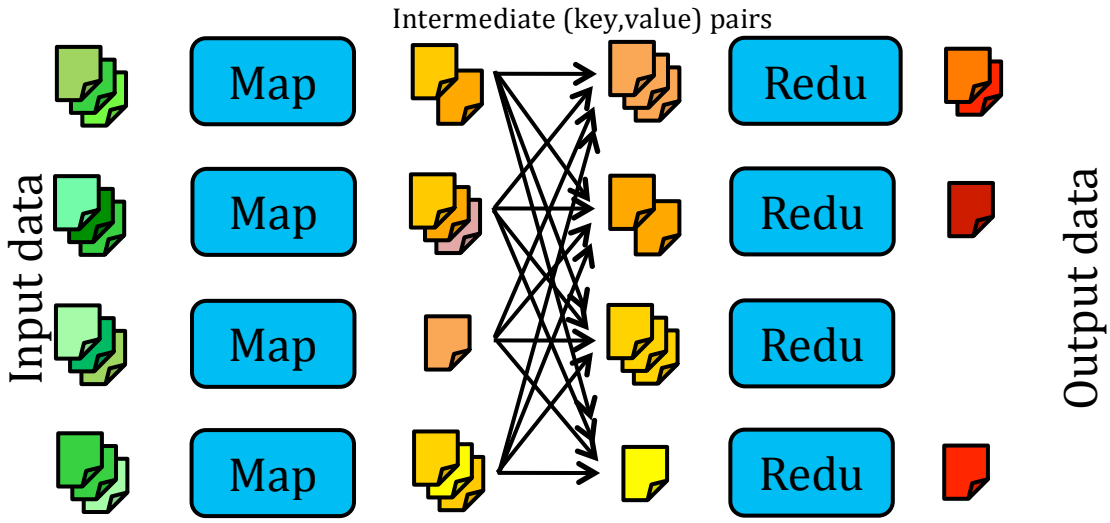



*Figure 1 Map and Reduce Tasks Suffle*

## 2.1 Common algorithms problems handled via Map And Reduce

As general consideration Map and Reduce approach can serve each algorithmic problem which computation can be parallelizable and aggregated at the end.
In the paper "*Map-Reduce for Machine Learning on Multicore*" [8] from University of Stanford, is analyzed the usage of that algorithm to solve some machine learning analysis, in particular for:

- *Locally Weighted Linear Regression*
- *Naive Bayes*
- *Gaussian Discriminative Analysis*
- *k-means*
- *Logistic Regression*
- *Neural Network*

8 http://www.cs.stanford.edu/people/ang/papers/nips06-mapreducemulticore.pdf

- *Principal Components Analysis*
- *Independent Component Analysis*
- *Expectation Maximization*
- *Support Vector Machine*

Since is a generally purpose analysis could be worth to show how to apply map-reduce to Principal Component Analysis[9].
Given a covariance matrix,

$$\sum \quad = \frac{1}{m} \left(\sum_{i=1}^{m} x_i x_i^T\right) - \mu\, \mu^T$$

It's possible to calculate the eigenvectors applying partial sum and then reducing to calculate the total.
In fact the term $\sum_{i=1}^{m} x_i x_i^T$ is already expressed in summation form, ma also the *mean* can be expressed as $\mu = \frac{1}{m} \sum_{i=1}^{m} x_i$.

The sums can be mapped to separate cores, and then the reducer will sum up the partial results to produce the final empirical covariance matrix.
Given these considerations, it's easy to conclude that it's possible to apply this algorithm to many operations regarding matrix analysis and transformation (matrix product as example).

In the following paragraph we'll show examples of common industrial problems than can be addressed using a Map and Reduce approach. Each of the following examples has an implicit assumption that the environment is distributed and quantity of data deserves a parallel processing.

### 2.1.1 Distributed "Grep"

This could be considered the simplest case when it's needed to *grep* a particular expression against a considerable amount of documents. In this case

9 https://en.wikipedia.org/wiki/Principal_component_analysis

the map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

### 2.1.2 Count of URL Access Frequency

Assume in this case that we want to count the URL access frequency in a set of log file (ad example Apache accesses logs file).

The map function processes logs of web page requests and output a simple $(\text{URL}, 1)$.

The reduce function adds together all values for the same URL and emits a $(\text{URL}, Total)$. Ad end of the reduction phase we have the aggregated value.

### 2.1.3 Reverse Web-Link Graph

In this case the goal is to have the list of the URL associated with a given target. The map function outputs $(target, source)$ pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $(\text{target}, list)$.

### 2.1.4 Term-Vector per Host

A term vector summarizes the most important words that occur in a document or a set of documents as a list of $(word, frequency)$ pairs.

The map function emits a $(hostname_1, term < V >)$ pair for each input document (where the hostname is extracted from the URL of the document).

The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a $(hostname_1, term < V >)$ pair.

### 2.1.5 Common links in a social tree: an advanced example.

Most of contemporary social networks implements the idea of list of contacts associated to an account.
These problems fallback into the general algorithms regarding graph visiting and search. One of the simplest way to explore a graph is the "Breadth first search[10]" which explores the graph using a per-level approach and has a time complexity of *O(n)* where *n* stands for the number of nodes in the graph.
That means that basically the algorithms evolves with a *time complexity*, which grows linearly with the size of the graph, and so it's hardly applicable to big graph like the social graph, i.e. graph representing relation between users of a very large community.

Map and Reduce proved to be a good algorithm strategy to implement this operations preprocessing the data and to reduce it to a common *key->value* data structure.
As example, let's consider extracting the common friends of a list of nodes in a social graph like Linkedin or Facebook.
Assume the friends are stored as Person->[List of Friends], the list is then:
A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D
Each line will be an argument to a mapper. For every friend in the list of friends, the mapper will output a key-value pair. The key will be a friend along with the person. The value will be the list of friends. The key will be sorted so that the friends are in order, causing all pairs of friends to go to the same reducer.
For map (A -> B C D) :

10 http://en.wikipedia.org/wiki/Breadth-first_search

(A B) -> B C D

(A C) -> B C D

(A D) -> B C D

For map (B -> A C D E): (Note that A comes before B in the key)

(A B) -> A C D E

(B C) -> A C D E

(B D) -> A C D E

(B E) -> A C D E

For map (C -> A B D E):

(A C) -> A B D E

(B C) -> A B D E

(C D) -> A B D E

(C E) -> A B D E

For map (D -> A B C E) :

(A D) -> A B C E

(B D) -> A B C E

(C D) -> A B C E

(D E) -> A B C E

And finally for map (E -> B C D):

(B E) -> B C D

(C E) -> B C D

(D E) -> B C D

Before being sent these key-value pairs to the reducers, they will be grouped by their keys and get:

(A B) -> (A C D E) (B C D)

(A C) -> (A B D E) (B C D)

(A D) -> (A B C E) (B C D)

(B C) -> (A B D E) (A C D E)

(B D) -> (A B C E) (A C D E)

(B E) -> (A C D E) (B C D)

(C D) -> (A B C E) (A B D E)

(C E) -> (A B D E) (B C D)

(D E) -> (A B C E) (B C D)

Each line will be passed as an argument to a reducer. The reduce function will simply intersect the lists of values and output the same key with the result of the intersection. For example, *reduce*((A B) -> (A C D E) (B C D)) will output (A B) : (C D) and means that friends A and B have C and D as common friends.

The result after reduction is:

(A B) -> (C D)

(A C) -> (B D)

(A D) -> (B C)

(B C) -> (A D E)

(B D) -> (A C E)

(B E) -> (C D)

(C D) -> (A B E)

(C E) -> (B D)

(D E) -> (B C)

Now when D visits B's profile, it is possible to look up (B D) and see that they have three friends in common, (A C E).

In simple words this approach reduce a *n level* graph in a redundant *hash table* which are the considerable advantage to have a constant time complexity for retrieving data (*O(1)).*

# 3 Apache Hadoop: a brief introduction

Apache Hadoop is an open source project that comes from Doug Cutting, the same inventor of Lucene, the most popular *open source* search library,[11] and its origin from another open source project, Apache Nutch[12], which originally was a web search engine originally created on 2002.

Apache Nutch project was very ambition[13] since the beginning and also from the first sketch of the overall architecture was clear that one of the most critical point was the scalability cost of the project, but also the simple possibility to store billions of entry in the indexes.

The solution came with the Google paper about Map and Reduce approach, so the team decided to operate a first porting of this implementation in Nutch in 2005. This project branched from Nutch in 2006 becoming an autonomous project called Hadoop.

The real success of Hadoop was due to the adoption of this project from Yahoo when the Cutting joined Yahoo.

In 2008 Yahoo announced to have a production 10000 cores Hadoop cluster[14].

As declared from the company those are the numbers managed:

- Number of links between pages in the index: roughly 1 trillion links
- Size of output: over 300 TB
- Number of cores used to run a single Map-Reduce job: over 10,000
- Raw disk used in the production cluster: over 5 Petabytes

Today Hadoop is used by a very large number of companies[15]: among the others Facebook declares to have:

- A 1100-machine cluster with 8800 cores and about 12 PB raw storage,
- A 300-machine cluster with 2400 cores and about 3 PB raw storage,

11 http://lucene.apache.org/
12 http://nutch.apache.org/
13 A story and the difficulties found in building this architecture can be found at http://dl.acm.org/citation.cfm?id=988408
14 Complete description of this first installation is at http://developer.yahoo.com/blogs/hadoop/yahoo-launches-world-largest-hadoop-production-application-398.html
15 http://wiki.apache.org/hadoop/PoweredBy

used mainly to parse logs and post process data.
Other companies like Linkedin, Last.fm, Rackspace etc, use Hadoop for different purposes varying from reporting to content analysis or machine learning. It's worth to mention that one of the really useful aspect of this technology is that really scales from small cluster to dimension that can be found just in a very few company in the entire world, and that's why adoption and implementation is growing as documented from many sources.

## 3.1 Distributed Virtual File System: HDFS

Apache Hadoop comes also with a distributed file system, which is the real core of the infrastructure. The nature of the HDFS (*Hadoop Distributed FyleSystem*) is to store data of big size. The architecture of the system is described in the paper "The Hadoop Distributed File System[16]"and defines the core of the architecture as follow:

*"Hadoop provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce paradigm. An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and executing application computations in parallel close to their data. A Hadoop cluster scales computation capacity, storage capacity and IO bandwidth by simply adding commodity servers".*

The real innovation in this architecture is to scale up to number potentially very large of nodes adding common components (commodity server) without depending a on a central mainframe or computer with extraordinary computations capabilities.
Since Hadoop implements in fact a file system, it stores application metadata and data separately.

16 http://storageconference.org/2010/Papers/MSST/Shvachko.pdf

The two main components of a Hadoop Clusters are the *NameNode* and *DataNode*.

*NameNode* is the server dedicated to store application metadata, while application data are stored in the *datanodes*. The communication is realized using the TCP/IP stack.

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented by *inodes*, which record attributes like permissions, modification and access times.

The entire *namespace* is stored in the primary memory (RAM) and it's used to map and address the single *inode* to the relative *datanode*.

Unlike other traditional File System implementation used on the operating system (like EXT4) the size of an *inode* in Hadoop is very large (64mb) and it's meant to have data of relevant dimensions.

The *name node* also stores the modification log of the image called the *journal* in the local host's native file system.

Two files represent each *inode* on the *datanode*: the first contains the data itself while the second is used to store checksum and generation stamp.

During the startup of the single *datanode* it makes and handshake with the *name node*. This process is used to verify the namespace ID and the to detect that there aren't version collisions.

The namespace ID assured the consistency of the *datanode* in the *namenode* tree memory structured, and therefore its reachability and interoperability.

In the distributed Hadoop architecture the namespace ID is a unique identifier, which sits on top of the IP network topology, meaning that remains unique and identifies the host even if the *datanode* changes IP address.

After handshaking *namenode* send *heartbeats* to the *datanode* to ensure that is still running within a short interval (3 seconds). Heartbeats are the small information messages passed across the network to retrieve the global status.

## 3.2 HDFS Client

Applications access the Hadoop network using a specific client.
HDFS supports operations to read, write and delete files and directories. The user references files and directories by paths in the namespace and so it's transparent to the application layer.
When a data read is required, the client first invokes the *namenode* for the list of *datanodes* that host replicas of the blocks of the file and list is sorted according to the network topology distance. The client contacts a *dataNode* directly and requests the transfer of the desired block. When there is a write operation it first asks the *namenode* to choose *datanode* to host replicas of the first block of the file.
On top of this mechanism there is an application client, which is syntactically similar to a command line interface of Unix Operating System, and allows executing most of the operations exposed to a normal file system, line creation, copy, delete, read both for file or folders.
Il Figure1 is represented the logical communication between *namenode* and *datanode* when the client is activated.

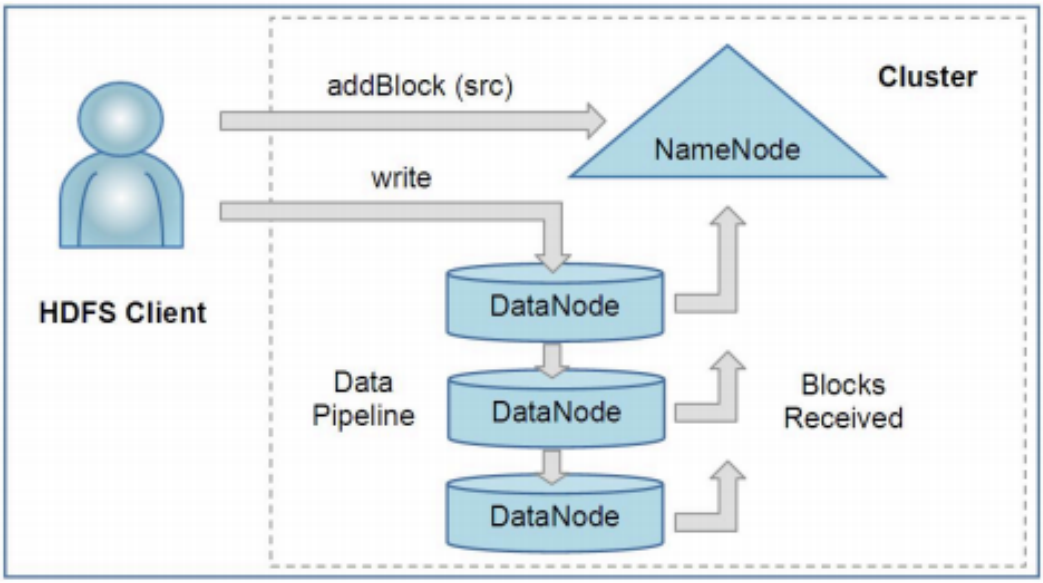



*Figure 2 The HDFS Client*

Examples of commands to access the client are the following:

```
hadoop fs -put [file] [hdfsPath]       Stores a file in HDFS
hadoop fs -ls [hdfsPath]        List a directory
hadoop fs -get [hdfsPath] [file]       Retrieves a file from HDFS
hadoop fs -rm [hdfsPath]        Deletes a file in HDFS
hadoop fs -mkdir [hdfsPath]     Makes a directory in HDFS
```

## 3.3 HDFS High-Availability

With this scenario it's easy to understand that *datanode* becomes a single point of failure (SPOF), since if it fails, all clients—including Map Reduce jobs—would be unable to read, write, or list files, because the *namenode* is the only and sole repository of the metadata and the file-to-block mapping. In such an event the whole Hadoop system would be in an inconsistent state or out of service.

A solution to this problem is to backup and recover the *datanode*, but event this scenario could present some consistent downsides because before having again the system running it is necessary that the new *datanode* has the namespace in

memory and have received enough block from the *datanode*. That could cost a not insignificant time on large clusters.
The 0.23 release series of Hadoop remedies this situation by adding support for HDFS high-availability (HA): a pair of *namenodes* in an active standby configuration. In the event of the failure of the active *namenode*, the standby takes over its duties to continue servicing client requests without a significant interruption. To accomplish this new scenario though:

1. The *namenodes* must use highly-available shared storage to share the edit log
2. Data nodes must send block reports to both *namenodes* since the block mappings are stored in a *namenode's* memory, and not on disk.
3. Clients must be configured to handle *namenode* failover, which uses a mechanism that is transparent to users.

Another important feature added from version 2.x is the HDFS Federation, which basically allows a single *namenode* to manage a portion of the file system namespace, like *"/user"* of *"/document"*. Federation allows a single *datanode* to manage a namespace's volume which is made up of the data in namespace and the blocks of the namespace itself.
Since this namespace don't communicate between each other the failure of one does not affect the others, so prevents that a single failure affects the total cluster.

## 3.4 Hadoop Job

Besides the HDFS, the other main component in a running installation, are Hadoop jobs.
The job is the actual software which implements the Map and Reduce approach against the data stored on the *datanodes*.

The job is usually a Jar file of a program implementing the two core interfaces: Map and Reduce. Using the usual example of the word count, it can be reduced to a single file jar where is implemented the map class:

```
public static class Map extends MapReduceBase implements
        Mapper<LongWritable, Text, Text, IntWritable> {
```

and the reduce class:

```
public static class Reduce extends MapReduceBase implements
        Reducer<Text, IntWritable, Text, IntWritable> {
```

Of course also the main file has some basic information about the input parameters to be passed to the job, like the output values and the input path where to find the input paths:

```
public static void main(String[] args) throws Exception {
            JobConf conf = new JobConf(WordCount.class);
            conf.setJobName("wordcount");
            conf.setOutputKeyClass(Text.class);
            conf.setOutputValueClass(IntWritable.class);
            conf.setMapperClass(Map.class);
            conf.setCombinerClass(Reduce.class);
            conf.setReducerClass(Reduce.class);
            conf.setInputFormat(TextInputFormat.class);
            conf.setOutputFormat(TextOutputFormat.class);
            FileInputFormat.setInputPaths(conf, new Path(args[0]));
            FileOutputFormat.setOutputPath(conf, new Path(args[1]));
            JobClient.runJob(conf);
      }
```

So the entire program will be just composed of the following code:

```
public class WordCount {

      public static class Map extends MapReduceBase implements
                  Mapper<LongWritable, Text, Text, IntWritable> {
            private final static IntWritable one = new
IntWritable(1);
            private Text word = new Text();

            public void map(LongWritable key, Text value,
                        OutputCollector<Text, IntWritable> output,
Reporter reporter)
                        throws IOException {
                  String line = value.toString();
```

```
                    StringTokenizer tokenizer = new
StringTokenizer(line);
                    while (tokenizer.hasMoreTokens()) {
                           word.set(tokenizer.nextToken());
                           output.collect(word, one);
                    }
             }
      }

      public static class Reduce extends MapReduceBase implements
                    Reducer<Text, IntWritable, Text, IntWritable> {
             public void reduce(Text key, Iterator<IntWritable>
values,
                           OutputCollector<Text, IntWritable> output,
Reporter reporter)
                           throws IOException {
                    int sum = 0;
                    while (values.hasNext()) {
                           sum += values.next().get();
                    }
                    output.collect(key, new IntWritable(sum));
             }
      }

      public static void main(String[] args) throws Exception {
             JobConf conf = new JobConf(WordCount.class);
             conf.setJobName("wordcount");

             conf.setOutputKeyClass(Text.class);
             conf.setOutputValueClass(IntWritable.class);

             conf.setMapperClass(Map.class);
             conf.setCombinerClass(Reduce.class);
             conf.setReducerClass(Reduce.class);

             conf.setInputFormat(TextInputFormat.class);
             conf.setOutputFormat(TextOutputFormat.class);

             FileInputFormat.setInputPaths(conf, new Path(args[0]));
             FileOutputFormat.setOutputPath(conf, new Path(args[1]));

             JobClient.runJob(conf);
      }
}
```

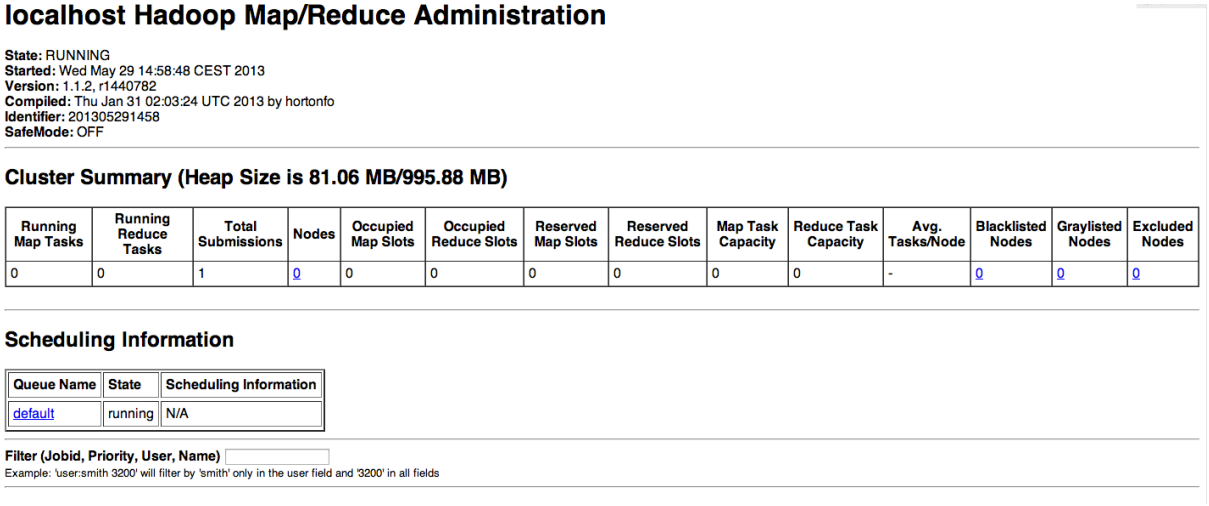
To run it on an Hadoop it is necessary to install it, using the HDFS client (passing the required parameters, and in particular the input folder and the output folder where results will be store):

```
bin/hadoop jar word_count.jar WordCount /user/mccalv/input_data
/user/mccalv/output
```

Once installed it produces the logs file regarding the map and reduce job and write the output of the reduceer to the relative folder ("/user/mccalv/output in this case").

## 3.5 Admin console

The installation by default comes with an administration console, which inspects the actual status of clusters, capacity, and jobs running on it. The administrative tools, which are part of the *suite,* have a relatively high impact when the dimension of the installation and number of nodes become high like the tree of folders in HDFS.



**localhost Hadoop Map/Reduce Administration**

**State:** RUNNING
**Started:** Wed May 29 14:58:48 CEST 2013
**Version:** 1.1.2, r1440782
**Compiled:** Thu Jan 31 02:03:24 UTC 2013 by hortonfo
**Identifier:** 201305291458
**SafeMode:** OFF

**Cluster Summary (Heap Size is 81.06 MB/995.88 MB)**

| Running Map Tasks | Running Reduce Tasks | Total Submissions | Nodes | Occupied Map Slots | Occupied Reduce Slots | Reserved Map Slots | Reserved Reduce Slots | Map Task Capacity | Reduce Task Capacity | Avg. Tasks/Node | Blacklisted Nodes | Graylisted Nodes | Excluded Nodes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | - | 0 | 0 | 0 |

**Scheduling Information**

| Queue Name | State | Scheduling Information |
|---|---|---|
| default | running | 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

The job page instead gives essential information about the job including all the properties and their values in effect during the job run.

The jobs results are stored by each single reduced by a file named "part-r-0000" to max "part-r-0029" in the output directory which is usually indicated in the configuration file. Since the amount of output produced by the reducer is usually very short a common way to get the total result is to *merge* the single outputs into one:

```
Hadoop fs –getmerge /output
```

This architecture describes a relatively simple network, but the complexity can be scaled to a higher level introducing a 2 level network topology with a top level switch and a lower level consisting or racks of node.

In cases like this one Hadoop should be aware of the network topology, represented internally with a *tree* with relative distances. Distances are used by

*name node* to determine where the closest replica is as input to a map task.

# 4 Implementation: The Distributing Indexing Approach

Scope of this section is to investigate the possibility and propose a solution to distribute search functionality to a Hadoop cluster, getting benefits from this combination primary in term of performances.

Typically a Search Engine consists of three main components: a crawler, an index and a search interface.

For the end user, of course, the only visible and accessible part is the search interface, which offers the high level access to the functionality.

The modern search engine, influenced from what done by major player like Google, Yahoo, Microsoft, offers a general purpose search which is usually presented as a *keywords based search*, which can eventually filter its results per *content type* (web pages, images, document).

In other terms, the search functionality is expected by the end user as a "semantic" attribute of the source and independent from the binary format of the inputs.

To fulfill these tasks is necessary to realize a centralized index, which stores all the documents, presents and provide a unique search API.

Besides this base component there are other really important aspects like relevance and scoring of the results with pretty much determines the validity and reliance of the toll itself at least from a user perspective.

In the next part of the work we'll introduce a standard, *de facto*, in the open source community for textual search, Apache Lucene.

Then comes the amount of data: how to search in a unified way against an amount of data, which exceeds the limit of physically single machine index capacity.

The solution proposed in this work is to integrate the algorithm approach

already expressed as Map and Reduce with the Lucene index.
The scope is to populate the index using the HDFS cluster and on the other hands, distributing and retrieving the query using the reduce part and eventually measures the benefits. We've already introduces the technology for *distributing* and *analyzing* the data, while in the section we'll introduce the one to search: Apache Lucene.

## 4.1 Apache Lucene: Architecture

Lucene is a very popular Java library for textual search. It was created again from the same creator of Hadoop, Doug Cutting, in 1999. Originally it was written in Java but these days several implementations are available for other languages like Python, C, C++, etc.
The core of Lucene and the reason why it is so popular is basically supporting full text searches against documents stored in various format, like PDF, HTML, Text and so on.
An overview of the index part of Lucene is show in Figure 3. As shown the basic flow starts from a document, which is binary file parsed, then analyzed and finally added to the index.
To convert a document and extract its content usually a *parser* is used. A parser is component capable of extracting the textual content of a document from its binary format.
In fact search library are mostly text based, meaning the input for the analyzing is a *text string,* but of course we intend the term *document* as a binary file with specific extensions (PDF, HTML, DOCX) and opened and edited by specific software.

There are several projects available to provide parser for binary proprietary files, one of the most important and also integrated into Lucene is Apache

Tika[17].

The main operation that a parser does is:

*parse (InputStream, ContentHandler)*

which is the basic operation of converting ad input stream into a *ContenHandler*, which is a textual representation of the file.

The parser is the generic interface for all the format specific parsers that can be invoked according to the *mime type* of the input document. In other words a specific parser has to be instantiated in order to parse an Html page and extract its content out of the html tags, or for a PDF, a word document an so on.

Once parsed, the text extracted is ready to be analyzed by Lucene. The following picture represents the general flow of documents, which, coming which different extensions, one parsed, can be *analyzed* and finally populate the Index.

17 http://tika.apache.org/

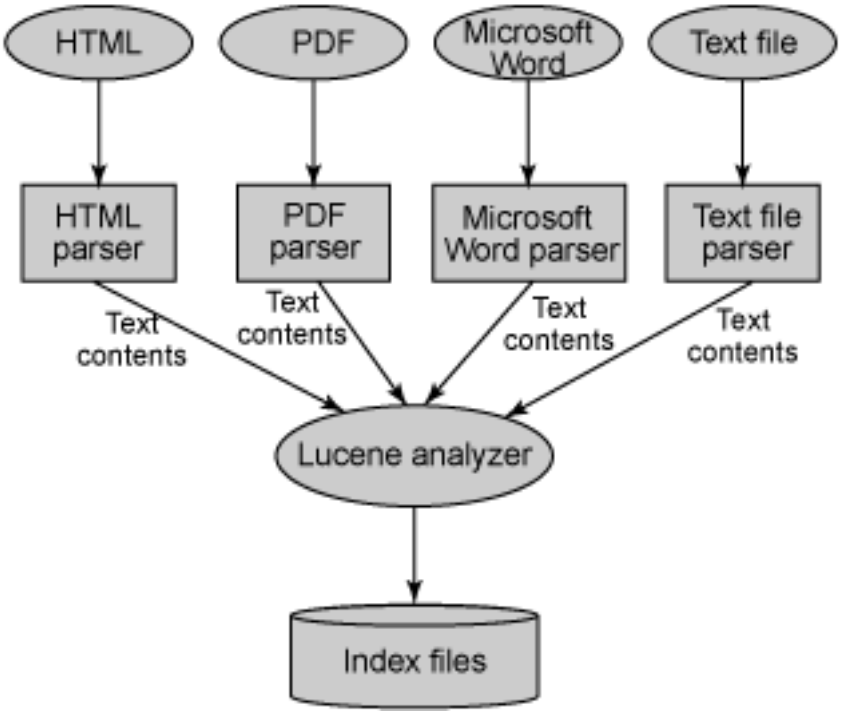



*Figure 3 Apache Lucene Index Population*

### 4.1.1 Analyzer

The analyzers are the software components used to tokenize text, i.e. the operation to reduce a text into a list of text units (token) which can correspond to a word but can be also a character sequence, a phrase, an email address, URL, etc.

Some tokenizers such as the *Standard Tokenizer* consider dashes to be word boundaries, so "top-level" would be two tokens ("top" and "level"). Other *tokenizers* such as the *Whitespace tokenizer* only considers whitespace to be word boundaries, so "top-level" would be a single token.

Tokenization is a bottom up operation, different from techniques that consider first the whole document, then the sentences to parse.

Analyzers can do more complex manipulations to achieve better results. For example, an analyzer might use a token filter to spell check words or introduce synonyms so that searching for "search" or "find" will both return the same

entries in the index.

As example we applied two different analyers (WhiteSpace and Standard) to the same input string "*The world is indeed full of peril and in it there are many dark places"*[18].

In the first case we have this ouput:

world (4 9)
indeed (13 19)
full (20 24)
peril (28 33)
many (54 58)
dark (59 63)
places (64 70)

While in the second case, the output was the following, including all the words like articles ("The"), prepositions ("of"), adverbs ("there"), auxiliary verbs ("is", "are"), which have been removed from the first analyzer because considered *stop words*: words that appears so frequently in a language, since they represent a consistent part of the syntax, and so to lose their function for search.

The (0 3)
world (4 9)
is (10 12)
indeed (13 19)
full (20 24)
of (25 27)
peril (28 33)
and (34 37)
in (38 40)
it (41 43)
there (44 49)
are (50 53)
many (54 58)
dark (59 63)
places. (64 71)

Of course the list of stop words isn't language neutral: different languages have

18 J.R.R. Tolkien, The Lord of the Rings

different lists, and both referenced *Analyzers* uses English as default language.
The architecture of Lucene allows though the creation of custom analyzers, which are language or *locale* sensitive, and use it instead of the provided ones.
The second part of the Lucene architecture is the Index and is the place where all the information is stored. To introduce the index we need to first describe the information unit which populates the index: the *document.*

### 4.1.2 Document

Lucene provides a general abstraction layer to add entries to the index: the document. As basic definition document can be described as a sequence of *fields,* each one is a <*name,value*>*pair*.
Different types of *fields* control which values are just stored from values *analyzed* and therefore *searchable.*
It's clear that *document* is a high level wrapper for a "piece" of data, and doesn't give any indication to the dimension or the nature of the information contained, and, of course, there isn't any connection to a *document* of the file system that originally contained all or part of that information.
In other words, it's an implementation decision which kind of granularity applying during the indexing process.
Some applications with specific searching needs may want to decide to have a *1-1* association between input documents and entries in the index, or have multiple entries (paragraph for example) from an initial document.

### 4.1.3 Index

The index is the place where the information is stored, and can be stored physically to a file system folder or in a memory directory: for our purposes we'll limited our exploration to persisted indexes.
A Lucene index is made up of 1-*n* segments. A single segment is just short of a self contained inverted index. The segmented nature of a Lucene index allows

efficient updates, because rather than replacing existing segments, it is possible to add new ones. These segments can be eventually merged together for more efficient access. Obviously the number of segments impacts the performance of the search.

Moreover a fewer segments also mean a many fewer open files on the file system by the same process, which prevents the OS to raise IO exceptions.

Of course, it is possible to use the compound file format, which writes out segments in a single file, significantly reducing the number of files in the index.

Optimizing a Lucene index is to operation to merge individual segments into a larger one. This makes searching more efficient – not only are fewer files touched, but with a single segment, which are a good performance impact because prevents Lucene to repeat basic setting operation for each segment.

Another strategy for maintaining a low segment count is to use a low merge factor on the *IndexWriter* when adding to the index. The merge factor controls how many segments an index can have: having a level lower than 10 be usually a good tradeoff.

The algorithmic approach to handle an incremental index is described by Coutting as follow:[19]

- incremental algorithm:
- maintain a stack of segment indices
- create index for each incoming document
- push new indexes onto the stack
- let b=10 be the merge factor; M=∞

In detail the algorithm can be expressed by this pseudo code:

```
for (size = 1; size < M; size *= b) {
  if (there are b indexes with size docs on top of the stack) {
   pop them off the stack;
   merge them into a single index;
   push the merged index onto the stack;
 } else {
```

19 http://lucene.sourceforge.net/talks/pisa/

```
   break;
 }
}
```

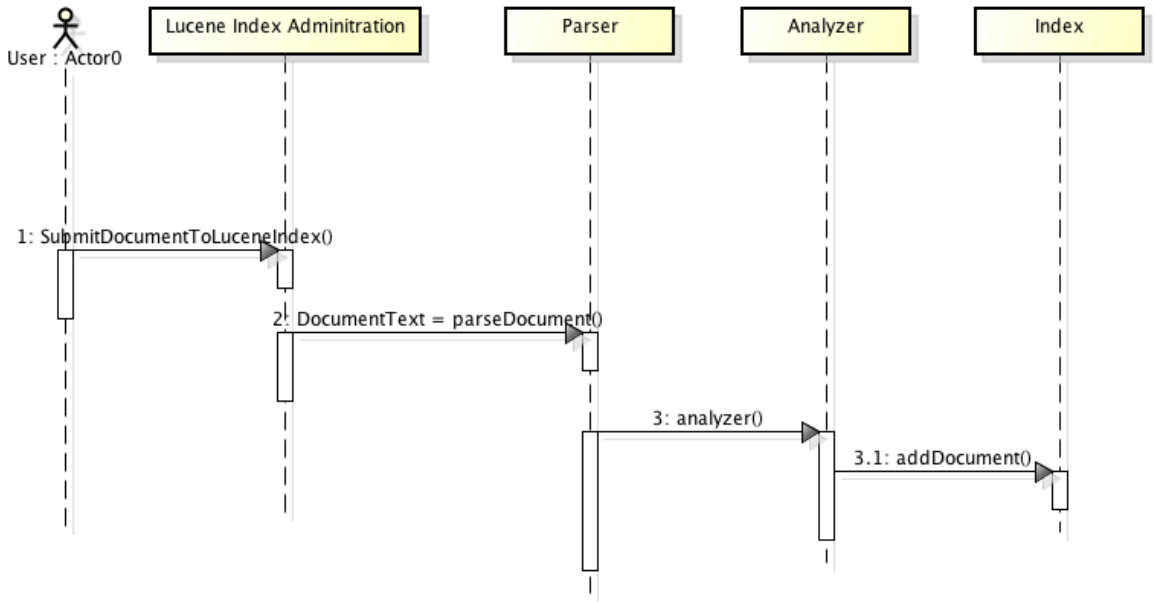
In other words the indexed are aggregated and pushed back on a data structure (*stack)* used to maintain the indexes list.

For a given merge factor of *b*, and N documents, the number of indexes can be expressed by the formula:

$$b * \frac{\log_b(N)}{2}$$

To conclude, the operation of creating an entry on an index has 3 major phases, the text extraction and parsing, the analysis and the creation of the *document.*

The following sequence diagram resume the logical sequence of the




operations:

*Figure 4 Submit and Indexing of documents in Lucene (UML sequence diagram)*

## 4.2 Query Parser

The query mechanism in a search engine technology is, of course, a core feature for comparing and evaluate the business value.
Searching the index in performant way it's the reason that gives to these tools a comparative better results than a traditional database approach based on standard SQL restrictions and Boolean operators.
The input of Lucene query is a text string, which is interpreted and translated by a component, the *QueryParser*, into a *Lucene Query*.
For example, the query:

*"(hobbit OR elves) and legends",*

Is translated into

*"+(contents:hobbit contents:elves) +contents:legend",*

Where "*contents*" is the name of the field in the document object and "+" means that the term must be present in the matching document. This Lucene query is then processed against the index.
Scope of the Query Parser is to Turns readable expression into Query instance.
The query types supported are:

**Term Query**

A term query is the base query and matches document containing a term. So the expression *"+content:legend*" is converted in *TermQuery("content", "legend")* which basically retrieves all the documents containing the term "*legend"* in the field "*content"*.

**Boolean Query**

In the original queries there were also Boolean operator like AND, OR; these operator can be combined using the Boolean Query which basically combines Term Queries in conjunctive or disjunctive way.
The Boolean Query is therefore a container for a list of term query restrictions:

$BooleanQuery(\quad).add(termQuery_i, BooleanClause).add(termQuery_j, BooleanClause)$ …

**Phrase Query**

In case it is necessary to match a number of terms in a document, it's possible to use phrase query with search against documents containing a particular sequence of terms.

Of course the difference between this query and a Boolean Query with two terms, as example, is that this query matches only document that have these terms in the sequence without any intervening others.

**Wild Card Query**

It's the particular query which contains a special meta character, like *, known as a trailing wildcard query: the system searches for any document containing the term and replacing the * with any possible sequence of characters, while to perform a single character wildcard search the symbol "?" is used

These kinds of queries are useful in case of singular/plural search of a term (in English), problems with the term spelling, and so on.

**Fuzzy Query**

To perform a fuzzy search it is necessary to use the tilde, "~", symbol at the end of a word term.

Fuzzy search are based on "Levenshtein Distance" or "Edit Distance algoritm"[20], which basically computes the number of single characters edit to transform a query into another.

Since distance can be measured, this query supports also a normalized parameter for example "legend~0.8" (while the default is 0.5)**.**

**Range query**

Range query are specific for those fields like *Date Field,* or numeric values for

20 http://en.wikipedia.org/wiki/Levenshtein_distance

which makes sense to search against an interval rather than to a specific value.
As last example it is worth mention the **proximity search** which support the search of words with a specific distance. An example is *"hobbits elves"~5* which searches for " *hobbits* " and " *elves* " within 5 words.

It's pretty visible, even from this short list, the gap and potential that a search system has if compared to a normal DBMS approach, where the only restrictions "*like"* based are absolutely inadequate to cover all these terms relations.

## 4.3 Retrieving and scoring results

## 4.4 The Lucene query system

Query system is a core part of what Lucene, and in general for any search libray. We've already introduced the concept of *document*, which is the base entry of an index, now it is necessary to introduce the concept of document similarity and in general the way the search is performed against an index.
Lucene scoring uses a combination of the Vector Space Model[21] (VSM) of Information Retrieval and the Boolean model to determine how relevant a given Document is to a User's query.
According to the Vector Space Model, each document is represented by a vector in a multidimensional space where the dimensions corresponds to a separate token together with its weight.

$$dj = (w_{1,j}, w_{2,j}, \dots, w_{1,j})$$
$$q = (w_{1,j}, w_{2,j}, \dots, w_{t,j})$$

21 http://en.wikipedia.org/wiki/Vector_Space_Model

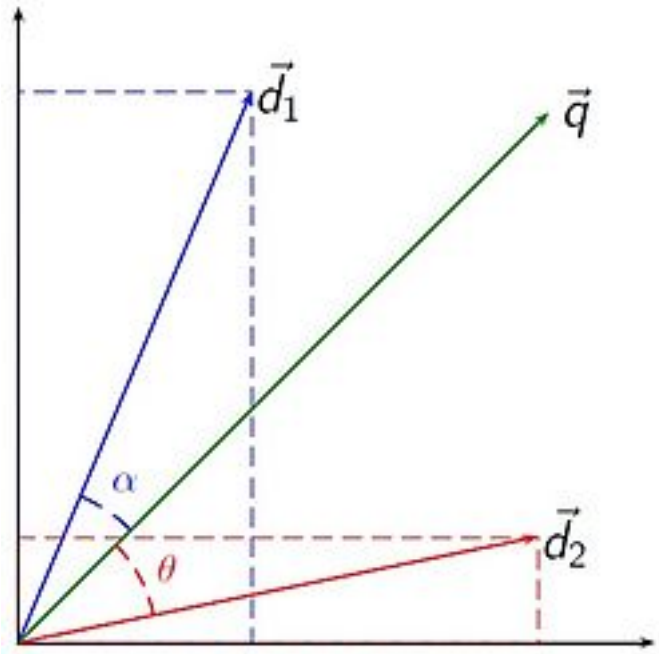



*Figure 5 Lucene query system*

A measure of the relevance is the value of the $\theta$ angle between the document and the query vector,

$$cos\theta = \frac{d_2 * q}{||d_2||||q||}$$

Where d2 is the vector representing the document, q the vector presenting the query, $||d_2||, ||q||$ are the norms, calculated respectively as follow:

$$||q|| = \sqrt{\sum_i^n q_i^2} \quad ||d_2|| = \sqrt{\sum_i^n {d_2}_i^2}$$

In general, the concept behind VSM is the more times a query term appears in a document relative to the number of times the term appears in all the documents in the collection, the more relevant that document is to the query. It uses the Boolean model to first narrow down the documents that need to be scored based on the use of Boolean logic in the Query specification.

The score of a document is a product of the coordination factor (*coord*) and the sum of the products of the query and term factors for all query terms that were

found in the document to be retrieved[22].

$$DocumentScore = coord * \sum_{i} (q_i * t_i)$$

Lucene allows the possibility to boost the result, which is basically the operation of altering the result of a query according to a major or minor weight given respectively to a document (*document boosting),* a field (*field boosting)* or a query (*query boosting).*

The boosting may occur at indexing phase (in the first 2 cases) or query level.

Boosting a field is a decision which of course has a large implication in terms of perceived effectiveness of the system: from the user perspective, in fact, the order of the results is extremely important, probably more than the completeness of the set retrieved by the search.

In other words, scope of a search application should be to show the interesting results for the user as first in list.

22 For a description of the algoritmic implementation, the complete documentation can be found at http://lucene.apache.org/core/3_6_2/api/core/org/apache/lucene/search/Similarity.html, which is part of the core documentation of the framework.

## 4.5 Distributing Lucene Indexes using HDFS

The objective of this section is to describe a practical implementation of Map Reduce approach, in detail the aim is to build a Lucene based search web portal using Apache Hadoop.

The core implementation will be to design a distributed index using HDFS and populate and query it using the Map and Reduce approach.
The input information to search against is a collection of technical papers, distributed over a Hadoop Cluster. The sources selected for the technical papers are extracted using the project Core [23] developed by the KMI (Knowledge Media Institute, London).
The description and mission of the project is:
"*CORE (COnnecting REpositories) aims to facilitate free access to scholarly publications distributed across many systems*".
The scope is therefore to collect, from different sources, the highest possible number of technical papers and distribute them using remote API over different channels (mobile, web, etc.).
An interesting part of the project is the possibility to download documents using a *Restful client* [24](over an HTTP connection). Using that connection and some concurrent threads we downloaded approximately 2GB of technical papers accessing the Core remote APIS.
So the input will be a collection of technical papers, parsed, indexed in Lucene and distributed using HFDS.
Of course, for experimental reasons, the dimension of the documents library must be comparable to a "Big Data" scenario, which is of course practically difficult to achieve with a limited number of machines used for the scope of this analysis.
From implementation and algorithm point of view, however, the solution

[23] http://core.kmi.open.ac.uk/search
[24] http://en.wikipedia.org/wiki/Representational_state_transfer This kind of approach has become very familiar in the last years because widely used as component part of a *Service Oriented Architecture* usually as alternative to SOAP.

described is built to scale up to potentially amount of data comparable with that scenario.

The final objective is to give to the user a unified interface against the query system dispatching the *Lucene* query through the Cluster.

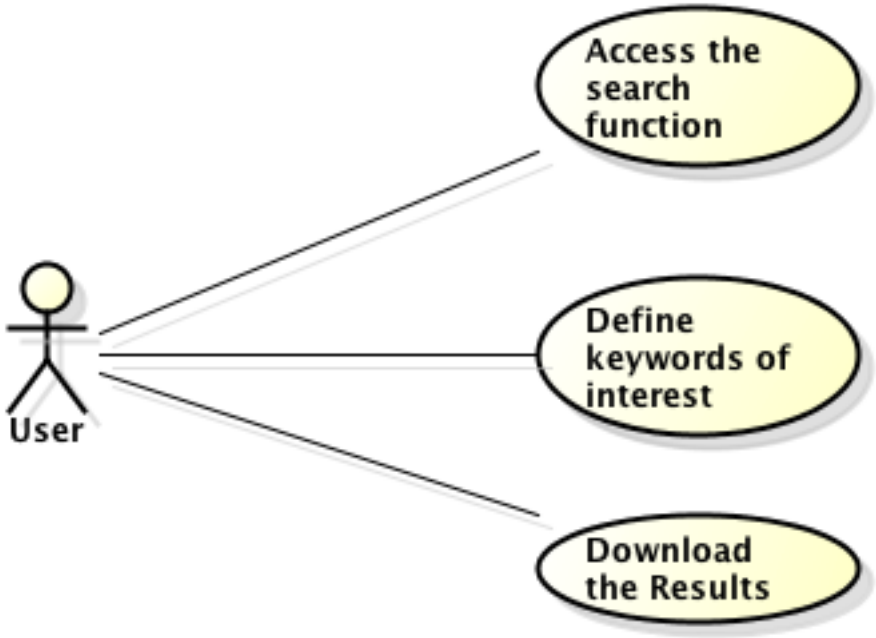



*Figure 6 Front end main use case*

From a user perspective we want to build a *web based* system where the users search for technical papers performing a keyword based search, retrieve the results and download the original papers.

*Keyword based search system* are those kind of system where users submit one or more keywords used to filter the results.

We've already introduced the Boolean model as part of the restriction that can be used with *Lucene*, so for example a list of words that *must be* or *can be* present in a list of documents.

Modern search engines basically in most of the cases hide this complexity from the user side and propose a simple unique text field: we use Google usually trying to match a single term or part of a phrase and so in detail applying the

kind of restrictions that, in Lucene terminology, are *Term query,* or *Phrase query*.

It is worth mentioning that we are analyzing *unstructured data,* so data extracted from a binary document or a textual document, and not from a database, which can be queried using the known SQL queries.

We're not using either an approach based on *knowledge representation* and *reasoning*, for example using an *Ontology* and a *TripleStore:* in this case each document added to the system should be annotated by an operator, generally human, against the *a priori* knowledge representation decided.

What we are analyzing are amount of data, coming in a number that would be extremely hard to annotate manually, and so has to be searched using the technologies already described before which match *keywords* against and *index*.

This use case will be a simple Lucene based search engine, if the index was unique, but we have multiples indexes, one for each *data node* and the search has to be represented as unified and consisted to the user.

The users access a *web based interface* and perform the following operations:

- Define a list of keywords
- Launch a search against the system
- Retrieve the results

We assume the operator of the service is an academic interested on a list of papers afferent to his domain of interest, or researcher exploring a new technology, for example. In these use cases the system need to find the papers matching the user's profile and proposing through the web interface or *pushing* them using other channels.

There is a list of consequences coming from these requirements:

1. It is necessary to get the user profile, therefore store the user interests
2. Having the user profile, results can be delivered asynchronously, for example to the user email or the user dashboard.

In particular the second point is crucial for the technology we're using: as we'll describe in the following sections, most of the analysis are made against a

*cluster of nodes,* using jobs triggered by an input, so could be a delay time from the request and the result: in this case the system will prompt the result to the user later using different channels.

In conclusion, most of the search will produce their results asynchronously.

The sequence diagram describes a basic flow of operation for the actor user. The first operation required is to register to the portal: the operation allows to get the information essentially to grant access to the portal and to match the library of grants documents to his profile.

In this phase there is the presence of an extra component, which is the portal DBMS used to store user profiles and credential and eventually to manage dynamic document or information needed by a Content Management System.

This aspect could be in conflict with the architecture described which claims to be not driven by a DBMS approach, which instead, is confirmed: the online database is just a support one for utility information that we want to keep separate from the Lucene indexes and in general from the Hadoop in order to have a reasonable *separation of concerns*[25].

Once registered the user enters a basic flows based on mechanism of pushing results and prompting results to his dashboard.

The user profile is converted into a *SearchFilter*, which is basically a representation, in term of Lucene query, of the user keywords of interest. The query is performed against *the frontend Hadoop Distributed Cache*.

The system is based, in fact, on a *distributed cache mechanism*: once the query is performed a mechanism similar to the normal cache is applied: first it is verified if there is a hit on the cache, and in case it's not present it is propagated to the backend were the Map and Reduce job is executed and finally a new entry is added to the distributed cache.

The entire system is based on a *messaging system* and a list of *triggers* activated by a new *hit* or a *miss* on the cache: for this reason the system is built up incrementally in a way to serve an increasing numbers of request on real time directly and without invoking the *backend*.

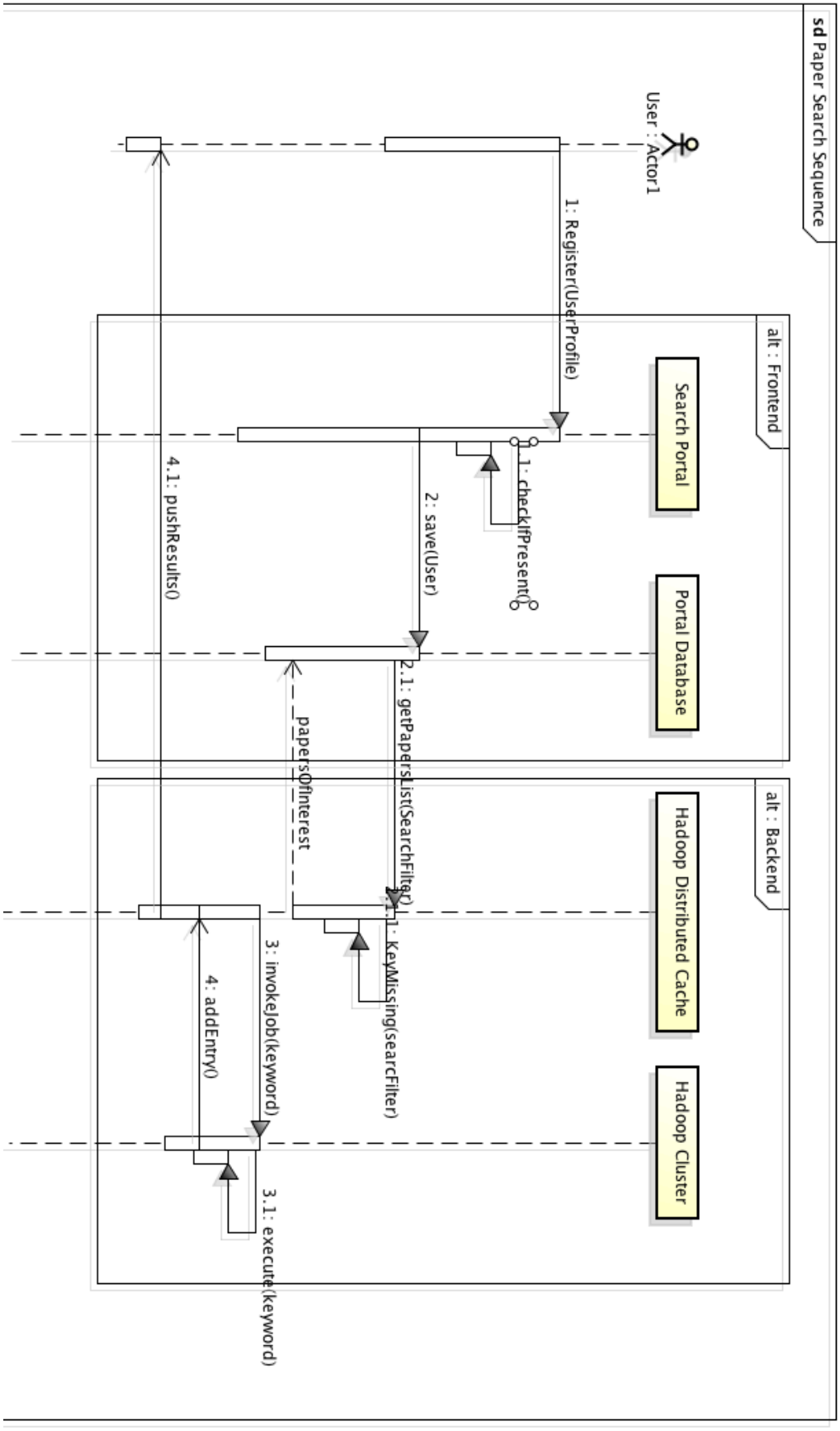
25 http://en.wikipedia.org/wiki/Separation_of_concerns

The cache could be associated in a way to an *inverted index* which links a *searchFilter* (which is a serialization of a list of *terms restrictions*) to a list of technical papers:

$$SearchFilter_{term_i} \rightarrow \{TechnicalPapers\}_{sFil_i}$$

Obviously using this approach would be possible to create intersection or union of the *result sets:*

$$SearchFilter_{(term_i\ AND\ term_j)} \rightarrow \{TechnicalPapers\}_{sFil_i} \cap \{TechnicalPapers\}_{sFil_j}$$
$$SearchFilter_{(term_i\ OR\ term_j)} \rightarrow \{TechnicalPapers\}_{sFil_i} \cup \{TechnicalPapers\}_{sFil_j}$$


sd Paper Search Sequence
User : Actor1
alt : Frontend
Search Portal
Portal Database
alt : Backend
Hadoop Distributed Cache
Hadoop Cluster
1: Register(UserProfile)
1.1: checkIfPresent()
2: save(User)
2.1: getPapersList(SearchFilter)
papersOfInterest
3: invokeJob(keyword)
3.1: execute(keyword)
4: addEntry()
4.1: pushResults()

The components present on the system are shown in the following diagram, together with their dependencies. The architecture proposed has a clear division between the *frontend*, which is the component essentially dedicated to extract input and presenting the result to the user, from the *back end,* which is where the data is analyzed, extracted and the indexes populated. As already declared in the first section, the solution proposed has a clear and marked distinction between the *read* operations, basically delegated to the cache mechanism, and the *write* operation (in our case represented by Lucene index updates), which are done by different component and distributing on a scalable cluster.

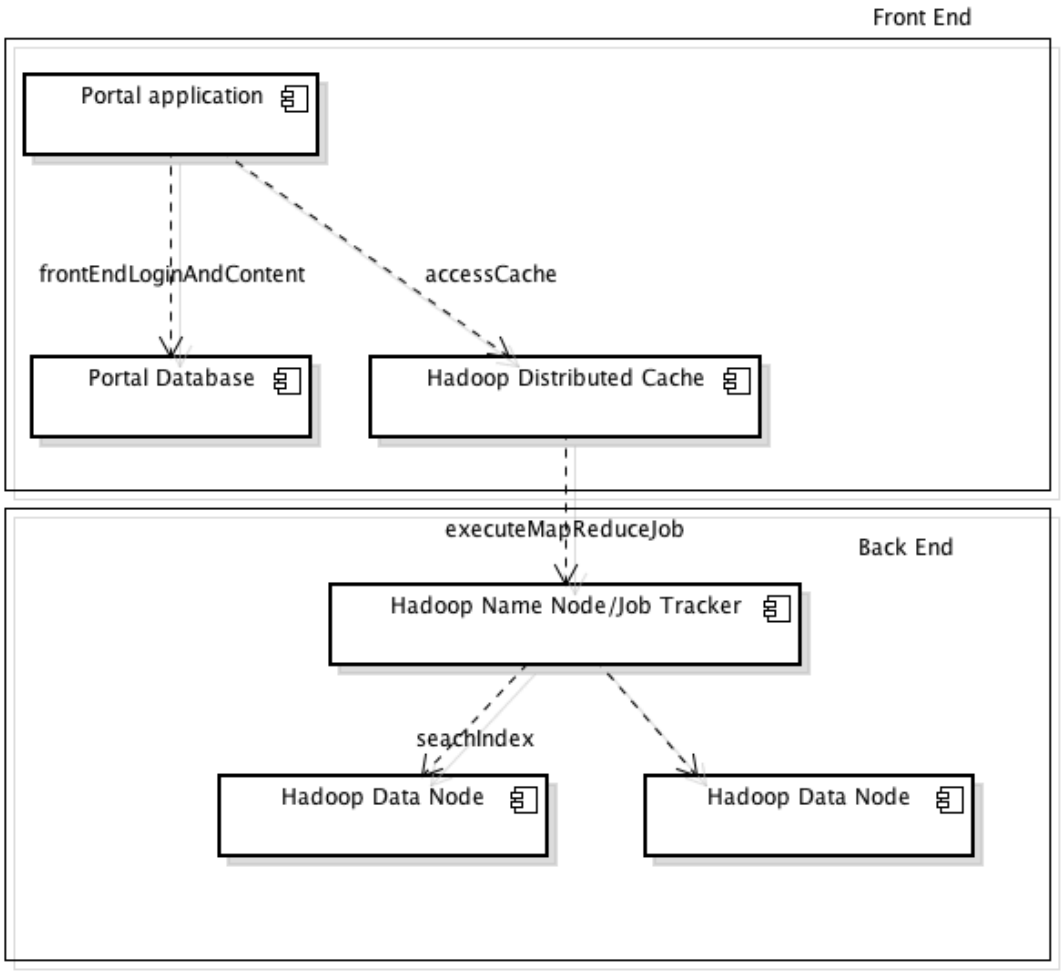



*Figure 7 Portal component diagram*

## 4.6 The Cluster Set up

To set up a Hadoop Cluster, it' necessary to have a sufficient number of machines, in order to distribute the computation and store resources among them.

We've already introduced the necessity that Hadoop is aware of the network topology chosen: the place where is declared is the list of *XML configuration files* where the *name node* and *data node, masters* and *slaves,* are indicated.

For our analysis we used a set up made by a name node, and a cluster of 2 *data node*s. By definition, the *name node* has the control and loads in memory the virtual file system.

Access through the single nodes is managed by SSH:

Secure Shell (SSH) is a network protocol based on cryptographic access used to remote command login and remote command execution.

The way the servers in the cluster access and execute commands on the other nodes is using an authentication based on RSA certificate: the *name node(s)* certificate, in fact, is listed in the list of the *authorized hosts* by the other servers in the Hadoop cluster.

Using RSA certificate (with 1024 bit encoding) is considered a secure way to login into a Unix based system (because of the combination of private and public keys) and also allows the *local* machine to execute remote command on the *remote* one automatically and without using a password-based login (like in a normal terminal).

This architecture allows, for example, starting simultaneously all nodes from the master. Master and slaves are concept familiar also to Hadoop: in particular master and slaves have a role both in the Map and Reduce Layer but also in the HDFS layer.

The configuration of the master and slaves roles is based on a declarative approach based on simple list of machines or hosts accessible from the same network.

In our case the entire cluster shares the same Operating System which is Linux Ubuntu 12.04, but this is not strictly necessary: what is required, in fact, is a uniform version of Hadoop.

When invoked, the startup procedure triggers the communication between the nodes in order to ensure that are reachable and accessible.

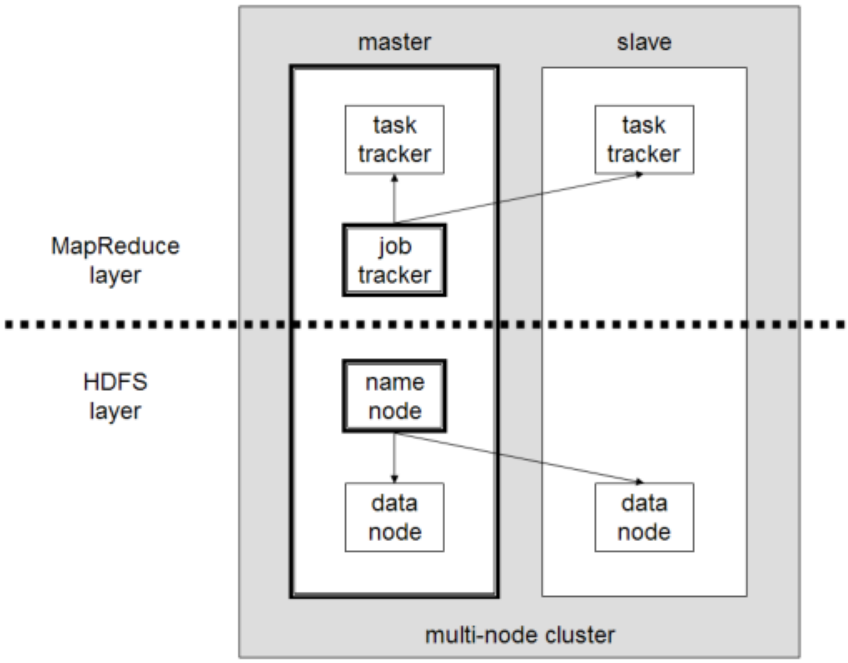



*Figure 8 HDFS task and job Tracker*

As clearly stated in the documentation, it´s necessary to have the same version running on all machines: we decided to opt for version 1.2.0 (last *1.** version stable).

Hadoop is completely written in Java (like Lucene), and so it's necessary also to have the same version of the *Java Run Time Environment* on the entire clusters: our network has the version 1.6 of the JRE provided by Oracle.

The architecture of the nodes is shown in the previous pictures: Name node is the also the master while the other 2 machines are slaves.

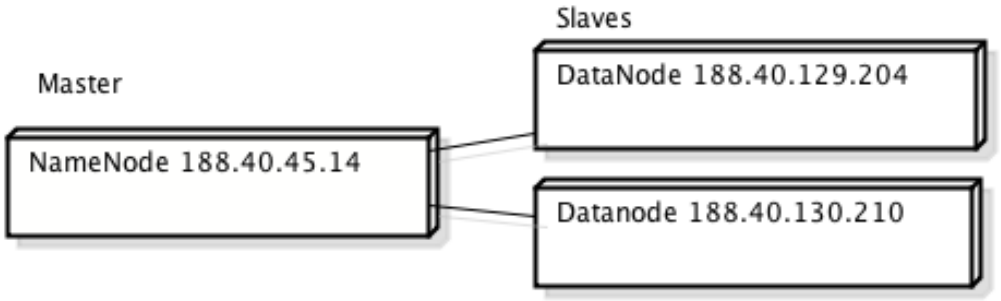



*Figure 9 Master and Slaves*

Having a setup made of multiple nodes can theoretically lead to concurrency problems: multiple nodes accessing the same file at the same time.
To avoid these kinds of potential deadlocks, the mechanism implemented is called "lease" and used originally in the Google File System
When a client contacts a data node to update an index, the data node tries to acquire a lease from the name node. If it fails, then it returns an error message indicating that a lease cannot be obtained, and therefore the client will try other replicas in order to find one.
Data nodes request a lease as part of the *heartbeating*, which is the bit of information exchanged at a fix interval between data node and name node: if a data node fails, the lease will become available again for other nodes. Consequently any data that was written to the index prior to the failure will be lost, so the client needs to take responsibility for uncommitted data.

### 4.6.1 Using Map-Reduce to populate the index

As already said we'll have a number of index equals to the *data nodes* present on the system (actually we could decide to use also the *name node* as part of this distributed index architecture, since it is part of the HDFS and have visibility of the virtual folders present).
The first operation has to be the one to populate the distributed Lucene index.

Assuming we divide equally the library of technical papers for each node we could store data in the index using a Map task. The Map task will emit an entry for each document into populating the index.
The Map phase of the Map/Reduce job formats, analyzes and parses the input (in parallel), while the Reduce phase collects and applies the updates to each Lucene instance (again in parallel). The updates are applied using the local file system where a Reduce task runs and then copied back to HDFS.
For example, if the updates caused a new Lucene segment to be created, the new segment would be created on the local file system first, and then copied back to HDFS.
When the job is completed the "new index" is ready to be queried. From a performance perspective is very important that they are created incrementally, so adding or updating documents but not rebuilding everything from scratch.
To distribute the index, it's necessary to assure consistency of some of the main components involved which are the *parser*, *analyzer* and the *input paths.*
The aim is to ensure that each *PDF file* representing a technical paper is parsed, analyzed and added to the local index in the same way for any *data node* registered to the cluster.
Actually this scope is fulfilled distributing the same job over the network, because the Lucene implementation is part of the "map" phase and therefore is a code dependency of the same bundle (the job *jar* file in detail).
Scope of the *reduce* phase is to propagate the query against the entire cluster and collect the results, ordering it by relevance and score.
The reduce function therefore for each key represented by the single index results, merges the two result set ordering it in the inverted index.

### 4.6.2 The Jobs architecture

For the purpose just mentioned we require at least two jobs: the indexing job which is the one used to build the n-indexes and search job which merge the result of the query and insert a new entry on the cache, and they need to be executed in order.

To orchestrate different jobs on a single instance there are several approaches possible: the most common is simply to indicate a sequence of jobs in the main configuration: this approach runs the jobs in a strict sequence: the second one starts only when the first one is completed.
There is also the possibility to design a workflow when there are several jobs running with a more complex and conditioned sequence.
Of course, in our case, a declarative approach will be enough an so there's no need to have a complex jobs sequence.

# 5 Building up an administration console

## 5.1 The architecture component

Scope of this section is to describe how to build the front-end architecture, both from a user experience and system perspective.

So far, we've analyzed a way to access and analyze data, using the Map and Reduce approach, now we want to illustrate how to deliver the search function using a distributed cluster and present it using a *web application*.

In general, having a system with owns a massive amount of data is theoretically independent from having a relevant number of users accessing the system, but of course, in practice, it's quite common to have this concurrent situation in particular because in most of the case are users the ones who generate the contents[26].

It would be also extremely complicated to justify an existence of this infrastructure without an adequate traffic and business implications.

To design this frontend architecture there are two main components that has to be studied:

1. *The front end architecture*
2. *The user experience*

## 5.2 The front end architecture

We already mentioned the fact that we access the inverted index of technical papers using a distributed cache sitting on top of the HDFS file system

The frontend application is though a web application hosted on of a web server, i.e. an application dedicated to distribute contents and responds to HTTP requests.

The entire technology stack used so far is based on the Open Source software

26 Actually this assumption is particurarly true when the system has a lot of *user generated data:* the most common example is a social network.

and we opted of course also in this case for the Apache Web Server and Apache Tomcat Server to build up the web application.

In particular the frontend application is a J2EE application distributed to a list of frontend servers, which can be compared to of Content Delivery Network.

For simplicity we assume to have a single load balancer which dispatches the request to a list of frontend servers Apache, which again proxy the request to the application server which deploys the application; of course when the scenario scales up again, and it's common to have a multiple load balancers or even multiple WAN used to delivered the content-.

This is the case when the data is not stored on cluster afferent to a single location but can be distributed geographically in different data center located in strategic point of the globe, in order to have the shortest path between the *request* and the *response* and serve them independently from their geographical origin.

The application, in the end, is the one that connect the distributed cache and respond to the server, so processes the requests (HTTP request, since our system is *web based),* and produces the response. The same package is delivered to each *front end server,* so each application server loads, using the cache, the inverted disturbed index and act as *shard* for the application.

The entire architecture is based on fail over strategy and a natural redundancy: in fact each frontend can eventually be detached without affecting the entire capability of the system to remain substantially stable.

An essential picture of the system is shown in the following diagram:

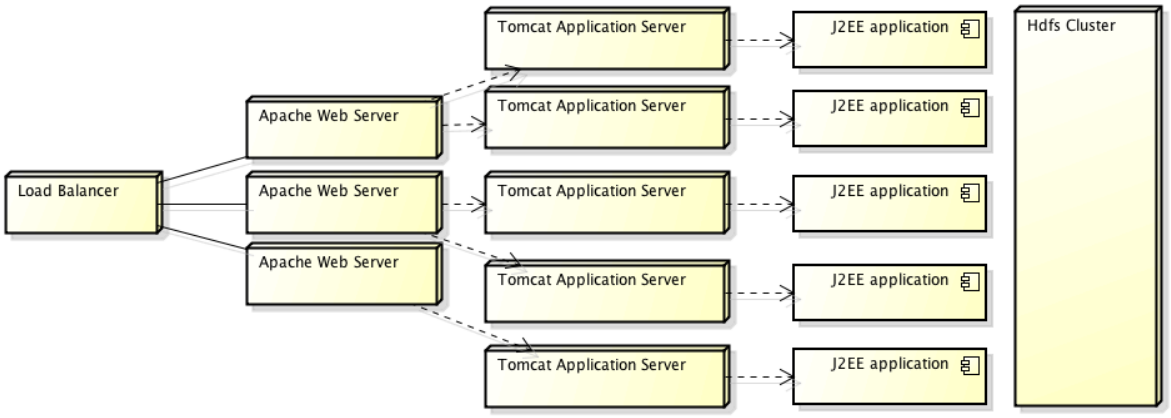



*Figure 10 Frontend servers architecture*

## 5.3 Front-end design and architecture

From the user perspective side we need to build the application in order to have different level of users and roles available.

The first role involved in the system is the *front end user*, which is the principal business actor of the system. As already said, the user accesses the main search functionality after the registration, which is a single operation of transferring the keywords of interest into the utility database to the system.

We're already mentioned that the frontend has a utility database which is functionally different from the Lucene index and stores user profiles and other information required from the portal.

For the scope of our analysis the use of this database was merely due to clearly separate the core business logic that informs the data processing from the one that on the other hands, is necessary to maintain a portal based on restricted access.

This database though, when the system becomes very big, could be a potential *bottle neck* for the same reason that already discussed and consequently we could apply the same solution: again Map and Reduce.

Actually we could also use a specific technology based on Hadoop to handle

distributed data base, in particular HBASe[27,] which is a distributed database on top of the Hadoop Cluter. Introducing this kind of technology is out of the scope for our work at the moment so we won't go into the details of this approach which basically is designed to see the entire cluster as a distributed DBMS which can be queried using a dialect similar to SQL, but of course without some of the advanced feature of a DBMS like transactionality, complex schema, etc.

## 5.4 User experience

From the information architecture we need to build a list of pages necessary for the basic set up of the system. In particular the main entry point will be:

- *Registration page*
- *Login page*
- *Search page*
- *Dashboard*

We've designed possibility that in case the result of the research is not present on the front end cache it can be delivered asynchronously using the user email or pushing a notification on the user dashboard. The front end user access the system having both options after the login to save a result set of data or perform a real time search against the system.

The scope of this functionality is to store previous searched performed by the user or, in case of a *miss* on the cache, to delivery to the user email and present to the dashboard for the next login.

27 http://hbase.apache.org//

A general dataflow of the user is shown in the following picture:

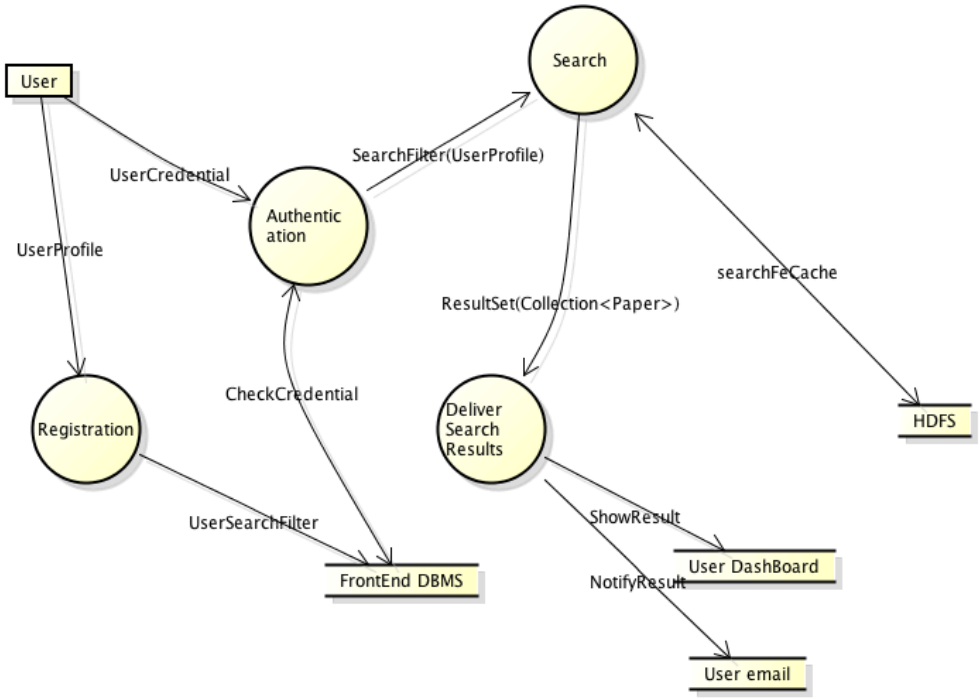



*Figure 11 Front end data flow*

A part For the front end users it is important to sketch the portal section dedicated to the administrative roles, which are:

- *Administrator*
- *Editor*

The administration console is the place dedicated to the Cluster maintenance, to rebuild the cache and check the status of the system. Hadoop provides some tools to monitor the cluster and check the status of the distributed file system: the administration console extends those functionality with also some functionalities to export and visualize data.

Like any file system we need to make sure that the all application is working properly, so eventually substitute the damages nodes, or add nodes when the total capacity of the system is reaching a security alert limit, which informs the administrator to extend the dimensios. Of course an important part of the system administration is to monitor the system analytics: number of patents,

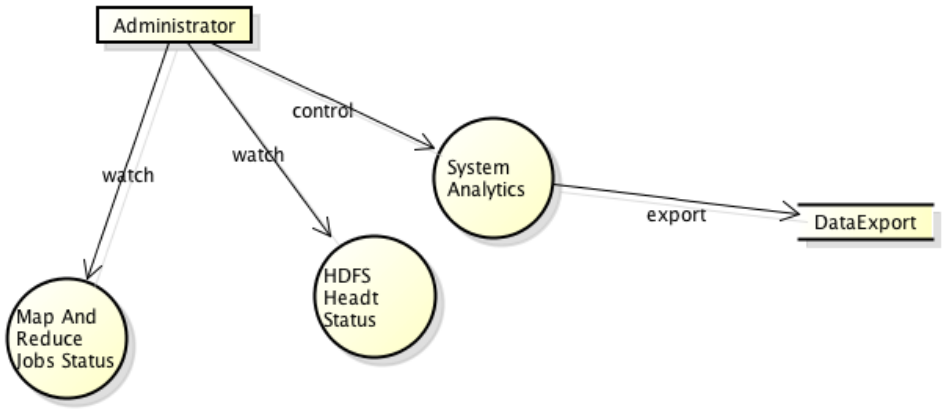



*Figure 12 Administrator Data Flow*

request per unit time, unique visitors.

All these features has to be delivered using a report tool which can *export* its data using a specific visualization (graphs, histogram and so on) or formatting data according to well known standard like CSV (*Comma Separated Value)* or Microsoft Excel.
The editor role is common in any frontend system because he *moderates* the contents and the user, and eventually operates as reporter for the administrator. An editor acts as support for the system and performs operation, which are related to the specific communication strategy of the portal, or the system design.

## 5.5 The service level

Finally it's worth to mention a different level of access that the platform can offer: the service layer.
We've introduced so far a design of the system based totally on a web based access, but most of modern system, influenced by the WEB 2.0 approach, offers their service as remote service, there *API*S (*Application Programm Interfaces* interoperable by remote machine call using *Soap* or *Restful services*.

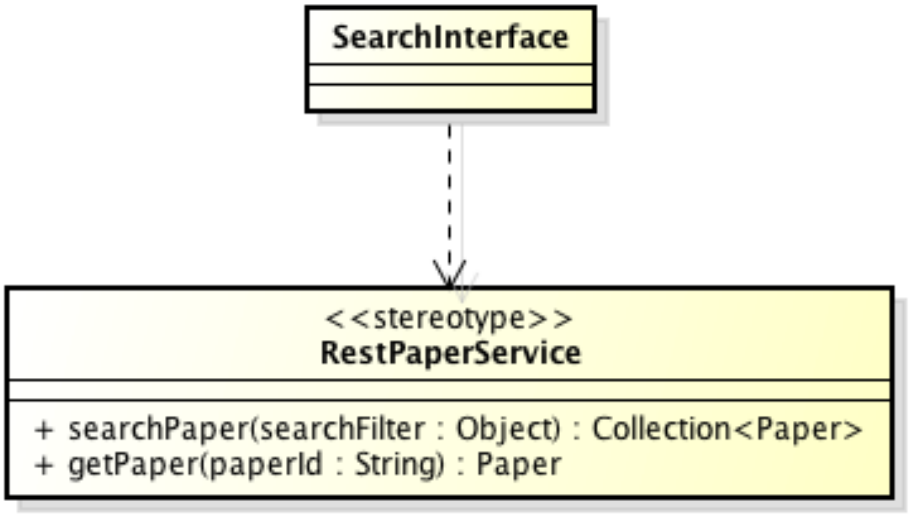



*Figure 13 Remote API Class Diagram*

This service level is particular actual and useful when the potential clients of the application are not just the web users, but also users who access the system using either mobile devices or non-standard interfaces: in this case it is necessary to exchange communication between the server and the client using remote calls over HTTP which return a *content representation* of the data using XML or JSON[28].

In other word a special section of the portal, for example separated with a special *URL pattern (service/**  ad example*)* will be dedicated to expose services using interfaces dedicated to machine. To design this part it is necessary to expose the core functions of the portal and deploy them as service:

- The search function
- The "get" paper function.

A basic usage of the platform will be based on SOAP or REST. In the first case the method is invoked using XML as envelope while in the second place the method is invoked using the HTTP methods (GET, POST, PUT, DELETE) and

28 JSON stands for *Javascript Object Notation* and is a way to represent an object using Javascript notation. An example is the following:

```
{
        name: 'Mirko', surname:'Calvaresi'
}
```

passing the object representation as input, and in same way for the output.

## 5.6 Information architecture and presentation

In order to complete the design of the frontend it's necessary to introduce the interface design, and the user experience. We'll not investigate the graphical implementation but we'll propose minimal information architecture for the front end user profile. The output of the design will be the *wireframes* of the two main pages of the system: the login page and the user dashboard. In particular the *dashboard* is the place where the user, after the login, accesses the main search functionality but also checks the detail of the single paper and eventually downloads it. We introduced at this point some options to *share* the results among other users: in fact it is possible to serialize a search filter an consequently share it across the network using for example the same XML or JSON format used for the *Rest Api*. From usability perspective it's essential that the dashboard is a *place* where each function is easy visible and accessible, links are marked properly and the working area is clearly indicated to the user.
From a technology perspective, the frontend design will be strongly influenced by the evolution of the interface paradigm introduced in the last years by the major players with wide adoption of HTML5 and JAVASCRIPT in particular.
The use of these technologies allows the user to have the feeling of immediate response from the system, avoid the waiting time necessary for the normal cycle of page processing and rendering to the client.
This perception of immediate response, also declined up to the interface level enforces the sense of immediate answer from the system, which is possible just with a pre-processing of the data.

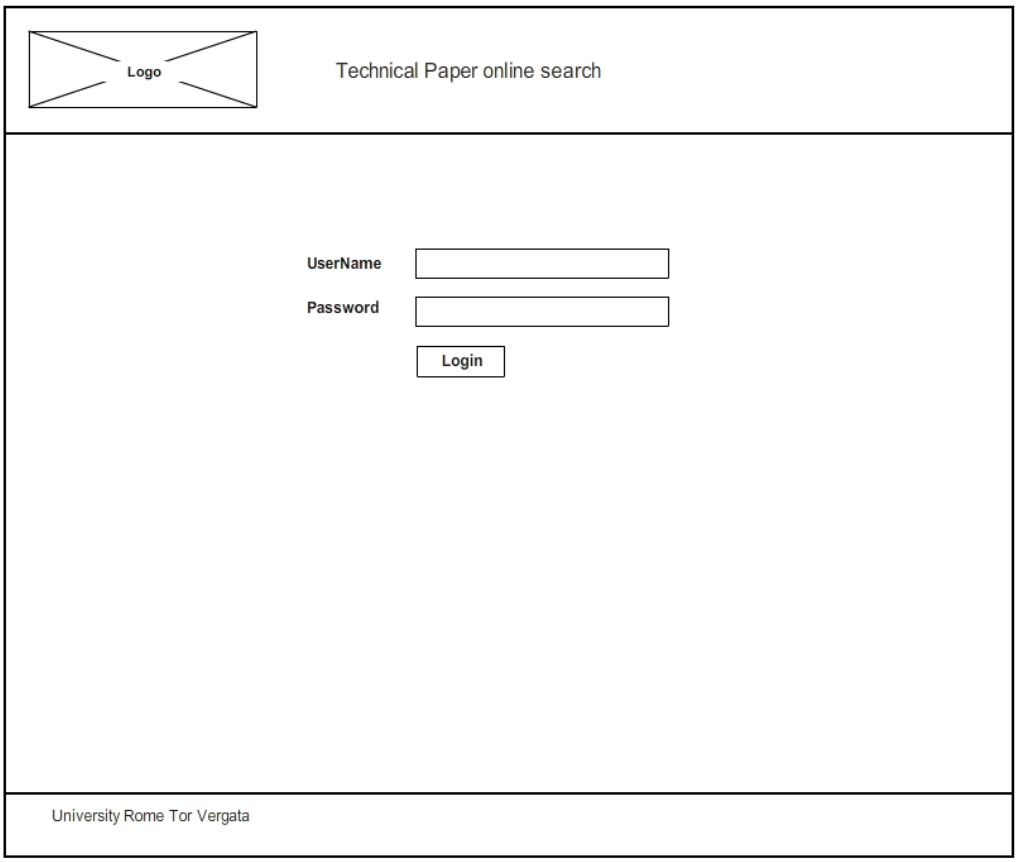



*Figure 15 Login Page*

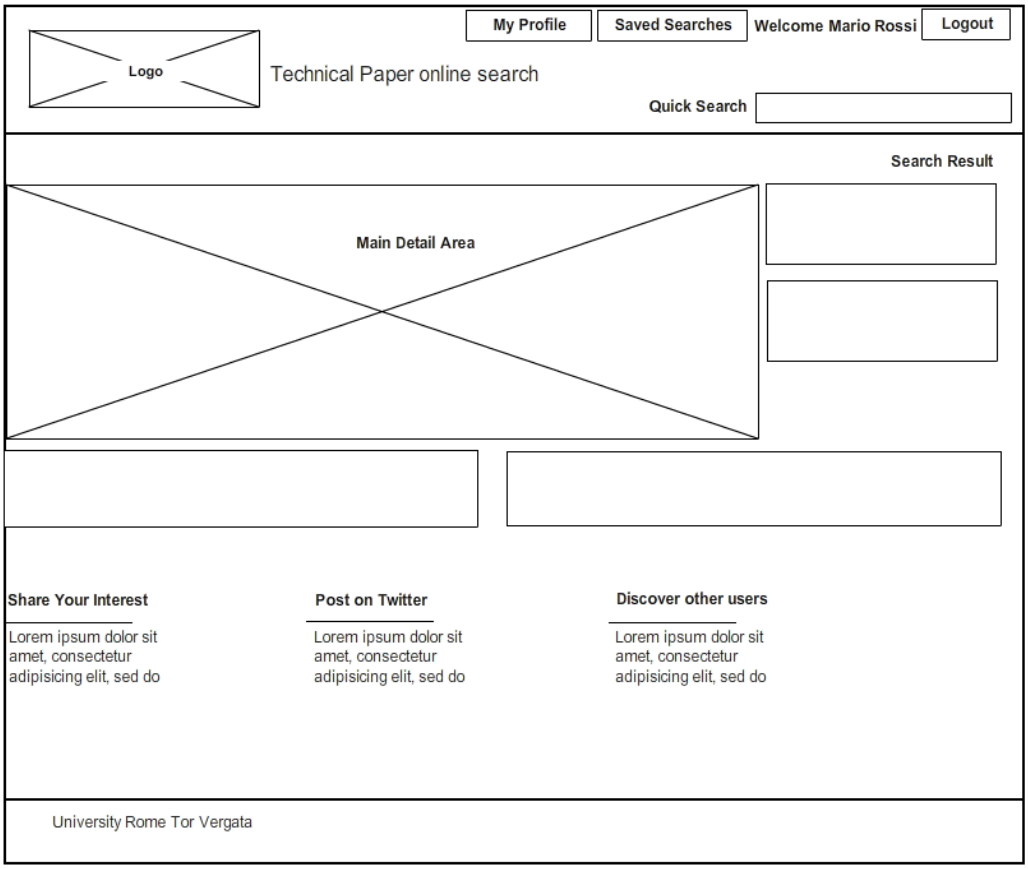



*Figure 15 Front End Dashboard*

# 6 Testing and comparing results

In order to measure effectively the performance we have to clearly divide the set up cost of the overall architecture and the computation cost of the tasks.
In other words, how much is the Hadoop cluster overhead from the effective time required for the operations that need to be performed on it.
As already said, to run a job in Hadoop it is necessary to first setup the cluster and namely *data nodes* and *name nodes*, but and an important function in terms of performance is also performed by the "Task Trackers", which are the nodes of the cluster with the function to assign and run tasks against the single machines involved in the cluster: The *Task Tracker* executes each Mapper/Reducer task as a child process in a separate *Java Virtual Machine*.
When a map and reduce job is started, the framework produces a number of map and reduce tasks, which are consumed by the system with the *capacity* of the overall cluster: for capacity we intend the max number of concurrent tasks running at the same time by the infrastructure.
Like shown in Figure 17 Job Execution Detail), which represents the real time statistic information for a single job, each phase (either Map or Reduce) are split in number of tasks and those are spread around the cluster.
So, the entire execution time of the job can be formally defined as:

$$Job_{ExcetutionTime} = cluster_{startup} + taskTracker_{setup} + MapTasks_{executionTime} + \alpha ReduceTasks_{executionTime}$$

In detail we have a *startup cost* which actually is required to dispatch the *heartbeats* information around the entire cluster, the cost to calculate the number of tasks and setup the execution queue and finally the Map and Reduce execution time.
We introduced a $\alpha$ factor, because the Reduce phase starts when the Map tasks

are still running (usually at 50% of completion), so this factor indicates the remain percentage of reduce tasks after Map completion.
Obviously Map and Reduce phases are the most time consuming; in detail they can be expressed as:

$$MapTasks(Reduce)_{executionTime} = NumberOfTasks * \frac{Avg_{taskExecutionTime}}{Avg_{concurrentRunningTasks}}$$

Therefore, given the same operation (same *average execution time)* what really determines the performance are the number of tasks and how many are running at the same time on the nodes.
To understand this point is necessary to describe how the number of tasks is calculated in Hadoop: it is in fact determined by the *InputFormat,* which is the kind of the input passed to the Map and Reduce. In the example already described of Word Counts it was used an input format, which creates a task for each input split calculated, and then passes this split to map, which count occurrences of terms line by line.
In our case, since the need was to index PDF files, we created a custom *InputFormat* which produces a single *task* for each file: again in the above example we have 1330 tasks which corresponds to the number of files in HDS path.
The number of concurrent tasks running on the entire cluster is calculated by Hadoop itself, based on the available *JAVA HEAP* memory, the computation time of the single task and the CPU load: this value is adjusted by the system during the job execution, but remains in the range of 2-8 per *data node*.
In conclusion, the execution time of a job is really determined by the number of concurrent Map and Reduce concurrently executed on the cluster, and this is determined by the number of *data nodes* available: *to reduce the time we need then to add machines to the node, much more than increase the computation capacity of the single machine*.
Finally it's worth to explore the overhead due to the job submission. In our simulation it takes up to 1 min and 30 seconds: this time is used to perform the

following operations:

- Checking the input and output specifications of the job.
- Computing the *InputSplit* values for the job.
- Copying the job's jar and configuration to the *Map Reduce* system directory on the *FileSystem*.
- Submitting the job to the JobTracker and optionally monitoring its status.

As mentioned the input splits is crucial for the performance of the jobs. Hadoop has its peculiarity to perform very well when analyzing large files rather than many small files, and the reason is basically to search in the peculiarity of HDFS, which default size of a node is 64 MB.

We'll show in the simulation that the real benefit of Hadoop is to analyze a relatively small number of large files (GB files).

**User:** hadoop
**Job Name:** luceneIndexer
**Job File:** hdfs://dev.innovationengineering.eu:54310/tmp/hadoop-hadoop/mapred/staging/hadoop/.staging/job_20130716154
**Submit Host:** dev.innovationengineering.eu
**Submit Host Address:** 188.40.45.14
**Job-ACLs: All users are allowed**
**Job Setup:** Successful
**Status:** Running
**Started at:** Tue Jul 16 16:23:52 CEST 2013
**Running for:** 23mins, 23sec
**Job Cleanup:** Pending

| Kind | % Complete | Num Tasks | Pending | Running | Complete | Killed | Failed/Killed Task Attempts |
|---|---|---|---|---|---|---|---|
| map | 87.59% | 1330 | 160 | 5 | 1165 | 0 | 0 / 0 |
| reduce | 0.00% | 0 | 0 | 0 | 0 | 0 | 0 / 0 |

| | Counter | Map | Reduce | Total |
|---|---|---|---|---|
| Job Counters | SLOTS_MILLIS_MAPS | 0 | 0 | 5,572,127 |
| | Rack-local map tasks | 0 | 0 | 1,170 |
| | Launched map tasks | 0 | 0 | 1,170 |
| File Output Format Counters | Bytes Written | 0 | 0 | 34,086 |
| File Input Format Counters | Bytes Read | 0 | 0 | 2,043,155,138 |
| FileSystemCounters | HDFS_BYTES_READ | 2,043,317,670 | 0 | 2,043,317,670 |
| | FILE_BYTES_WRITTEN | 64,616,727 | 0 | 64,616,727 |
| | HDFS_BYTES_WRITTEN | 34,086 | 0 | 34,086 |
| | Map input records | 0 | 0 | 1,165 |
| | Physical memory (bytes) snapshot | 0 | 0 | 296,088,920,064 |

*Figure 17 Job Execution Detail*

## 6.1 Simulations

We ran a simulation involving a time consuming task: parsing 2 GB of PDF file coming from 1330 patents and adding them to a distributed *Lucene* Index.
In particular we ran this simulation with the system architecture already described with a *name node* and an increasing number of *data nodes*.
The results are showed in the following table, where the last row indicate a scenario without Hadoop:

| Data nodes | CPU Of the Nodes | RAM available | Name Nodes | Number of file | Total Bytes read | Job Submission Cost | Total Job Time |
|---|---|---|---|---|---|---|---|
| 2 | Intel i7 CPU 2.67 GHZ | 4 GB | 1 | 1330 | 2.043 GB | 1 min 5 sec | 24m 35 sec |
| 3 | Intel i7 CPU 2.67 GHZ | 4 GB | 1 | 1330 | 2.043 GB | 1 min 21 sec | 12 min 10 sec |
| 4 | Intel i7 CPU 2.67 GHZ | 4 GB | 1 | 1330 | 2.043 GB | 1 min 40 sec | 8 min 22 sec |
| 1 (No Hadoop) | Intel i7 CPU 2.67 GHZ | 4GB | 0 | 1330 | 2043 GB | 0 | 10 min 11 sec |

As general consideration we can see that the adding data nodes increase the overall capacity of the system producing a major increase in the average number of active tasks and therefore a consistent reduction of the total job time.
The absolute time is, on the other hand, not very satisfactory: that's because we have too many files in the HDFS.
Comparing these results with a single machine node, it turns out that the real benefit of Hadoop is sensible just with an increasing number of *data nodes* and is pretty useless with a small cluster.

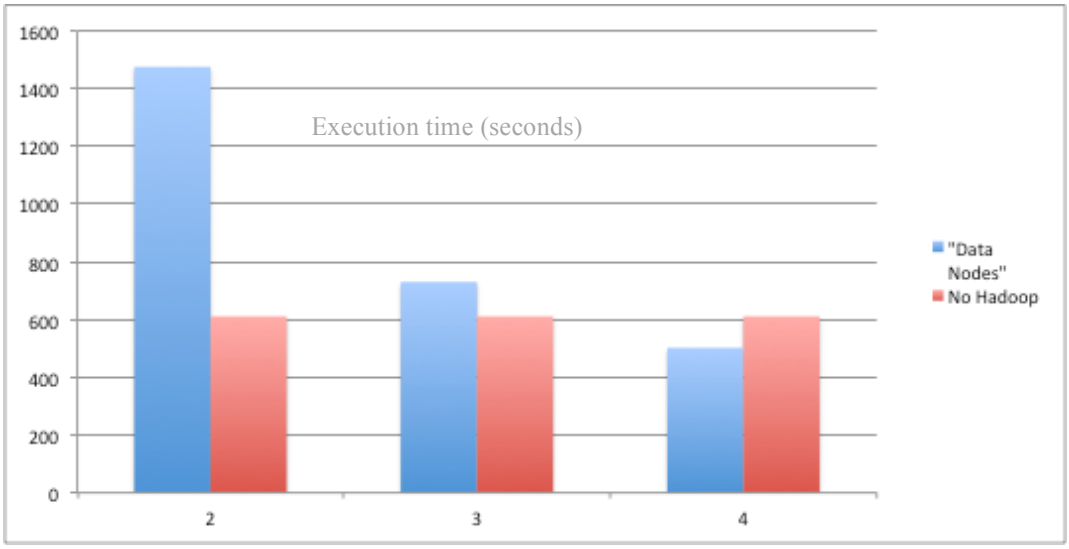



*Figure 18 Performances for Lucene Indexing*

We conducted also another simulation in a familiar scenario for Hadoop: we downloaded the entire Italian Wikipedia article database (around 1 GB approximately) and ran the "WordCount" jobs against it.
We compared the result with a *stand alone* Java program which executes exactly the same task, but without using Map and Reduce (of course possible with a file dimension still acceptable).

The job, theoretically much heavier from a computation perspective, it is executed in a much shorter time (as indicated the table below): the reason is that the split produces a limited number of tasks and the execution queue is shorter.

| Data nodes | Cpu Of the Nodes | RAM available | Name Nodes | Number of file | Total Bytes read | Job Submission Cost |
|---|---|---|---|---|---|---|
| 3 | Intel i7 Cpu 2.67 GHZ | 4 GB | 1 | 1 | 942 MB | 3 min 18 sec |
| 4 | Intel i7 Cpu 2.67 GHZ | 4 GB | 1 | 1 | 942 MB | 2 min 17 sec |
| 1 (No Hadoop) | Intel i7 Cpu 2.67 GHZ | 4 GB | 1 | 1 | 942 MB | 4 min 27 sec |

In conclusion this technology is designer to serve a large cluster of common

servers and expresses its best performance when:

- The quantity of data are expressed by a relatively limited number of big files
- The number of nodes and the capacity of the cluster are designed to avoid the tasks queue, so are chosen coherently with the data splits.

# 7 Alternatives based on real time processing

Scope of our analysis was to investigate not merely the technology declination of Map and Reduce but the architectural and essential innovation produced by this approach.

Our goal was not to follow the last technology trend, which obviously tends to be obsolete after just some periods, but to explore a different paradigm to build application, which can be applied to Big Data scenario.

The question is of course if there's just this alternative or other possibilities can be explored in order to manage the same complexity.

As first answer can be offered by another "hot" topic in the contemporary enterprise architectural debate, which is in a way similar to Map And Reduce: NOSQL Databases.

Relational databases are built on the so-called ACID model, which requires that the database always preserve the *atomicity, consistency, isolation and durability of transactions.*

NoSQL, on the contrary, introduces the BASE model: *basic availability, soft state and eventual consistency.*

Instead of using structured tables to store fields, NoSQL databases use, again, the notion of a key/value store: there is no schema for the database. It simply stores values for each provided key, distributes them across the database and then allows their efficient retrieval.

To give an essential example using the probably most famous one, MongoDb, the operation of inserting an element into the database can be executed using the following commands:

```
db.testData.insert({ name : "mongo" }
```

which inserts a *document* into the schema "testData". *Document* for NoSql database are concept very similar to what already seen for Lucene: in various

sense these technologies are pretty comparable to the architecture described because of the intensive use of indexes to store data and retrieve them efficiently. The structure of a document it's similar to the already introduced JSON serialization, which is a conventional textual representation for an object.
On the other side to get data the application has to perform a *find:*

```
db.testData.find()
```

This operation returns the following results. The *"ObjectId"* values will be unique:

```
{"_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo"
}
```

NoSQL belongs to several types:

- Basic Key/Value which associates basically a binary object to a key represented by an hash
- Document stores like the one already described
- Columnar databases, which are hybrid between NOSql and relational databases because they present some rough row and column structure
- Graph databases, that represents relations using a tree structure

The core of the NoSQL database is the *hash function*: the key, in fact, is used to retrieve the single object from the database, but also the *hash* is used to direct the pair to a particular NoSQL database server, where the record is stored for later retrieval.
NoSQL databases don't support, of course, advanced queries which can be expressed by SQL: that's because they're designed explicitly to rapid, efficient storage of *key->document* pair and to perform a mostly an *reads* operation more than *write once.* In particular it can't be demonstrated that refusing the Acid principle, the system can acquire a major benefit in terms of performance.
In particular, renouncing to atomicity reduces the time tables (sets of data) are

locked; avoiding consistency allows to scale up writes across cluster nodes, and dropping durability gives the possibility to respond to write commands without flushing to disk.
The simplistic architecture of NoSQL databases is a major benefit when it comes to redundancy and scalability. To add redundancy to a database, administrators simply add duplicate nodes and configure replication: scalability therefore is simply a matter of adding additional nodes. When those nodes are added, the NoSQL engine adjusts the hash function to assign records to the new node in a balanced fashion.

To conclude, breaking the ACID principal with is part of the specification for DBMS opens the possibility to scale up more efficiently when the scenario present ad amount of data that really requires it.
This approach can be compared to Map and Reduce even though their similarities are limited, because it offers a scalable approach but it's not based on preprocessing data, with is the basic algorithmic innovation.
On the specific of your use case, which involves the distribution of the potentiality of Lucene search against a large cluster, it's necessary to mention a specific technology based on Lucene, which offers a *Cloud* approach to textual search: Apache SOLR.
Essentially Apache SOLR is a set of *facades* for accessing Lucene index, in particular offering an administration environment for defining the document types called Schema, but most importantly a set of RESTful interface what can be in invoked using simple XML or JSON content representation.
This utilities allows to offer the search functionality as support function to any other system, eventually built on different technologies, and serving its data as a separate component to any systems.
This architecture is becoming very popular because it externalizes some high impact components like Search to and external and auto-consistent system, which establish to the other components a contract accessible via HTTP.
From version 4, moreover, Apache SOLR offers a native function to replicate

the indexes to a Cluster and keep them synchronized: Zookeper.

In conclusion the emerging Big Data scenario are stimulating the adoption of different technologies and algorithm approaches: one of the most important principle is to separate the data analysis from the request processing services.

Our goal was to show how Map and Reduce represents an efficient solution especially when it required analyzing an important amount of raw data applying specific algorithms but also to show that overall cost of the architecture is not trivial, so it really has an effective benefit when the scenario matches a real Big Data scenario and the number of nodes are consequently high.

# 8 Bibliography

1) Cheng-Tao Chu, Sang Kyun Kim , Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, Kunle Olukotun. *Map-Reduce for Machine Learning on Multicore* CS. Department, Stanford University 353 Serra Mall.

2) Dhruba Borthakur. *The Hadoop Distributed File System: Architecture and design*. Document on Hadoop Wiki, 2008.

3) Doug Cutting. *Proposal: index server project*. Email message on Lucene-General email list.

4) Douglas, Laney. *The Importance of 'Big Data': A Definition*. Gartner. Retrieved 21 June 2012.

5) Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. *Bigtable: a distributed storage system for structured data.*

6) Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters.

7) Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler. *The Hadoop Distributed File System.*

8) L. A. Barroso, Jeffrey Dean, and U. H¨olzle. *Web search for a planet: The Google cluster architecture. IEEE Micro,* pages 22–28, March-April 2003.

9) Mike Burrows. *The chubby lock service for loosely coupled distributed systems.* In USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation, pages 24–24, Berkeley, CA, USA, 2006. USENIX Association.

10) Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google file system*. In SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, pages 29–43, New York, NY, USA, 2003. ACM Press.

11) Tom White, *Hadoop the definitive guide.* O'Relly 2012.

12) Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. *Paxos made live: an engineering perspective*.

## 8.1 Web Resources

1) Amazon Dynamo. http://www.allthingsdistributed.com/2007/10/amazons dynamo.html.
2) Apache Hadoop. http://hadoop.apache.org/
3) Apache Lucene. http://lucene.apache.org/java/docs/index.html.
4) Apache Nutch. http://lucene.apache.org/nutch/.
5) Apache Tika. http://tika.apache.org/
6) Apache Zookeeeper. http://www.sourceforge.net/projects/zookeeper/.
7) Bailey. http://www.sourceforge.net/projects/bailey.
8) http://techcrunch.com/2012/08/22/how-big-is-facebooks-data-2-5-billion-pieces-of-content-and-500-terabytes-ingested-every-day/
9) http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf
10) Katta (An open source Hadoop implementation). http://www.sourceforge.net/projects/katta/.
11) Werner Vogel. *Eventually Consistent*. http://www.allthingsdistributed.com/2007/12/eventually consistent.html.