UNIVERSITÀ DELLA CALABRIA

Department of Computer Science, Modeling, Electronics and Systems Engineering
(DIMES)

PHD THESIS

Methodologies and Applications for Big Data Analytics

Supervisors:
Prof. Sergio FLESCA
Prof. Elio MASCIARI

Author:
Nunziato CASSAVIA

Coordinator:
Prof. Felice CRUPI

XXXII Cycle of PhD in ICT

SSD: ING-INF/05

UNIVERSITÀ DELLA CALABRIA - DIMES

Abstract

PhD in ICT

Methodologies and Applications for Big Data Analytics

by Nunziato CASSAVIA

Due to the emerging Big Data paradigm, driven by the increase availability of users generated data, traditional data management techniques are inadequate in many real life scenarios. The availability of huge amounts of data pertaining to user social interactions calls for advanced analysis strategies in order to extract meaningful information. Furthermore, heterogeneity and high speed of user generated data require suitable data storage and management and a huge amount of computing power. This dissertation presents a Big Data framework able to enhances user quest for information by exploiting previous knowledge about their social environment. Moreover an introduction to Big Data and NoSQL systems is provided and two basic architecture for Big Data analysis are presented. The framework that enhances user quest, leverages the extent of influence that the users are potentially subject to and the influence they may exert on other users. User influence spread, across the network, is dynamically computed as well to improve user search strategy by providing specific suggestions, represented as tailored faceted features. The approach is tested in an important application scenario such as tourist recommendation where several experiment have been performed to assess system scalability and data read/write efficiency. The study of this system and of advanced analysis on Big Data has shown the need for a huge computing power. To this end an high performance computing system named Coremuniti^{TM} is presented. This system represents a P2P solution for solving complex works by using the idling computational resources of users connected to this network. Users help each other by asking the network computational resources when they face high computing demanding tasks. Differently from many proposals available for volunteer computing, users providing their resources are rewarded with tangible credits. This approach is tested in an interesting scenario as 3D rendering where its efficiency has been compared with "traditional" commercial solutions like cloud platforms and render farms showing shorter task completion times at low cost.

Acknowledgements

There are a number of people who have supported me along this journey in obtaining my Phd. First of all I would like to thank my thesis supervisors, Dr. Elio Masciari and Dr. Sergio Flesca for starting me in the world of research and academia and for their patience and desire to teach me the language, methods and skills of this world. I still have much to learn.

I would also like to express my gratitude to Dr. Andrea Calì for his support, useful advice and encouragements during my study abroad period at the Birkbeck University in London and all my colleagues of the labs.

I also thank all my friends starting from Pietrangelo, Mimma, Vincenzo, Ivana, Roberto, Anna to Siro, Michele and passing to all "San Giacomo" crew that I could not insert everyone because of the little space available as well as all my friends who for one reason or another are far from this beautiful and complicated land called Calabria but that, despite everything, we always managed to stay in touch.

A big tank go to all my Family, to my uncles, my in-laws and my cousins Nicola e Michele.

A special tanks go to my parent, Domenico e Rosetta for their constant support and because they always continue to do much more than their duty as parents.

And finally, most important, I would like to thank my wife Francesca for their understanding and patience and for supporting me beyond all expectations, with warmth and trust.

Contents

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**

2 Dealing with Big Data **7**
2.1 Background 7
2.2 The NoSQL Paradigm 8
2.3 The Relational Databases 11
2.4 The CAP Theorem 12
2.5 NoSQL Consistency 14
2.5.1 Consistency limitations 14
2.6 NoSQL System Classification 15
2.6.1 Key-Value Datastore 18
2.6.2 Document Datastore 19
2.6.3 Graph Datastore 20
2.6.4 Column-Family Datastore 22
2.7 Open Source Tools for Big Data analysis 24
2.7.1 Hadoop Distributed File System 24
2.7.2 Apache HBase 25
2.7.3 Map Reduce 27
Google Map Reduce 30
Apache Hadoop 31
2.7.4 Apache Hive 33
2.7.5 Apache Spark 35

3 Analytics on Big Data **39**
3.1 Background 39

3.1.1 The Lambda Architecture 39
3.1.2 The Kappa Architecture 40
3.1.3 Architectures Implementation 42
3.2 Search Based Analysis Architecture 43
3.2.1 Complex Search 43
3.2.2 The Sigma Architecture 46
3.2.3 A Sigma Implementation for Search Based Analysis . . . 48
3.2.4 System Description and Integration 51
Data Indexing 53
3.2.5 Open Source Tools For The System 55
Apache Solr 55
SolrCloud 58
Carrot2 60
Hue 61
3.3 OLAP Based Analysis Architecture 62
3.3.1 The Pentaho Data Warehouse Analysis Tools 62
Pentaho Data Integration 63
Pentaho Business Analytics Platform 64
Schema Workbench 65
Display Tools 66
Pentaho Analysis Operation 68
3.3.2 Big Data OLAP System Architecture 69

4 High Performance Computing On Peer to Peer Network 73
4.1 Background 73
4.1.1 Coremuniti in a nutshell 76
4.2 Coremuniti Architecture 78
4.2.1 Communication Protocols 82
4.3 Subtasks Assignment and Credit Rewarding 88
4.4 Case study: Efficient and Effective 3D rendering 93
4.5 Experimental Evaluation 98
4.5.1 System Performances 99
Time Performances 100
Cost Evaluation 102
4.5.2 User Revenues Evaluation 103

5 Enhancing Big Data Information Search **107**
5.1 Introduction 107
5.1.1 The proposed framework in a nutshell. 111
5.1.2 Improving Faceted Navigation by Clustering 112
5.2 Data Exchange and Count Constraints 115
5.3 Modeling discovered dimensions diffusion 120
5.3.1 The Algorithm for Influence Evaluation 121
5.4 System Architecture 126
5.5 Experimental results 128
5.5.1 Data Gathering. 129
5.5.2 Experimental Setup. 131
5.5.3 System Scalability Evaluation 131
Map Reduce Efficiency 131
Read and Write Efficiency 132

6 Conclusions **135**

Bibliography **139**

List of Figures

2.1 Impedance Mismatch 10
2.2 CAP Theorem 13
2.3 Key Value Content 19
2.4 Graph Example 21
2.5 Column Family Model 23
2.6 Super Column Family Model 24
2.7 HDFS Write 25
2.8 HBase Architecture 27
2.9 HBase Storage Methodologies 28
2.10 Google Map Reduce Overview 30
2.11 Hadoop Map Reduce Working Scheme 32
2.12 Hive Architecture 34
2.13 Spark Stack 36
2.14 Spark Architecture 37

3.1 Lambda Architecture 41
3.2 Kappa Architecture 42
3.3 Learning By Results 43
3.4 Amazon search 44
3.5 Faceted Navigation Example 45
3.6 Sigma Architecture 46
3.7 Big Data Search Architecture 49
3.8 Hue-Based Search Interface 53
3.9 Solr Indexes Generation 54
3.10 Inverted Index Example 56
3.11 SolrCloud Architecture 59
3.12 Carrot2 Web Interface 60
3.13 Hue Interface 62

3.14 Pentaho Data Integration Interface 64
3.15 Pentaho BA Server Interface 65
3.16 Schema Workbench Interface 66
3.17 JPivot Interface 67
3.18 Saiku Interface 68
3.19 Pentaho System Interactions 69
3.20 OLAP Analysis Architecture 70
3.21 System User Interactions 71

4.1 Coremuniti System at Work 77
4.2 System Architecture 79
4.3 Adding New Project Protocol 84
4.4 Task Assignment and Execution Protocol 86
4.5 Coremuniti Server At Work 96
4.6 Mozaiko Rendering View 97
4.7 Execution Times Comparison - 3D Rendering 101
4.8 Execution Times Comparison - Matrix multiplication 102
4.9 Rendering Cost Comparison 103
4.10 Matrix Multiplication Cost Comparison 104

5.1 Influence Model for User Search 110
5.2 Overview of Framework 111
5.3 Diffusion Graph and Dimensions Assignment 125
5.4 Diffusion process of Dimensions 126
5.5 System Architecture for Big Data Search 127
5.6 Execution Time vs. Data Shard Size 132
5.7 Execution Time vs. Arrival Rate 133

List of Tables

2.1 Use Cases Of NoSQL Datastores. 16

4.1 NodeServerAgent Beat Message Content 86
4.2 Computation Power of Resource Providers in Testnet Network . 99
4.3 Gain Comparison 105

5.1 Response Time vs. Number of Keywords 134

List of Abbreviations

P2P	**P**eer **To** **P**eer
DAG	**D**irected **A**cyclic **G**raph
MEA	**M**ain **E**xecution **A**gent
EA	**E**xecution **A**gent
PA	**P**eer **A**gent
TEA	**T**ask **E**valuation **A**gent
HDFS	**H**adoop **D**istributed **F**ile **S**ystem
DML	**D**ata **M**anipulation **L**anguage
RDD	**R**esilient **D**istributed **D**ataset
PDI	**P**entaho **D**ata **I**ntegration
OLAP	**O**n-**L**ine **A**nalytical **P**rocessing
MDX	**M**ulti**d**imensional **E**xpressions
SW	**S**chema **W**orkbench

Chapter 1

Introduction

The impressive progress and development of Internet and on-line technologies has led to an increasing availability of a huge volume of data generated by heterogeneous sources at high production rates (Economist, 2010). These massive data, referred as *Big Data* (Nature, 2008), exhibit a great variety and may be exploited to gather information about people, things, services and their interactions. In this respect, a great deal of attention has been devoted to the design of novel algorithms for analyzing information available from Twitter, Google, Facebook, and Wikipedia, to cite a few of the main big data producers. Furthermore, almost every individual leaves digital traces when connected to the plethora of available Sensor Networks, Cloud Storages, and Global Positioning services, through smart phones and tablets. By using their devices, people get in touch (e.g., with friends, followers and fans) and perform several actions (e.g., they post comments, videos, photos, they link resources, they rate products and they express their sentiment about something or someone). The availability of such unprecedented large amount of heterogeneous information sources is quite challenging as they provide the ground for the analysis of human behavior, and their evolution when influenced by other people's opinion/suggestion. Predicting user behaviors is crucial in many application scenarios such as viral marketing, sentiment analysis and epidemic modeling (Kempe, Kleinberg, and Tardos, 2003; Tang, Xiao, and Shi, 2014; Newman, 2010). As a result there is an increasing deal of interest, shared by both researchers (Agrawal et al., 2012) and industries (Manyika et al., 2011), for a deep understanding of user attitudes, user preferences and user interaction patterns with complex systems, which is crucial for implementing effective tools that are the basis of high quality services able to satisfy user requests.

The definition of *Big Data* usually use the 3v model:

- *Volume*: refers to the amount of data (structured, unstructured) generated every second
- *Variety*: refers to the different types of data that are generated and collected
- *Velocity*: refers to the speed at which new data is generated

Big Data management require new archiving and analysis approaches. More in detail, the traditional relational model is replaced with the so-called NoSQL systems. While the relational databases aim at a vertical scalability, which consists practically in enhancing the machine on which the DBMS is hosted, the NoSQL datastores aim at a horizontal scalability, which consists in the possibility of dynamically adding new resources.

The heterogeneity and velocity these data are generated require appropriate ad-hoc designed data storage. A major problem in the management of this enormous amount of data is the extraction of "new" information that has a consistent value with respect to the content of the entire database and its consultation in near real-time timing. This new information is crucial and should be stored and used within the whole analysis process.

The issue of devising novel solutions for analyzing big data is becoming more and more compelling in the construction of *Intelligent Information Systems* (IIS) to assist end user in the search of relevant information and in the interaction with services in the net.

A first challenge in the design of a such system is to *extract and enhance the information content of source data*. An example could be related to tourist context. Assuming to have only customer reviews of a large number of restaurants we may want to classify restaurants according to the various “dimensions” (i.e., categorized properties or other features) that have been singled out and evaluated in the reviews. A solution will be presented based on a modified version of data exchange.

However, the generation of new information and therefore the creation of new dimensions analysis can lead to the generation of a too large number of dimensions. So, a second challenge consists in *detecting on-the-fly features (i.e. dimensions) that are relevant in the search context and tailored to the user behavior.* A

possible solution is based on the influence that users of a system exert between them. As example, consider the case of a user expressing an enthusiastic comment on a recently visited city (e.g., by a Facebook “like" and/or post). Most likely, his/her friends could be interested to utilize the positive feedback on that city when deciding their next travel destination. In this respect, the system should suggest some categorized high level information (e.g. "touristic city" for family or "funny city" for young people) in response to a search for a specific destination based on the influence exerted on him by other system users.

Some issue arises from this challenges as *How to allow relevant information about user search preferences propagate over the network?* or *How to measure the possible information spread*? This issues are preliminary to the activity of discovering new dimensions to be added to the information sources and to be later used to support a search mixed with tailored browsing. Moreover, data gathered by social network and search engines are inherently non structured; therefore, a data exchange task has to be performed for moving source data into a target “structured” database enabling an effective analysis of user behavior.

This work presents a framework tailored for analyzing user interactions with IIS while seeking for some domain specific information (e.g. choosing a good restaurant in a visited area). The framework enhance user quest for information by exploiting previous knowledge about their social environment, the extent of influence the users are potentially subject to and the influence they may exert on other users. User influence spread, across the network, is dynamically computed as well to improve user search strategy by providing specific suggestions, represented as tailored faceted features. Such features are the result of data exchange activity that enriches information sources with additional background information and knowledge derived from experiences and behavioral properties of domain experts and users. The approach is tested in the tourist recommendation scenario using data gathered from Yelp, Twitter and Facebook. Several experiment have been performed on this dataset to assess system scalability and data read/write efficiency.

Analysis on Big Data require a lot of computational power. In recent years there has been the development of projects whose complexity becomes very challenging and requires enormous computing power.

In order to solve similar of problems, a new computing paradigm has born: *crowdsourcing*. This new paradigm is based on the idea of gathering the resources needed to complete a task from the crowd in order to parallelize its execution. The majority of systems that exploit this paradigm is based on peer to peer network (P2P). A well known example of project involving resources from crowd is BOINC (Berkeley Open Infrastructure for Network Computing) [1]. It allow users to contribute voluntarily to a variety of project. In the last few years, another approach has been developed related to cryptocurrency mining. Users of a P2P network of this type provide disk and calculation resources to maintains the system and earn a tangible reward. Some famous example of this type of system are Bitcoin (Nakamoto, 2008) or Ethereum (Wood, 2014).

In this work a system named *Coremuniti*^{TM} is presented, which is inspired by the collaborative model used in BOINC while implementing an ad hoc rewarding strategy similar to cryptocurrency mining. This system allows users to share their CPU and memory in a secure and efficient way. By doing this, users help each other by asking the network computational resources when they face high computing demanding tasks. Differently from others proposals available for volunteer computing, users providing their resources are rewarded with tangible credits, i.e., they can redeem their credits by asking for computational power to solve their own task and/or by exchanging them for money.

The design of a peer to peer network presents a large number of issues such as peer discovery, task partitioning and assignment, security of communications and data. Alongside these typical issues of peer to peer systems, other problems need to be addressed like those related to the quality of service that must be comparable to those offered by centralized server farms and the reuse of peer resources that are not completely used. So, main challenges addressed in the design of this framework are:

- *provide a P2P service better or comparable w.r.t centralized server farm*
- *design a subtask assignment that from one side minimize the expected completion time for overall task and from another take into account the user revenue expectation*
- *design a network with a better revenue system than cryptocurrency mining*

[1] https://boinc.berkeley.edu

The designed framework can be used in several application scenarios, e.g. computer simulation and advanced Big Data analysis and it is well suited for vertical implementation of computing intensive tasks, representing a trans-disciplinary opportunity. The first application developed is aimed at 3d rendering industry that turns to be a severe test bench for this technology. The rendering of photorealistic images takes, on average, several hours of computing on typical user devices. Moreover, the calculation time becomes enormous in case of movies rendering where for every second they usually need 20 frames. To this end, a specialized plugin named *Mozaiko*^{TM} has been developed which allows to render Blender 3D models in the distributed P2P network.

In order to validate this framework a comparison was conducted against some of the most popular specialized rendering server farms (i.e. RenderStreet and RebusFarm) and some clusters on well known cloud service platform (i.e. Microsoft Azure, Amazon EC2, Google Cloud, IBM Cloud and DigitalOcean). The experimental analysis shows that existing solutions are slower and more expensive than the proposed solution. Moreover, the designed system does not require to frequently purchase new hardware, since users continuously provide (up to date) computational power. Finally, the better re-use of already powered resources could induce a beneficial systemic effect by reducing the overall energy consumption for complex tasks execution.

This document is structured as follows. Chapter 2 introduces Big Data, that is the main topic of our research and the NoSQL Paradigm. So, the main difference between NoSQL and relational databases are shown. Moreover a NoSQL classification based on data model is provided as well as a description of main open source tools to deal with big data. Chapter 3 focuses on Big Data architectures and two basic models are presented. The first architecture, called Sigma, describes a solution for the construction of a complete, interactive and scalable Big Data system while the second is based on an OLAP type approach. Chapter 4 describe the architecture and the main components of the *Coremuniti*^{TM} P2P network. Finally, Chapter 5 describe a Big Data solutions for extract and enhance the information content of source data and for detect features (i.e. dimensions) that are relevant in the search context and tailored to the user behavior.

Chapter 2

Dealing with Big Data

2.1 Background

Due to the continuous improvement in data generation techniques and the widespread use of pervasive tools like sensor networks, a huge volume of heterogeneous, stream based and complex data are generated every day. This scenario leads to various issues that affect entire data life cycle, from acquisition phase to analysis phase. In particular, since data are collected at high speed and from different sources, it is necessary to make some choices (typically ad-hoc) about which data to keep, which to discard and which metadata to use to store them reliably. Many issues also arise in data pre-processing phase. In particular, the following aspects must be addressed:

- *Structure.* Data is often generated in an unstructured format (for example, in sensor networks data can be generated by heterogeneous sensors, perhaps sold by different suppliers and operating with different protocols);
- *Semantics.* Data can refer to different concepts (for example, in sensor networks, data can refer to different physical properties that are observed for various purposes);
- *Integration.* the value of data grows considerably when sources can be linked to other sources. In data value chain, integration is a very important task.

However, main challenge remains to organize and model data in order to allow analysis of all possible information of interest in useful time. It must be

said that in analysis phase the choice of algorithms to be used also represents a crucial point, this in order to efficiently analyze data contained on these highly distributed system. Finally we have knowledge extraction, presentation of results and their interpretation, all operations that should be treated by domain experts (even non-technical). The classical relational model show several limitation when used to support storage and retrieving in Big Data sources ; to overcome this limitations a new paradigm, called *NoSQL* has been proposed in recent years. The *NoSQL* (that stands for Not Only SQL) systems provide a mechanism for data storage and retrieval that is modeled differently than tabular relations used in relational databases. *NoSQL* databases use multiple data models to access and manage data and are specifically optimized for applications that require large volumes of data, low latency and flexible data models, obtained by streamlining some of data consistency criteria of other databases. The definition of NoSQL was used for the first time in 1998 by Carlo Strozzi (*C. Strozzi. Nosql relational database management system.*) for an open source relational database that did not use an SQL interface. The term was reintroduced in 2009 by Rackspace employee Eric Evans when Johan Oskarsson of Last.fm organized an event to discuss distributed open source databases. The name was an attempt to describe the development of a large number of non-relational distributed data storage systems that often do not attempt to provide classic ACID guarantees.

In the next paragraphs will be provided a detailed characterization of the NoSQL paradigm and a description of a series of open source tools to operate on these types of databases.

2.2 The NoSQL Paradigm

The NoSQL movement was born from the need to manage an enormous amount of data, which drove large companies like Google and Amazon to no longer rely on relational systems for this purpose, but to build large hardware platforms on inexpensive server clusters (the so-called "commodity servers"). Relational systems are designed to run on very powerful single machines and although there have been many attempts to obtain from them a scalable solution, many difficulties remaining; some examples may be the operation management of joins on distributed tables and the high costs of software licenses,

generally provided for individual servers. While relational databases aim at vertical scalability, which basically consists in upgrading the machine on which DBMS runs, NoSQL datastores aim at horizontal scalability, which consists in the possibility of dynamically adding new machines to system. Contrary to what could be deduced from NoSQL term, this type of system does not prohibit the use of SQL language, but aims to propose an alternative solution to data management and database modeling in those cases where large amounts of distributed and unstructured data are involved. NoSQL is the acronym of *"Not only SQL"*, probably the expression was taken as a reference only because SQL is most common query language used in relational databases and therefore represents a symbol of whole relational paradigm.

Although supporters of new movement claim that these systems are more performing, more scalable and easier to use, they do not claim to replace relational model, because there are still many situations in which relational databases are only the valid solution.

In addition to the desire of medium and large companies to be able to capture as much data as possible and be able to process them quickly, another reason that favored the birth of NoSQL systems was the demand for greater productivity in application development. Much of effort in this process is caused by mapping between data structures that are in main memory and relational databases, a problem known as *"Impedance Mismatch"*. In relational model data is organized in tables and rows, where a tuple is a set of name value pairs and a relation is a set of tuples. All operations in SQL operate on relations and return relations, similar to what happens in relational algebra. However, if on one hand these tools provide elegance and simplicity, they also introduce many limitations, including the impossibility of inserting complex structures in tables, such as lists or nested records. However, these limitations are not only in main memory, but are necessary when working with object-oriented programming languages, such as Java. To solve this problem in 1990 the first object-oriented databases were developed, but they had no success. The result was that, if you wanted to exploit certain structures in main memory, a phase of data translation was mandatory before storing them on disk, as you can observe in figure 2.1.

Although relational databases have always been seen as a universal solution, NoSQL datastores in general provide a data model that better fit the

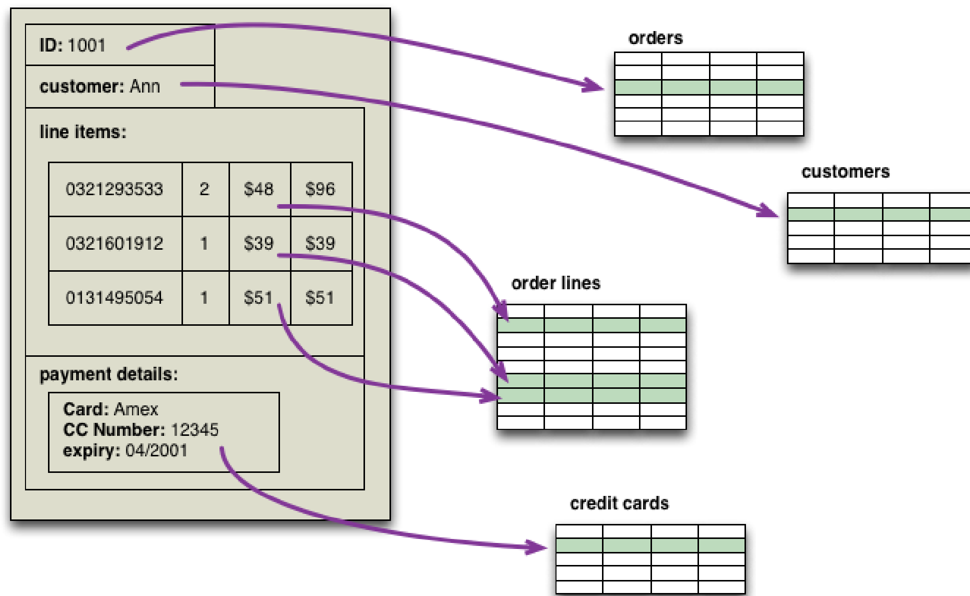


FIGURE 2.1: Translation of aggregated data toward a relational representation

needs of applications and also require less code to write, debug and update. Relational databases offer a good compatibility with different software implementation and an high level of abstraction thanks to tabular data model and query language. On the contrary, NoSQL systems, of which there are currently more than 200 implementations[1], present greater differences on implementations starting from interrogation methods. The main features they have in common are:

- *No explicit schema*: Lack of a schema specified through a formal language, on the contrary, an implicit schema is used, that is contained in the code that manipulates data. This is useful for dynamically add new attributes to data records and easily manage non-uniform data, avoiding waste in storing sparse tables.
- *Horizontal scalability*: The ability to work well in clusters through data replication and segmentation

[1]For a complete list you can refer to the site http://nosql-database.org/

- *Ability to manage huge amounts of data*: Automatically balance workload between various network nodes
- *Different Transaction Model*: Different from relational transaction model, the idea is that removing ACID constraints can improve performance and scalability.
- *Open Source Philosophy*: Crucial to contribute to increasing potential of used technologies

2.3 The Relational Databases

For a long time the only question that software developers asked themselves when launching a new project was *which relational database* to use, as long as their company didn't already have preferences for any particular database. So it is also natural to ask what are reasons why relational databases have survived so long over time. Certainly one of main problems with business applications is management of concurrency, that is, how to manage cases in which multiple users simultaneously access the same piece of data. Concurrency is a notoriously difficult problem to deal with and relational databases allows to manage control of all data accesses in a fairly simple manner using transactions. The latter also play a fundamental role in error management, in fact if an error occurs during execution of an operation, it is possible to do roll back operation and restore initial state of things, all in a transparent way to developer. Almost all RDBMS support ACID transactions:

- *Atomicity*: The transaction is indivisible: all parts that compose it are applied to a database or none.
- *Consistency*: The database must remain in a consistent state before and after execution of a transaction, so there must be no contradictions between data stored in DB.
- *Isolation*: When multiple transactions are executed by one or more users simultaneously, a transaction does not see effects of other concurrent transactions.

- *Durability*: When a transaction has been completed (with a commit), its changes become persistent, meaning they are no longer lost.

One of biggest advantages of relational databases was ability to make applications coexist in a single and rich ecosystem, even if different from each other. Thanks to database sharing mechanisms, multiple applications can store their data in a single database, which means that each application can access data of other applications with necessary permissions while database system manage simultaneous accesses. However, some applications must work on billions and trillions of data every day and consistency is not the only desirable property in these cases, scalability is the concept that plays the most important role. The only way to deal with huge amounts of data is to move towards a distributed system and in 2001 the same large producers of relational databases released on the market distributed solutions like Oracle RAC (Real Application Clusters). These solutions proved to be difficult to configure and highly unnatural; in fact, the integrity constraints or the control of consistency on data segments (shards) placed on different nodes were lost. In this direction, important limitations are highlighted by CAP Theorem.

2.4 The CAP Theorem

The CAP theorem was theorized by Eric Brewer (Brewer, 2000), first born as a mathematical conjecture and only two years later became theorem as a formal proof was presented. The three letters that make up his name stand for: C = Consistency, A = Availability, P = Partition-Tolerance and these principles have the following definition:

- ***Consistency (strong)***: If a data is written in a node and is read by another node in a distributed system, the system will return last written value (the consistent one);
- ***Availability***: The distributed system must always satisfy client requests, so eventual failure of a node must not block the whole system.
- ***Partition-Tolerance***: The ability of a distributed system to be tolerant to partitioning, the addition/removal of a node must not block system execution.

This theorem establishes that only two of the three properties just described can be satisfied simultaneously.

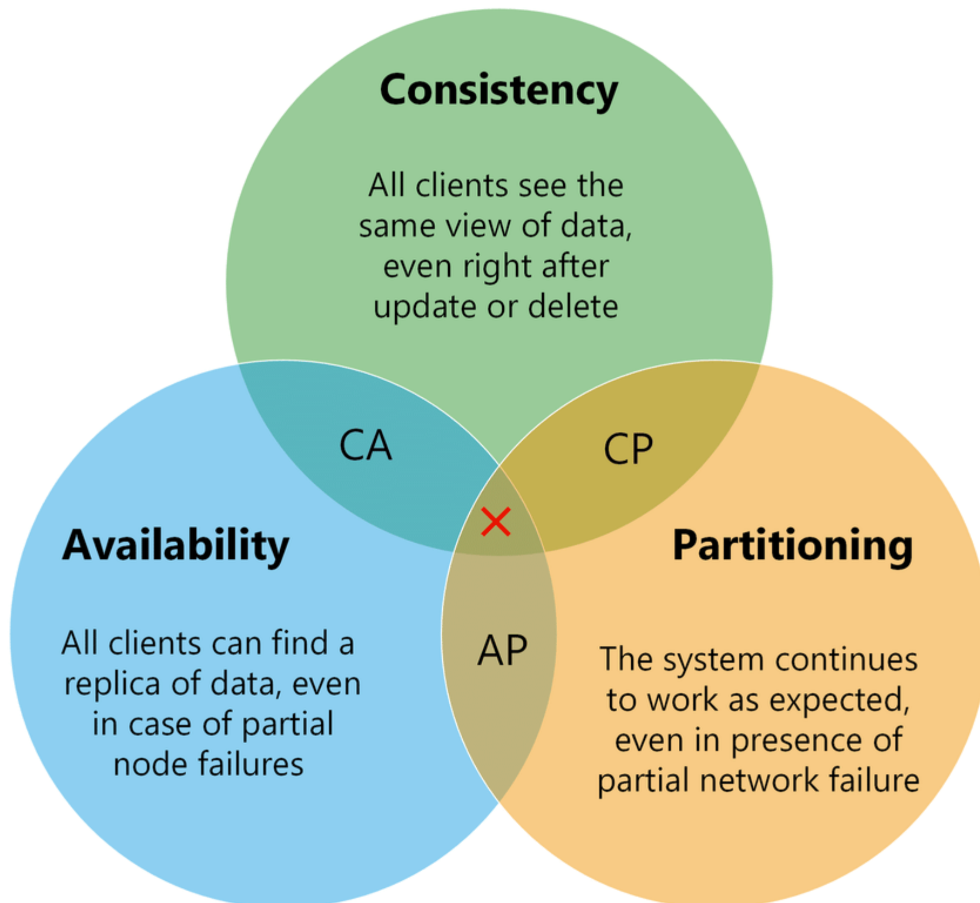


FIGURE 2.2: Set theory representation of CAP theorem

It is possible to classify distributed databases in three categories: CA, CP and AP, depending on whether system renounces Availability, Consistency or Partition Tolerance.

- *CA*: Traditional RDBMS are contained in this category, thus guaranteeing consistency and availability but not tolerance to message loss. Algorithms like two-phase commit allow you to manage distributed transactions but literally block system when a network partition is needed.

- *CP*: It is possible to obtain more scalable solutions using replication on multiple nodes or through segmentation techniques but with the risk of losing access to some data when some node fails.
- *AP*: In this solution it is assumed that system can return inaccurate data in exchange for highest levels of availability and tolerance to partitioning

It would therefore seem that alternatives to relational databases are limited to last two classes only, but in reality NoSQL systems offer hybrid solutions, giving up strong consistency in exchange of more efficient and valid solutions.

2.5 NoSQL Consistency

2.5.1 Consistency limitations

The CAP theorem demonstrates that there are severe limitations on distributed systems. While relational databases guarantee strong consistency, many NoSQL datastores sacrifice it for the so-called *"eventual consistency"* or *"final consistency"*. The writing operation on a data are not executed immediately on all replicas of system, but over time and in absence of further updates the replicas become substantial in all system. This solution is tolerable in many cases as DNS Services or Web sites. Problems may arise, however, when users access multiple replicas, because there may be inconsistencies.

Most common approach to detect inconsistencies used by NoSQL peer-to-peer systems is a special versioning system, called *vector stamp*. This uses a vector of counters, each of which is associated with a network node, for example for three nodes (red, green, blue) we could have the vector [red: 1, green: 5, blue: 2]. Whenever a node modifies one of its resources it increments its counter in vector and every time two nodes communicate they exchange this vector. To detect conflicts, it is necessary to simple compare them. For example, the vector [red: 2, green: 5, blue: 2] is bigger than vector [red: 1, green: 5, blue: 2] because at least one of counters has a higher value than those of second vector, while the vector [red: 2, green: 5, blue: 2] and vector [red: 1, green: 6, blue: 2] both have a bigger value of other, which means that a conflict has occurred. Sometimes in addition to consistency, another property of ACID transactions is also sacrificed: the isolation, so a small probability that transactions will interfere each other is tolerated.

With advent of NoSQL datastores an optimistic alternative to ACID solution called *BASE* has been defined and stands for:

- ***Basic Availability.*** The system offer availability, but not necessarily for all nodes at any time, partial failures are accepted, provided that entire system does not stop working.
- ***Soft-state.*** Data held in client's cache expires if they are not updated by server requests.
- ***Eventual Consistency.*** It is not required that database is always in a consistent state, but that it becomes over time.

Developing BASE applications that support outdated data, give uncertain answers and are tolerant to failures, it is much more difficult than managing system with ACID transactions, but CAP theorem states that there is no other choice if you want a scalable application. Another important difference between ACID and BASE lies in the way data is processed and stored, while in first solution, durability/persistence property ensures that data is always modified on disk, in second it works mainly in main memory and only periodically perform a flush operation to secondary memory. In this way, if on the one hand response times are significantly reduced, the data would be lost if system were to crash. To limit this effect, data replication mechanisms are exploited.

2.6 NoSQL System Classification

Usually NoSQL datastores are grouped into 10 categories:

- Key-Value datastore
- Document datastore
- Column family datastore
- Graph datastore
- Multimodel Databases
- Object Databases

TABLE 2.1: Use Cases Of NoSQL Datastores.

DB Name	Data Model	Used By	CAP
Big Table	Column Family	Google	CP
HBase	Column Family	Twitter	CP
Cassandra	Key-Value / Column Family	Facebook	AP
MemcachedDB	Key-Value	Wikipedia	CP
SimpleDB	Key-Value	Amazon	AP
DynamoDB	Key-Value	Amazon	AP
Voldemort	Key-Value	Linkedin	AP
CouchDB	Document Store	Ubuntu One	AP

- Grid & Cloud Database Solutions
- XML Databases
- Multidimensional Databases
- Multivalue Database

However, this complete classification is not considered by most of scientific documents that address this topic; usually it is preferred to use the so-called "data model" as only element of distinction. The latter is the model through which we perceive and manipulate data, it differs from storage model, which instead describes how database stores and manipulates data internally. In this way we can distinguish four main classes, which will be analyzed later individually:

- Key-Value datastore
- Document datastore
- Column family datastore
- Graph datastore

Table 2.1 shows some examples of datastores used by big companies. It is important to say that boundaries of this classification are not well defined, so some systems fall into several categories.

The first three categories share a common feature among their data models: the orientation to aggregates, that is a different approach compared to relational model. While in relational model information is maintained in sets of tuples belonging to different tables, the term "aggregate" indicates a more complex structure, which allows use of lists and records inserted inside it. In practice it corresponds to a unit for data manipulation and consistency management and is naturally combined with mechanisms required in a distributed system, such as data replication and segmentation.

The one shown here is an example of aggregate relating to a records of an order for an e-commerce site.

```
{"customer": {
"id": 1,
"name": "Nunzio",
"billingAddress": [{"city": "Cosenza"}],
"orders": [{
"id":99,
"customerId":1,
"orderItems":[
{
"productId":25,
"price": 39.23,
"productName": "Book"
}],
"shippingAddress":[{"city":"Cosenza"}]
      "orderPayment":[{
"ccinfo":"2300-4500-1560-10670",
"txnId":"bbkfuiuer879rft",
"billingAddress": {"city": "Cosenza"}
}],
 }]}}
```

The concept of aggregate is not used from all NoSQL datastores, Graph datastores and relational databases can be defined *"aggregate-ignorant"*. A very important property of aggregates is related to management of concurrency. While in relational databases ACID transactions must refer several rows of different tables and must guarantee atomicity of operations on this set of data, on aggregates, atomicity is generally guaranteed only on single aggregate. If it

is necessary to guarantee consistency on different aggregates, the management is delegated to programmers.

2.6.1 Key-Value Datastore

Key-value datastores offer a form of *opaque aggregate*. The data to store is an object identified by a key and formed by an array of bits, with possible limitations on maximum size. This flattening makes very high performances (typically in reading) and at the same time facilitates work of distributing load on multiple machines (sharding) realizing an almost linear scalability. The only search operation offered by DB is by key and it is not possible to construct indexes on aggregate object, the only solution is to get entire object and then manipulated as desired at application level. Of course this is not always true for all key-value datastores, there are some, such as Redis[2], which offer more complex structures for storing and accessing data, such as lists or sets and which allow to perform also particular queries such as unions, intersections or range queries. In general, these datastores can be seen as common hash tables and represent simplest category to use, among NoSQL systems, from API point of view. The classic operations made available are: get, put and delete of a value. In this type of datastore, the design of primary key is of fundamental importance, due to fact that storage is based on direct addressing (hash keys) and efficient execution of searches and sorting is not permitted. Furthermore, creation of secondary indexes (via B + Tree or their variants) causes a reduction in performance (especially in writing) and complicates management of sharding and clustering, as well as consistency. The best way to improve performance is to know the keys to be used in advance, preferring natural keys (as show in Figure 2.3), which avoids dependence to a central authority to generate them.

Many Key-Value datastores existed before the birth of NoSQL movement and were known as *"application caches"* or "application accelerators", in practice they had (and some still have) a single purpose: to store most frequently accessed data in primary memory. Some implementations do not save, by choice, data in secondary memory (e.g. Memcached[3]) and they would not survive a system crash, but they are still ideal solution for storing web sessions or

[2]https://redis.io
[3]https://www.memcached.org

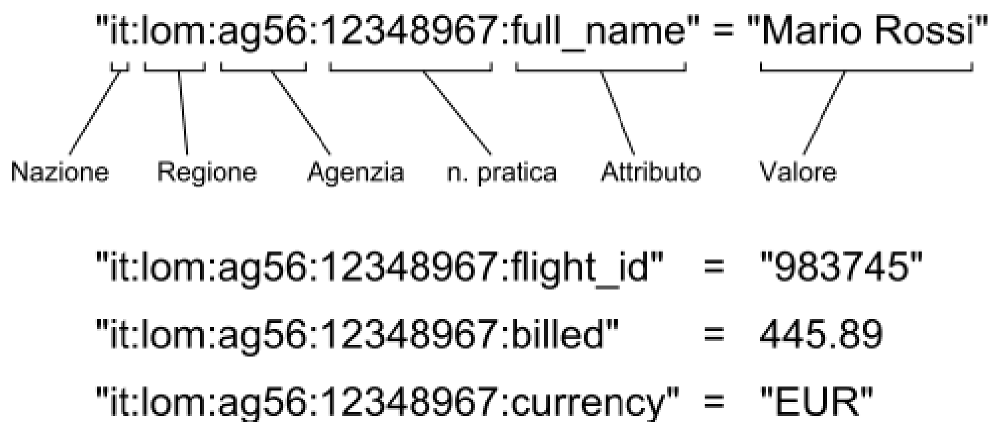


FIGURE 2.3: Key Value Content Example

"shopping carts". Typically this type of datastore supports very well computations that follow MapReduce pattern, as long as database remains simple and poorly structured.

2.6.2 Document Datastore

A document datastore is able to "see" the structure of an aggregate and allows to perform queries based on parts of the same. However, the lack of a predetermined scheme means that datastore cannot optimize data put and data get phases. Almost all implementations allow to insert references to other documents making them very similar to object databases. This way of handling data is very useful for developers, who can have a dynamic database during development phases of a new project or a prototype to attract financiers. The documents can also contain nested structures, such as lists or lists of documents and unlike key-value based databases, these systems support secondary indexes (usually implemented through B+tree), replication and ad hoc queries. The choice to use JSON data format for interaction with applications and Javascript as a programming language was of great importance for development and diffusion of Document Datastore. Both Javascript and JSON are particularly appreciated by developers of interactive web applications. Examples of databases that uses this languages are MongoDB[4] and CouchDB[5].

[4]https://www.mongodb.com
[5]https://couchdb.apache.org

The JSON format in particular is taking the place of XML in context of web services, for various reasons:

- More concise syntax than XML / SOAP
- Ease of use of libraries, available for virtually any programming language (not just Javascript)
- Less "formality" in definition of data interchange formats

All these features however decrease control (data schema) and predictability of services (evolution of message format), making JSON less attractive in enterprise and integration of complex systems, where XML / SOAP continues to be the most used protocol .

2.6.3 Graph Datastore

Aggregates are very useful structures when dealing with data that must be accessed together, but there are cases in which this does not happen. Even in example used so far several times, where customer and his purchase orders seem to be optimally modeled as a single aggregate, everything depends on how that data is used. In fact, if we consider relationships between customers and executed orders, for an application that requires a customer order history, the modeling seen above is just fine, but the same cannot be said in case of an application that needs to access all orders made by all customer. What represents an advantage for first application becomes a disadvantage for second. In this case, in fact, it is necessary to search into all aggregates to obtain necessary information, when instead it would be sufficient to treat orders and customers as two separate aggregates and memorize client's ID in every order, so that when it is necessary to derive who executed a certain order, it is sufficient to perform a further query to DB using customer code. The problem is basically that datastores seen so far, do not give any importance to links between aggregates, indeed these relationships are completely invisible. Breaking down an aggregate brings problems in update operations. In fact, while NoSQL solutions guarantee atomicity property on single aggregate and delegate to developers management of failures, the relational databases offer a simple solution such as ACID transactions, but suffer from low performance due to join operations.

In these cases and in all those where a small amount of records must be managed with respect to number of complex interconnections between them, a NoSQL alternative is given by Graph Datastores. Figure 2.4 shows an example of these systems, where data are treated as nodes of a graph and relations as edges.

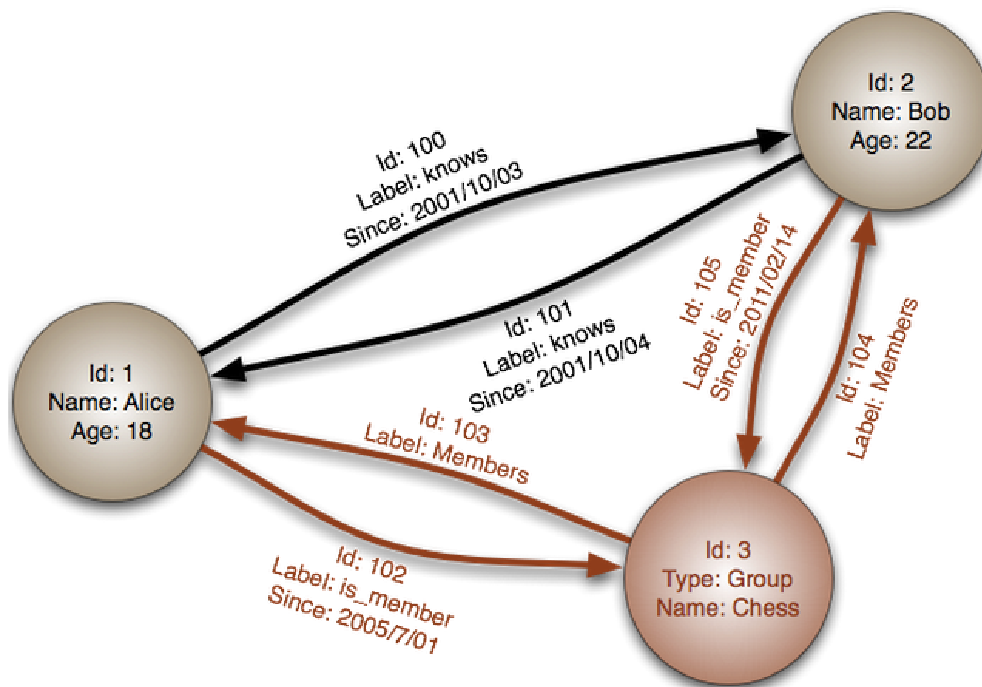


FIGURE 2.4: Graph of relationships between people

In the same way a graph could be used to capture complex relationships of a social network or in a network of preferences on commercial products. A complex query like finding products your friends like, would be easy to calculate with this data model. There are many differences in implementations of these systems, starting from the additional properties that can be specified on nodes and edges up to possibility of directly storing objects written in a particular programming language such Java. Most important difference compared to relational databases remains the way of executing queries: while in Graph Datastores queries allows to *"navigate"* at low cost the pre-built graph, in relational databases it is necessary to perform appropriate and costly join

operations. Problems arise in phase of construction of graph and in high time costs that are shifted on insert operations of nodes and edges.

Graph datastores are becoming indispensable when an analysis of relationships between data is needed, rather than data itself. They allows to extract very valuable data, in reasonable time and in a very elegant way in most cases. Some examples are:

- Classification using proximity and clustering algorithms
- Analysis of various types of flows, for example browsing websites and social networks
- User profiling and suggestions for friendships or purchases

2.6.4 Column-Family Datastore

Most Column-Family Datastore systems have been influenced by project created by Google, known as BigTable (Chang et al., 2008). These are datastores with a tabular structure, made with sparse columns (common columns that have storage optimized for null values) and generally in absence of predefined schemes. Rather than thinking to these systems as huge tables it is more useful to see them in form of two-level maps. In a sense, it is as the tables were growing horizontally (in the sense of the columns) instead of vertically (in the sense of the lines). Each column family represents a type of record, for example a set of columns could be used to identify a user's profile, while another set, and then another column family, to define orders made by that customer and so on. A two-level map (see Figure 2.5) makes a better idea of these systems, because if on the one hand it is possible to access the whole aggregate using row key (as for key-value datastores), on the other it is also possible to use column key to access in detail various object sections, just as accessing a further map.

A read operation takes the following form:

```
get('row_id_1 ', 'column1')
```

although in some implementations (including BigTable) it is even possible to specify an additional parameter, i.e. the timestamp of data. Each column must belong to a single column family and it is assumed that data associated with

Row id	Column families			
	ColumnFamily1	ColumnFamily2	...	ColumnFamilyN
row_id_1	column1="value1" column2="value2"	column3="value3" ...	...	...
row_id_2				
...				
row_id_m				

FIGURE 2.5: Column Family Model Example

columns of a column family are accessed together. This grouping information can be exploited for optimized data access and management. For example a way to exploit this structure could be to store orders made by a customer as an ordered list of columns inside the same column family. The names of these columns could be given by concatenation of purchase date and order ID (eg 20190115-12345) and in this way it becomes simple to execute range queries by date. There is also an extension of Standard Column Family Store, which is represented by *"Super Column Family Store"* (see Figure 2.6). This model adds an additional level of indexing between row key and the set of columns, the so-called *super column*. This key is used to group related attributes between them, belonging to same aggregate in order to obtain various advantages:

- Allows to have a more orderly and easily usable database from applications
- Easy implementation of efficient partitioning strategies (sharding)
- Easy reliability and safety management, dividing database into smaller segments.

Thanks to their adaptability to "rich" data models, Column Family datastores are increasingly used instead of relational model in fields such as websites, web services or as back-end for mobile applications, and thanks to the ease with which they can partitioning data and consequent possibility of greatly optimizing writings are often used to store application logs.

Row id	SuperColumnFamily1		...	...
	ColumnFamily1	ColumnFamily2	...	ColumnFamilyN
row_id_1	column1="value1" column2="value2"	column3="value3" ...	...	...
...				

FIGURE 2.6: Super Column Family Model Example

2.7 Open Source Tools for Big Data analysis

In recent years, several frameworks have been developed that allow to work with Big Data. In this section, we will describe the most widely used open source tools to work efficiently with this type of data.

2.7.1 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is a distributed file system designed to handle large files (several TBs) through sequential read/write operations. Each file is divided into large pieces, stored as local files on different nodes, called Data Nodes.

The framework is composed of a central node called *NameNode* that keeps track of directory structures and file parts location. This node can also be replicated if necessary (Secondary NameNode). To read a file, the client's Hadoop libraries calculate the block index based on offset of file pointer and send a request to NameNode. NameNode then responds with the id of DataNode which has a copy of that block. At this point, client contacts DataNode directly, avoiding the NameNode. To write a file (see Figure 2.7), the client node must first contact NameNode to request primary replica that handles requested file. The NameNode response contains primary and secondary replicas. The client updates all DataNode copies, but these changes are stored in a buffer by each DataNode. After all copies have stored these changes in their buffer, the client commits to NameNode which updates all copies. When all copies have successfully completed the update operation, the primary replica send to client the success response.

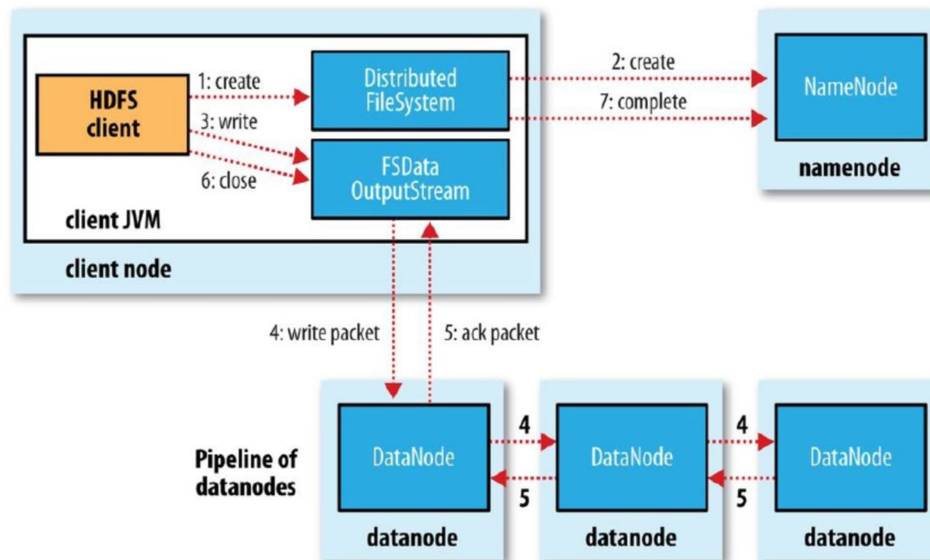


FIGURE 2.7: A read operation on HDFS

The NameNode uses a log file to recover its actions after a crash and also periodically saves its status in a file. To recover from crashes, a new NameNode can start after restoring state from last log file and repeating read operations. Each node periodically sends signals to other nodes to signal its existence. So, when a DataNode crashes, this is immediately detected by NameNode because it no longer receives notifications. The NameNode then removes from cluster the node that crashed and spreads its blocks to other DataNodes that are active. In this way, the replication factor of each piece is maintained on cluster. If DataNode resumes from crash, it sends to NameNode all blocks of files it manages. Each block has a version number, which is incremented with each update. Therefore, NameNode can easily understand if one of DataNode blocks is an outdated version and delete it.

2.7.2 Apache HBase

HBase[6] is a distributed Open Source database modeled on Google's BigTable and written in Java. The project was developed as part of Hadoop Project of

[6]https://hbase.apache.org

Apache Software Foundation and run on HDFS providing capabilities similar to Google's BigTable for Hadoop. It allows to store a large amount of data by offering a scalable and fault-tolerant architecture, through addition of commodity nodes to cluster and native integration with MapReduce. It also provides access to resources through simple REST APIs and ability to work in memory to increase performance. HBase uses a Column Family type data model, so applications that use it, store information in tables composed of rows and columns, whose intersections (cells) are versioned and can contain content of various kinds, formally considered an array of bytes. By default version number is a timestamp generated by HBase during insert operations. Row keys are also an array of bytes and can theoretically contain anything from strings to binary data. Rows in tables are ordered through row key that allows access to contained information through queries. Columns are grouped into column families and can be added at run-time (by specifying column family through a prefix). Physically all columns of a column family are stored together in filesystem. In addition to concept of column, table and row, HBase also uses the so-called regions. In fact, the tables in HBase are automatically partitioned horizontally into regions that are distributed in cluster. Each region includes a subset of rows from a table, in this way, a table that is too large to fit on one server can be distributed to different servers in cluster. Moreover, according to concept of aggregate, HBase row updates are atomic, regardless number of columns of the row.

The HBase Architecture (see Figure 2.8) is composed by a HMaster node that manages the cluster and one or more RegionServer slaves. The HMaster node is responsible for initializing cluster, assigning regions to RegionServers and managing RegionServer failures. As already mentioned, RegionServers manage one or more regions, provide read/write functions and also manage data sent by HMaster node containing information about new cluster nodes. HBase uses HDFS file system and HFile format for data persistence (the equivalent of SSTable on GFS for BigTable), although other implementations such as local filesystem or Amazon S3 can be used. Zookeeper is an independent Open Source project and it is integrated into all versions of HBase. It represents a centralized system that offers: maintenance of configuration and naming information, distributed synchronization, distribution of services and other group services supplying (including quorum-based protocols).

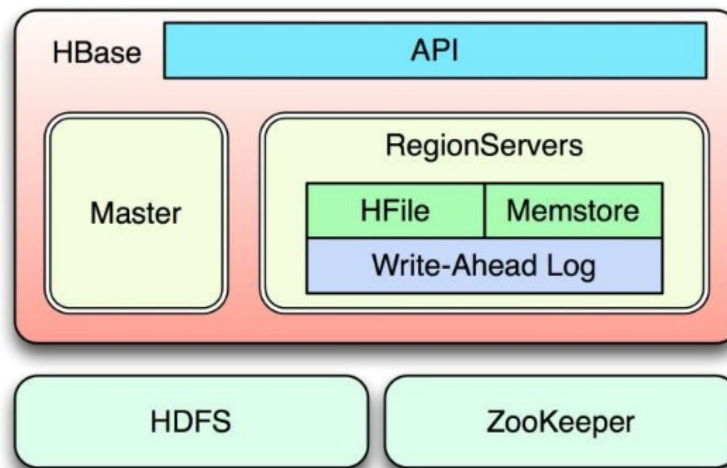


FIGURE 2.8: HBase Architecture

All these types of services are used in many distributed applications, so it is very useful to avoid implementing them again each time.

HBase provides 4 access way:

- *HBase Shell.* A command line shell.
- *API Java Native.* Most common and fastest access path.
- *REST Server.* An integrated REST server provided by internal API.
- *Thrift Server.* A server compatible with different programming languages

HBase allows column-oriented storage only for columns belonging to the same Column Family as shown in Figure 2.9. This structure allows improved performance when different types of analysis are needed on same data. For this reason it is necessary to group in the same column family the data that must be accessed together.

2.7.3 Map Reduce

Parallel computing allows complex applications to handle large problems (petabytes of data) relatively quickly. At these levels, parallel processing must be implemented by distributing computation over a large number (thousands)

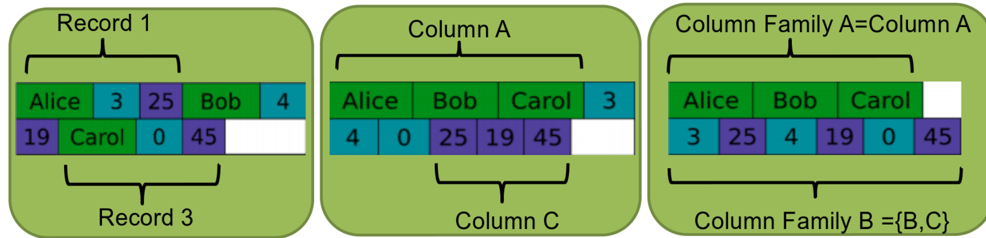


FIGURE 2.9: HBase Storage Methodologies

of ordinary computers and therefore "not expensive". Google is one of main users of parallel computing systems, whose reference software environment is MapReduce (Dean and Ghemawat, 2004), a programming model used to process and analyze large amounts of data in a highly performing way. This system offer a framework that allows to implement extremely parallel applications and simple to use and manage from the point of view of developer. The system automatically parallels computation on a large number of machines, manages failures of individual sub-computations, schedules communication between machines with the aim of make an efficient use of network and disks. The MapReduce model is inspired by map and reduce functions present in Lisp and in other functional languages. In fact, system users specify the computation in terms of *"map"* functions that process key/value pairs to generate a set of intermediate key/value pairs, and a *"reduce"* function that combines all values associated with the same intermediate key.

A typical MapReduce problem is solved in the following steps:

- Iterate over a large number of records
- Extract something interesting from each record (map)
- Order intermediate results and distribute them on relevant reducers
- Aggregate intermediate results (reduce)
- Generate final output

Map and Reduce functions must be specified by user. Map function is applied in parallel on each element of input dataset, it processes a key/value pair to generate a set of key/value intermediate pairs. Reduce function combines

all intermediate values associated with the same intermediate key, therefore it return a list of values, even if typically only one or none is produced.

MapReduce manage a series of problems making them invisible to programmer. In fact, the framework automatically manage distribution of data, starting from assignment of initial data to map workers, up to transfer of intermediate key-value pairs to reduce worker, including optimizations based on principle of locality when possible. Two other operations managed automatically are scheduling, that is the assignment of workers to map or reduce type of tasks and fault management, which identify failed workers and re-execute assigned tasks. Among optimizations that the system provides, there are the ability to move process where data to be analyzed are located and the possibility of executing "combine" function on same machine of mapper as a sort of "mini-reduce" on data just generated, so it is possible to send more elaborate data to reducer in order to save bandwidth. Finally, there is also a solution to problem of existence of slow mappers, which could slow down entire computation. To avoid finding machines that slow down entire process, a redundant execution of map is introduced, so that the first mapper that ends execution sends data to reducer.

The entire data transformation process is described in detail by the following steps:

- A MapReduce job starts with a predefined set of input data (normally placed in a directory of a distributed file system). A master process (which is the central coordinator) is launched and gets a "job configuration", which describes the job to be performed.

- According to job configuration, the master launches several mapper and reducer processes on different machines. It also launches an "input reader" to read data from input directory (located in distributed file system). The input reader partitions the read data and distributes it to various mappers. This phase is called split and starts parallelism.

- After obtaining the data part, the mapper process executes a map function (provided by user) and produces a collection of pairs (key/value). Each pair of results is sorted and, according to key, is sent to the corresponding reducer process. This is the shuffle phase.

- All pairs with same intermediate key are sent to same reducer process, which collects all pairs of same intermediate key and invokes a reduce function (provided by the user) that produces a single pair (key, aggregated values) as a result. This phase is called merge.
- The output of reducer process is collected by output writer. This phase is named join phase and ends the MapReduce parallelism.

Google Map Reduce

The Google MapReduce framework, of which the official operating scheme is shown in Figure 2.10, uses GFS (Google File System) as support for the execution and represents a proprietary solution.

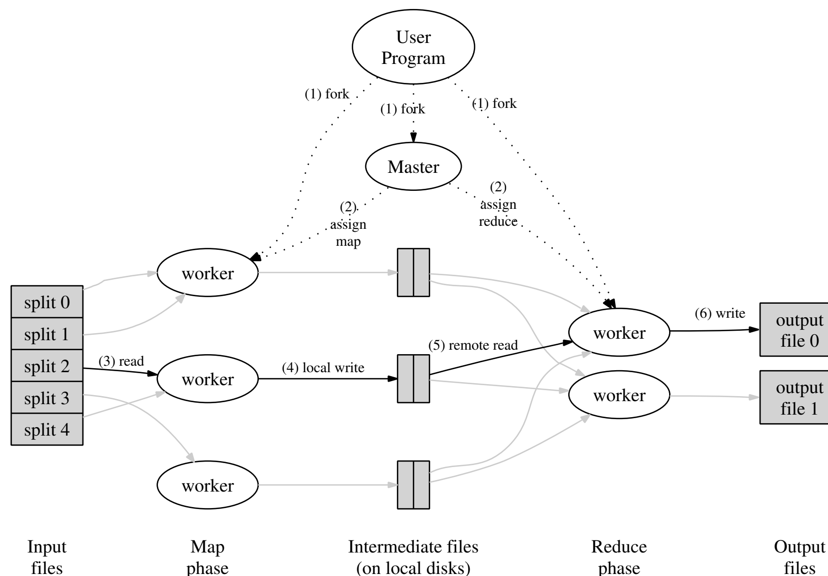


FIGURE 2.10: Google Map Reduce Overview

The sequence of operations performed during a normal operation of this library is described below, this sequence is connected by numbers to the diagram in Figure 2.10.

1. The program divides initial file into M parts typically from 16-64 MB per block (the size of the parts can be specified by user) and Initialize a series of copies of MapReduce program within the cluster.
2. A copy program will be the master. The others will be divided by master into M workers who will manage map tasks, and R workers who will manage reduce tasks.
3. The mappers reads the contents of split input assigned to them and analyze related key/value pairs using the map function specified by user. This function produces a key/value intermediate pairs that are buffered in memory.
4. Periodically, buffer is written to local disk, partitioned into R regions by partition function. The locations of these regions are sent back to master who is responsible for forwarding them to reduce worker.
5. When a reduce worker is notified by the master, this uses a remote procedure to read data stored in local mapper disk. When a reducer reads all data from its partitions, it sorts data by intermediate key, in order to group all occurrences with the same key.
6. For each intermediate key, the reducer finds a list of values that are sent to reduce function specified by user. The output of a reduce function is added to a final output file for this partition. When all map tasks and reduce tasks are completed, the master sends the results to the user node.

Apache Hadoop

Apache Hadoop is a framework written entirely in Java and corresponds to an open source version of Google MapReduce. In this case, however, the distributed file system used is HDFS. The operation of Hadoop library is exactly the same of those of Google library, so in this paragraph will be described the entities participating in the management of MapReduce process rather than operations seen above.

The execution of process (see Figure 2.11) begins when client, on its own Java Virtual Machine, requests the creation of a Job, then it loads system parameters and copies all necessary resources (mapper classes and reducer, input and output files, split sizes where the initial file will be divided, etc.) into a

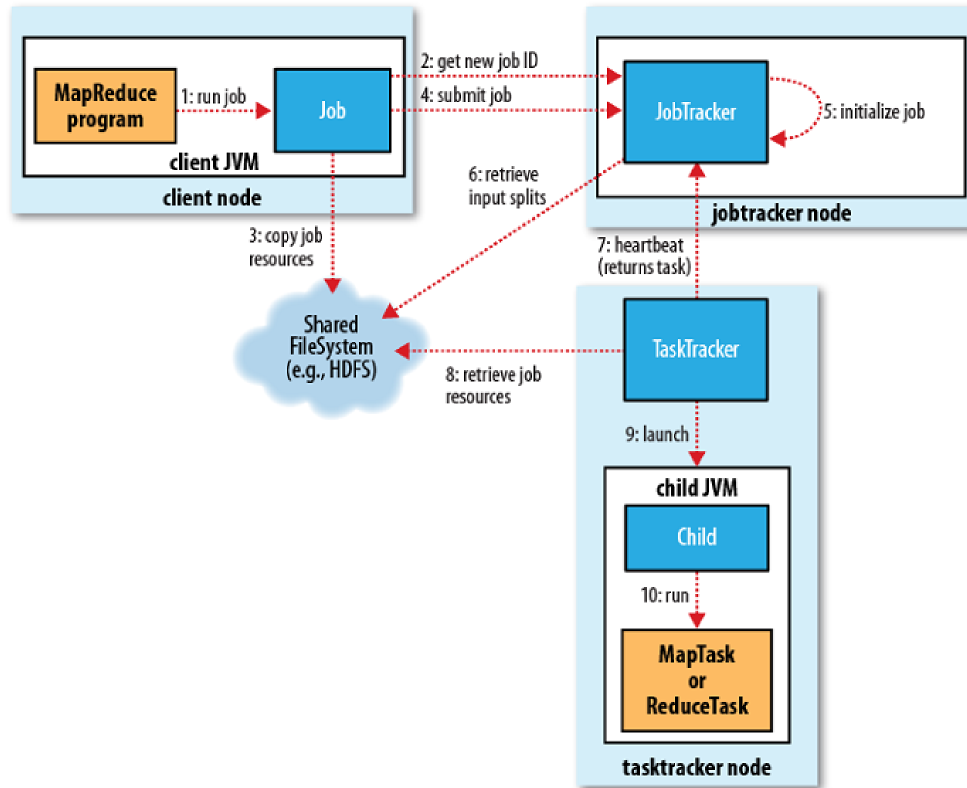


FIGURE 2.11: Hadoop Map Reduce Working Scheme

location on HDFS file system. The program notifies submission to JobTracker thus the latter returns a Job Id to client and the computation begins. JobTracker splt and assigns data to TaskTrackers based on proximity principle (or location), having precedence first those on the same node, then on the same rack in the cluster and finally those on the same network. Assigned TaskTracker creates a new MapTask process that extracts input data from split (using RecordReader and InputFormat classes) and invokes map function specified by user. This function emits a number of key pairs/values in the buffer memory. Data stored in memory must be periodically saved on disk (to avoid memory saturation). For efficiency reasons it is possible to use a local reduce that invokes the combine function to aggregate values for each key. At this point a partition function is invoked which calculates the relative reduce node for each key, which will reduce the data. Then the intermediate key/value pairs are

written to R local files (assuming that there are R reducer nodes). When the JobTracker understands that some map tasks have been completed, it begins to allocate reduce tasks to TaskTrackers, which remotely download the region file of completed map nodes and link them all into one file. When all map tasks have been completed, Job Tracker notify the Task Trackers to proceed with the reduce phase. To allow framework to manage the failures, TaskTracker periodically sends their status to JobTracker. If the latter does not receive notifications for a long time from a node, it assumes that it has crashed and reassigns its tasks to another TaskTracker.

2.7.4 Apache Hive

Apache Hive is an Open Source project, initially developed by Facebook, which allows to execute SQL queries on a DB stored in the distributed HDFS file system using the Map Reduce framework. It is a Data Warehousing infrastructure built on top of Hadoop ecosystem, which simplifies the querying and the management of large datasets stored on distributed storage. Hive provides a mechanism to create data structures and perform queries using SQL language, which automatically translates SQL queries into Map Reduce jobs to be executed on Hadoop. An Hadoop cluster is an archive of heterogeneous data from many sources and in different formats. Hive allows the user to explore, analyze these data and transform them into business. Within a Hive database, table data is serialized and each table corresponds to a directory on HDFS.

Hive provides the following features:

- Tools to enable easy access to data via SQL, thus enabling data warehousing tasks such as extract/transform/load (ETL), reporting, and data analysis.
- A mechanism to impose structure on a variety of data formats
- Access to files stored either directly in HDFS or in other data storage systems such as HBase
- Query execution via Apache Spark, or MapReduce

Figure 2.12 shows the major components of Hive and its interactions with Hadoop. As shown in that figure, the main components of Hive are:

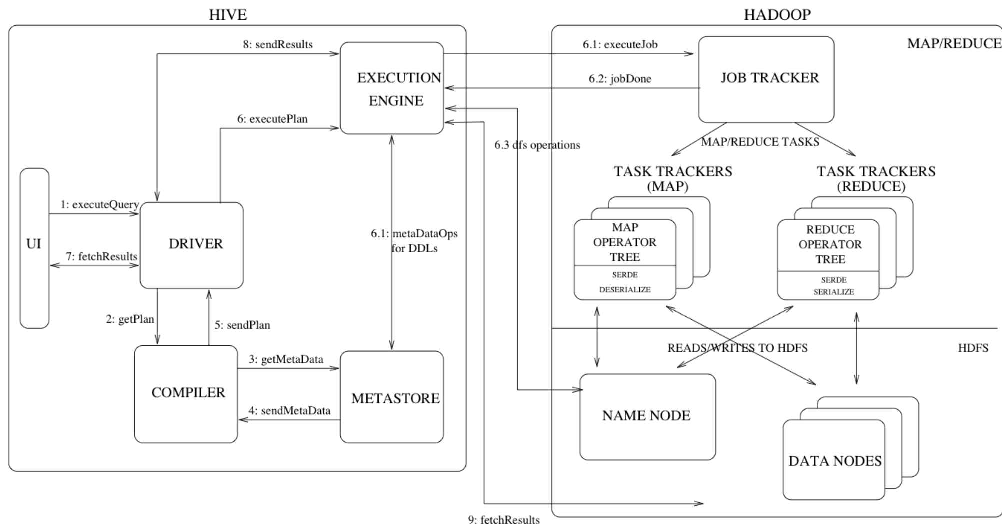


FIGURE 2.12: Hive Architecture

- ***UI***. The user interface for users to submit queries and other operations to the system. The system has a command line interface and a web based GUI.
- ***Driver***. The component which receives the queries. This component implements the notion of session handles and provides execute and fetch APIs modeled on JDBC/ODBC interfaces.
- ***Compiler***. The component that parses the query and makes semantic analysis on different query blocks and query expressions and eventually generates an execution plan with the help of the table and partition metadata looked up from the metastore.
- ***Metastore***. The component that stores all the structure information of the various tables and partitions in the warehouse including column and column type information, the serializers and deserializers necessary to read and write data and the corresponding HDFS files where the data is stored.
- ***Execution Engine***. The component which executes the execution plan created by the compiler. The plan is a DAG of stages. The execution

engine manages the dependencies between these different stages of the plan and executes these stages on the appropriate system components.

There are two different aspects concerning the storage of tables in Hive: the line format and the file format. Line format regulates how the fields are stored in a particular line. In Hive this format is defined by SerDe (Serializer-Deserializer). When it acts as a deserializer, i.e. in the case in which a table is queried, a SerDe converts a row of data from bytes into objects used internally by Hive to operate with that data. When instead the SerDe acts as a serializer, i.e. in INSERT case, the internal representation of Hive is stored as bytes in output file. On the other hand, the file format dictates the type of file to be used to store data on file system. The simplest format available is text file, but there are other more complex formats such as RCFile (a columnar file data structure) and the binary format.

2.7.5 Apache Spark

Apache Spark is an open source distributed computing framework developed by the University of California's Berkeley's AMPLab and later donated to Apache Software Foundation. Characterized by the ability to store (usually partial) results in central memory, it offers a valid alternative to Map Reduce, which necessarily stores results of computations on disk. The optimal use of memory allows Spark to be orders of magnitude faster, compared to MapReduce, in the execution of iterative algorithms, i.e. for those algorithms that iteratively carry out the same instructions on different data until a certain condition occurs. This framework remains a valid alternative, probably more flexible and easy to use, even when you want or need to use the hard disk, for example because the data does not all fit into memory. It offers an API much easier to use than MapReduce paradigm. Data can be read from a multitude of sources, including HDFS, Amazon S3, Cassandra, HBase, etc., as well as numerous structured, semi-structured or unstructured file formats. Spark can be used in Java, Scala, Python and R programming languages.

Apache Spark has as its architectural foundation the resilient distributed dataset (RDD), a read-only multi-set of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way.

Figure 2.13 show the Spark stack composed of the following modules:

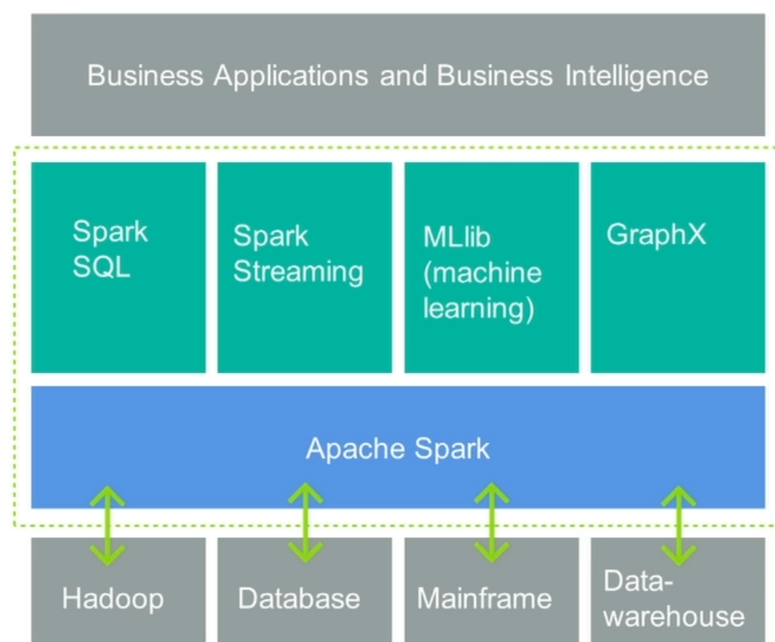


FIGURE 2.13: Spark Stack

- **Spark Core**: contains basic functionalities of Spark, including memory management, network management, scheduling, recovery from cluster node failure, etc. The Spark core API manage RDDs.
- **Spark SQL**: it is a component on top of Spark Core that introduced a data abstraction called DataFrames, which provides support for structured and semi-structured data. Spark SQL provides a domain-specific language (DSL) to manipulate DataFrames in Scala, Java, or Python. It also provides SQL language support, with command-line interfaces and ODBC/JDBC server.
- **Spark Streaming**: Spark Streaming allows to analyze real-time data flows, such as error logs or stream of tweets. This component perfectly fit with the others modules. For example, it is possible to make a join between data flow and an historical database in real time. Streams flows can come from sources like Apache Flume, Kafka, or HDFS, and are used in small batches for analysis.

- **MLlib**: It is a highly optimized machine learning library, which exploits in memory data stored. Many of the machine learning algorithms are iterative, so MLlib offers the possibility to use only algorithms that are inherently parallelizable, including linear regression, K-means, random forests, etc. It is possible to use Spark Streaming streams for machine algorithms training phase.

- **GraphX**: GraphX is a library for the analysis of very large graphs that could not be analyzed on a single machine (for example the graphs of social networks). A graph is a collection of nodes (or vertices) connected by arcs. The library offers different algorithms such as PageRank (to measure how "important" each node of a graph is), computation of connected components, computation of triangles, etc. GraphX unifies ETL, exploratory analysis and iterative calculation on graphs in a single system. The graphs are managed like the other datasets, with which you can even make joins.

Spark uses a master/slave architecture, where there is one coordinator process and many worker processes. The coordinator is named *driver*, while the worker is named *executor*. Since each execution takes place in a separate process, different applications cannot share data unless they are first written to disk. Working on a single node allows to have only one process that contains both the *driver* and an *executor*, but this is a special case and usual it is used for application testing.

The basic Apache Spark architecture is shown in the Figure 2.14.

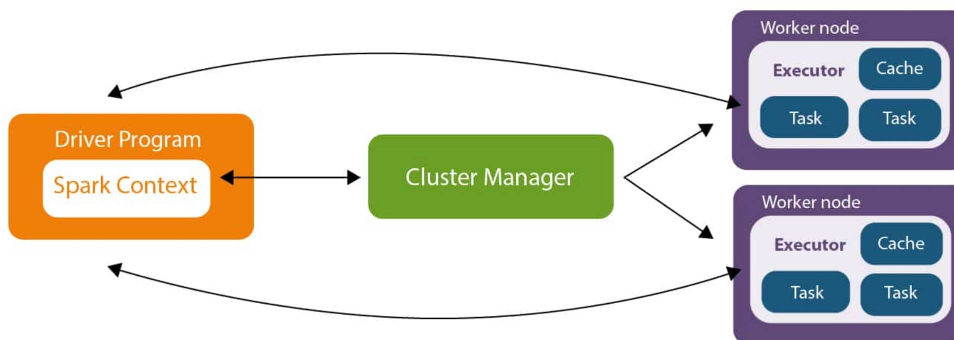


FIGURE 2.14: Spark Architecture

The *driver* manages the execution of a Spark program, deciding the tasks to be performed by *executor* processes, which are usually running in the cluster (server machines). This process is usually running on the client machine. In the main program of a Spark application (the *driver* program) there is an object of type SparkContext, whose instance communicates with the cluster resource manager to request a set of resources (e.g. RAM, core) for executors. Several cluster managers are supported including YARN, Mesos, EC2 and Spark's Standalone cluster manager.

The driver is the main process and contains the source code of transformation operations and actions on RDD that must be performed in parallel by executor processes distributed in the cluster.

Executors carry out tasks chosen by the driver. Each application run on its own executors (i.e. its own processes) each of which can have multiple threads. The executors have an amount of assigned memory (configurable) which allows to store part of datasets in memory if requested by user application. The executors of different Spark applications cannot communicate with each other, in this way, different applications cannot share data between them unless they are first written to disk. The life cycle of an executor is valid for the duration of one application; if an executor fails, Spark continue to run program by recalculating only lost data.

Chapter 3

Analytics on Big Data

3.1 Background

Due to great interest in Big Data by academia, industry and individual, a variety of architectures for Big Data Analysis have been defined in recent years (Manogaran and Lopez, 2017) (Liu, Iftikhar, and Xie, 2014) . However there is no generic architecture available for analytical big data systems and a number of small-scale architectures have been proposed by various organisations to fulfil their own requirements (Thota and Vijayakumar, 2016). Most of these architectures are developed for specific purposes such as stream processing, batch processing, security and storage. A part of these architectures are product-oriented and thereby they limit their scope to specific products from a company, while other architectures are technology-oriented thereby skipping a functional view and mappings of technology to functions. In this section we focus on two well known and state-of-the-art architectures: Lambda and Kappa.

3.1.1 The Lambda Architecture

Marz and Warren (Marz and Warren, 2015) introduced the Lambda Architecture that provides a set of architectural principles to ingest and process both stream and batch data. The main idea of the Lambda Architecture is to build Big Data systems as a series of layers: Batch Layers, Serving Layer and Speed Layer. Each layer satisfies a subset of the properties and builds upon the functionality provided by the layers beneath it.

The **Batch Layer** stores the master copy of the dataset and precomputes batch views on that master dataset. The master dataset can be thought as

a very large list of records. This layer needs to be able to do two things: store an immutable, constantly growing master dataset, and compute arbitrary functions on that dataset. This type of processing is best done using batch-processing like Map-Reduce. The output of this layer (batch view) can be represented by the equation:

```
batch view = function(all data);
```

The **Serving Layer** is always connected with the batch layer to store the batch views. This layer is a specialized distributed database that loads in a batch view and makes it possible to do random reads on it. When new batch views are available, the serving layer automatically swaps those in so that more up-to-date results are available.

In general, due to high latency the batch views is always out of date. This latency issue can be solved by **Speed Layer**, because this layer is always responsible for any data that are not yet available in the Serving Layer. It updates the realtime views as it receives new data instead of recomputing the views from scratch like the Batch Layer does. The speed layer does incremental computation instead of the recomputation done in the Batch Layer. It is possible to formalize the data flow on the Speed Layer with the following equation:

```
realtime view = function (realtime view, new data)
```

Figure 3.1 shows the complete Lambda Architecture.

To solve queries it is necessary to look at both the batch and realtime views and merge the results together instead of resolve queries by just using only the batch view. The Lambda Architecture in full is summarized by these three equations:

```
Batch View = function (all data)
Realtime View = function (realtime view, new data)
Query = function (batch view, realtime view)
```

3.1.2 The Kappa Architecture

Kappa Architecture was first proposed by Kreps (Kreps, 2014) for management and analisys of data stream. It simplifies the Lambda architecture by

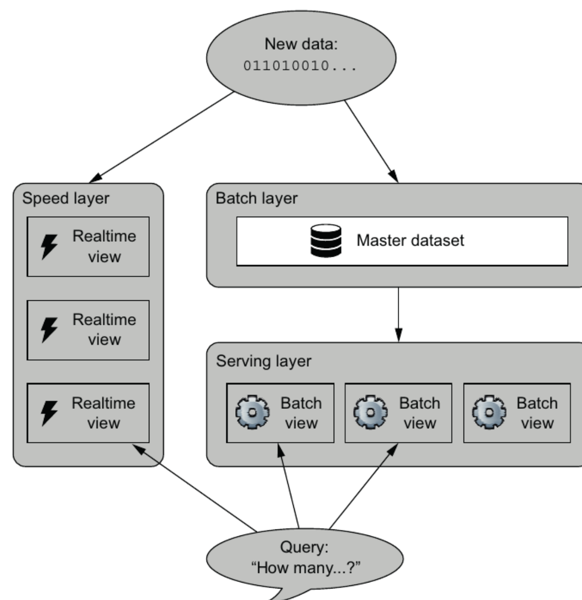


FIGURE 3.1: Lambda Architecture

removing the batch layer and replacing it with a streaming layer. The basic idea is that a batch is a data set with a start and an end (bounded), while a stream has no start or end and is infinite (unbounded). Because a batch is a bounded stream, one can conclude that batch processing is a subset of stream processing. Hence, the Lambda batch layer results can also be obtained by using a streaming engine. This simplification reduces the architecture to a single streaming engine capable of ingesting the needed volumes of data. The advantage of Kappa architecture over Lambda architecture is in simplicity. With Lambda, you would need to maintain two different processes and possibly different set of codes which can put pressure on small budget projects. In Kappa, there's only one level of process and one set of code so it's cheaper to implement. Also from end-user perspective, with Kappa there's only one plug-in required to read the data while in Lambda there are two different views for batch and real-time data results.

Figure 3.2 shows the Kappa Architecture where the Straming Layer ingests ordered data events while the Serving Layer manages the query results.

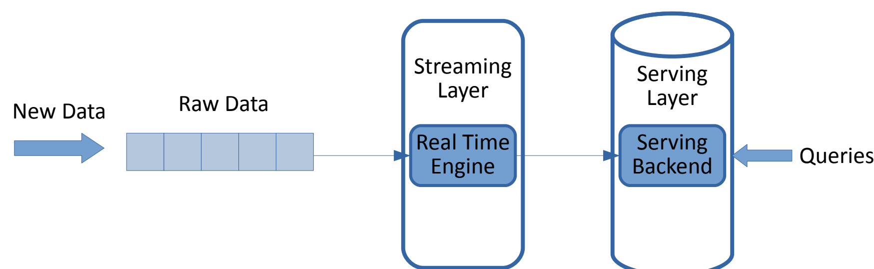


FIGURE 3.2: Kappa Architecture

3.1.3 Architectures Implementation

The implementation of these architectures can be done by using different frameworks. With regard to Lambda Architecture, a first implementation definition was illustrated by Marz and Warren, 2015. They proposed to use HDFS as Master Dataset and Hadoop as Batch View Engine for the Batch Layer; while to use ElephantDB[1] as Batch View Storage for the Serving Layer. Finally for the Speed Layer to use Kafka as Realtime View engine and Cassandra[2] as Real Time View storage.

A different implementation of Lambda Architecture is proposed in Kiran et al., 2015 where a benchmark was conducted on the popular cloud computing platform Amazon EC2. In this paper the architecture is implemented with S3[3] as Master Dataset and Elastic Map Reduce[4] as Batch View Engine for Batch Layer while into the Speed Layer the flows are managed with Kinesis Stream[5] and stored on S3 Storage.

For Kappa Architecture, instead, Kreps, 2014 proposed to use Apache Kafka as Log Data Store and Apache Samza[6] as distributed stream processing framework in Real Time Engine.

In Persico et al., 2018 a benchmark of both architectures was performed. The Lambda Architecture has been implemented using Apache Spark while for Kappa Architecture it was used Apache Storm. The benchmark has been carried out on Microsoft Azure public cloud using as benchmark a diffusion

[1] https://github.com/nathanmarz/elephantdb
[2] http://cassandra.apache.org/
[3] https://aws.amazon.com/it/s3
[4] https://aws.amazon.com/emr/
[5] https://aws.amazon.com/kinesis/data-streams/
[6] https://samza.apache.org/

network algorithm. Experimental campaigns have been carried out on the Yahoo Flickr Creative Commons 100 Million (YFCC100M)[7] and the results showed that Lambda outperforms Kappa architecture for this class of problems.

3.2 Search Based Analysis Architecture

3.2.1 Complex Search

Search engines have been proposed since the early stage of Internet, however, results returned by search engines are often quite far from the expected query answers from a user viewpoint. Indeed, search results can be improved by building a custom map that, based on the initial query results, tries to learn additional knowledge about data being queried by iterative refinement of search dimensions and parameters. Figure 3.3 shows the above mentioned scenario.

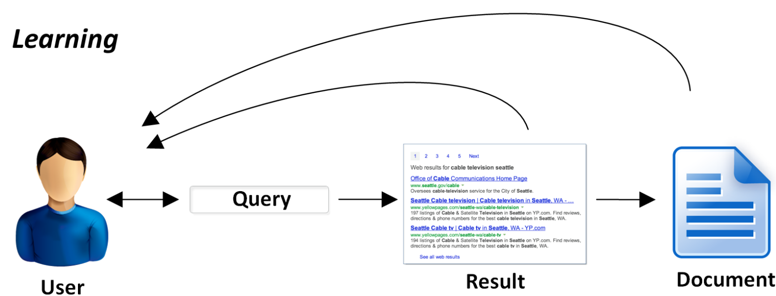


FIGURE 3.3: Learning By Results

In this scenario, the type of query being performed plays a crucial role. Unfortunately, this process is suitable only for simple search of well-defined terms. On the contrary, dynamic learning by exploratory research cannot be performed by this naive process. Obviously enough, for well defined queries, a search engine like Google, is able to provide correct results in a few milliseconds [8].

[7]www.yfcc100m.org

[8]As a matter of fact, due to its quick result presentation, many users go through Google even if they exactly know the URLs of the resources they are interested in

However, in some cases users do not know exactly how to find the desired information about an object or a service (e.g. a book or a restaurant). In this case, the model depicted in Figure 3.4 is more suitable.

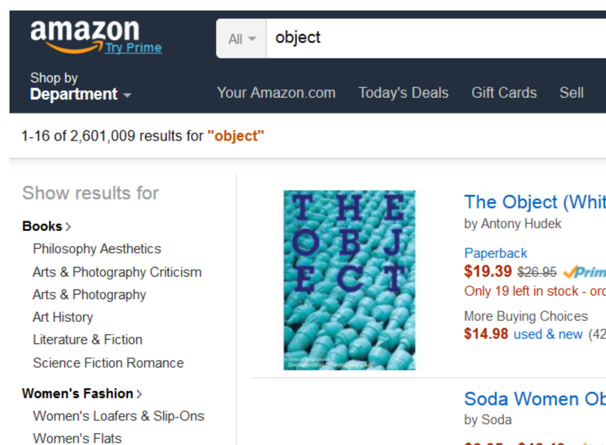


FIGURE 3.4: Amazon search

More in detail, Amazon-like search tools, feature product categorization and recommender systems, thus making the user search experience quite interactive and iterative. In a sense, intermediate results guide users to a better definition of target information. Furthermore, search engines usually allow non-structured queries (referred as “ranked retrieval") whose results are sorted according to some relevance criteria w.r.t. the target search. As a matter of fact, these queries are easier to pose by users compared to boolean expressions, but they can produce low quality results.

In order to overcome this limitation, some categorization service like Yahoo!Directory, exploits context information [9]. More in detail, directory contents are hierarchically organized in order to guide users through a subset of documents potentially related to information being queried, thus limiting the possibility to input free text queries. In this respect, users re-think and refine their needs by learning the adjustments to the search being performed by exploiting the available choices. To better understand how directory navigation works, we resurge to accommodation booking portals analogy. Indeed, those portals offer a hierarchical navigation systems, i.e. from the home page, user can choose the desired country, then he can specify the city and finally the type

[9]Yahoo!Directory is no longer active since 2014, however it is worth mentioning as it was one of the first services for massive assisted browsing

of structure he is interested in. This navigation model suffers a great limitation due to taxonomy specification. Indeed, taxonomy specified by the service designer may not meet user needs. A solution to overcome the above mentioned limitations is the implementation of *faceted* navigation that helps users in the information “surfing" process. Next section is devoted to clarify this issue.

Consider the faceted view of a search engine depicted in Figure 3.5. Starting from the home page, the user has the chance to search information about the location *and* several attributes pertaining to the search (i.e. the type of structure, the rating and so on).

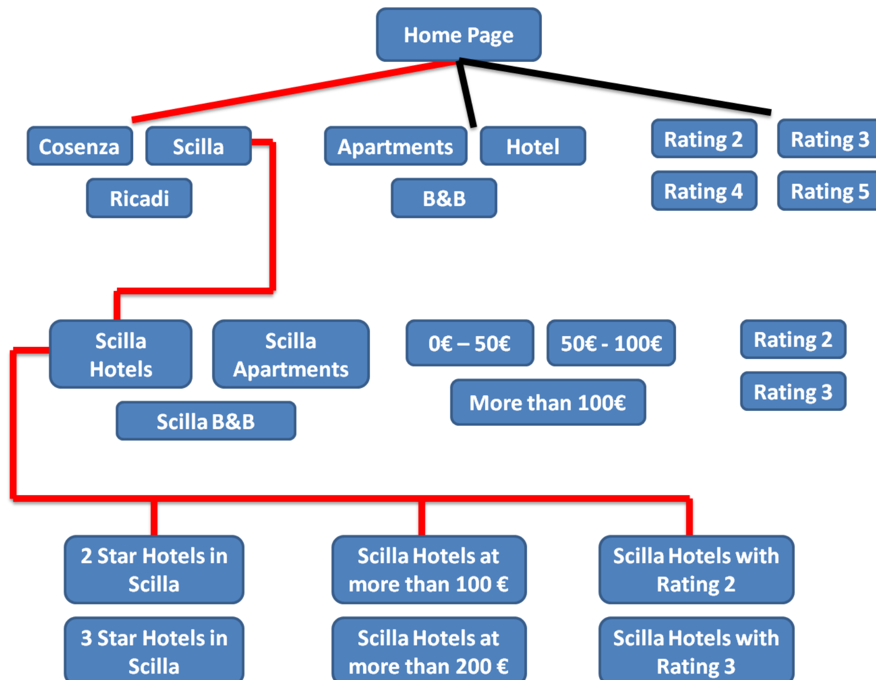


FIGURE 3.5: Faceted Navigation Example

For example, he can browse the cities $(Cosenza, Scilla, \ldots)$, the structure types $(Hotel, B\&B, apartment, \ldots)$ and their star rating $(2, 3, 4, \ldots)$. As a feature is selected, the user can choose other attributes among those available for the current search status. Moreover, during the browsing process, it is also possible to discard features no longer relevant to the search (i.e. user can perform *dimensional filtering*). This iterative process guides the user through the

accommodation search by selecting a custom path instead of a hierarchy provided by the service designer.

It is worth noticing that efficient faceted navigation (i.e. easy to use and providing access to richer information) relies on the availability of a meaningful feature set that characterize the domain being searched. In particular, it is mandatory, for implementing effective faceted navigation, that the objects a user may choose share some common feature.

Browsing through facets makes data exploration very similar to OLAP analysis. Drill-down and roll-up operation can be performed by selecting items from facets, while slice and dice operations can be simulated using appropriate filters.

3.2.2 The Sigma Architecture

A new architecture, named *Sigma*, is proposed to provide a solution for build a complete Big Data System, interactive and scalable, using a variety of tools and techniques. This Architecture is composed of three Layer as depicted in Figure 3.6.

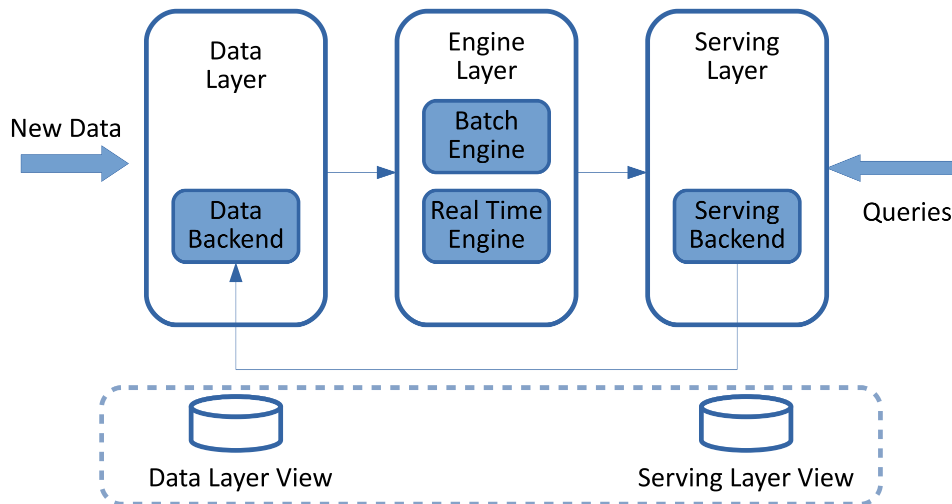


FIGURE 3.6: Sigma Architecture

The **Data Layer** stores a copy of all raw data that come to the system. This layer need to be able to store an immutable, constantly growing dataset (*Data Layer View*) and to offer a backend able to perform random reads on the whole

content. The **Engine Layer** is responsible to compute arbitrary function on the data layer and to store results on the Serving Layer. This computation can be executed in batch mode on the whole Data Layer View (via Batch Engine) or can be executed in real time mode every time new data arrives in the Data Layer (via Real Time Engine). The **Serving Layer** is a specialized distributed database that loads in the results of Engine Layer Computation (*Serving Layer View*) and makes it possible to do random reads on it.

Let's start from new data coming into the system. Typically Big Data comes under forms of stream, so the Data Layer need to serialize these flows and persist it. Moreover, in this step a first operation of ETL is performed on incoming data flows. The Sigma Architecture allows to manage different situations. It is possible that data are already present in a data warehouse and in this case it is convenient to create the Serving Layer View using a batch operation via Batch Engine using for example the Map Reduce paradigm. In other situations, instead, data can only come in the form of streams and in this case it is convenient to create the Serving Layer View using real time functions via Real Time Engine. Maintaining all data on Data Layer allows to easily recover the system in case of failure on Serving Layer and allows to extends the Data Layer content by applying Batch Engine functions on the starting contents of the Data Layer. Once the Engine Layer created the Serving Layer View on the Serving Layer, the system is ready to receive queries via Serving Backend. To satisfy incoming queries, the Serving Backend uses data from Serving Layer View and data from Data Layer View, performing one or more requests to Data Backend. This mechanism is useful to optimize the content of the Serving Layer allowing him to keep only an index for some data thus avoiding excessive data replication between Data Layer and Serving Layer. The Sigma Architecture can therefore be summarized by these equations:

```
Data Layer View = all input data
Serving Layer View = function (Data Layer View)
Query = function (Serving Layer View, Data Layer View)
```

The Sigma Architecture differs from Lambda Architecture in many respects. The Data Layer is similar to the Batch Layer, but in the Sigma Architecture it can be accessed both from Engine Layer and from Serving Layer. For the computation of data on Serving Layer, the proposed Architecture can use Batch **or** Real Time Engine and not all two simultaneously as in Lambda. As explained

above, the Batch Engine is typically used when persistent data is available, when there are system failures for system recovery or when there is need to operate on Data Layer. The Real Time Engine, instead, is used to compute data flows. An implementation of the Sigma Architecture could be done only with Batch Engine or only with Real Time Engine thus becoming similar to the Kappa Architecture. The decoupling between Batch and Real Time Engine allows to considerably simplify the implementation and the maintenance of the system as this was one of the main criticisms of Lambda.

As for the Kappa architecture, the proposed solution add the Data Layer and the use of batch processing. Adding this level allows you to decouple the input data from the Engine Layer at the cost of slightly increasing the complexity of the system. However, this layer optimizes the data stored on Serving Layer reducing its size. Moreover, the use of Batch Engine allows to use Sigma Architecture in a in a greater number of instances w.r.t Kappa, for example when you need to analyze persistent data or when you need to perform batch operations on input data.

3.2.3 A Sigma Implementation for Search Based Analysis

In this section an implementation of the Sigma Architecture will be presented with the primary objective of offering a full-text search mechanism, which is interactive, scalable on Big Data and which is also dynamic. To design and develop a similar tool it is necessary to use complex data indexing and management. To this end, some famous open source projects have been integrated into a final system like Apache Hadoop, Flume, HBase, Solr, Lily HBase Indexer and Hue. Figure 3.7 show system architecture.

Let's start from Data Layer with data extraction. It is possible to extract data from different information sources and in this case it was supposed to have data streams. For this purpose it was decided to exploit the Apache Flume[10] project, which is a reliable and distributed service to efficiently collect, aggregate and move large amounts of data. It has an architecture that is simple and flexible, based on data streaming management. It is also a robust and fault tolerant system with configurable reliability mechanisms. This tool was chosen over others (like Apache Kafka [11]) for the close coupling with the

[10] https://flume.apache.org
[11] https://kafka.apache.org

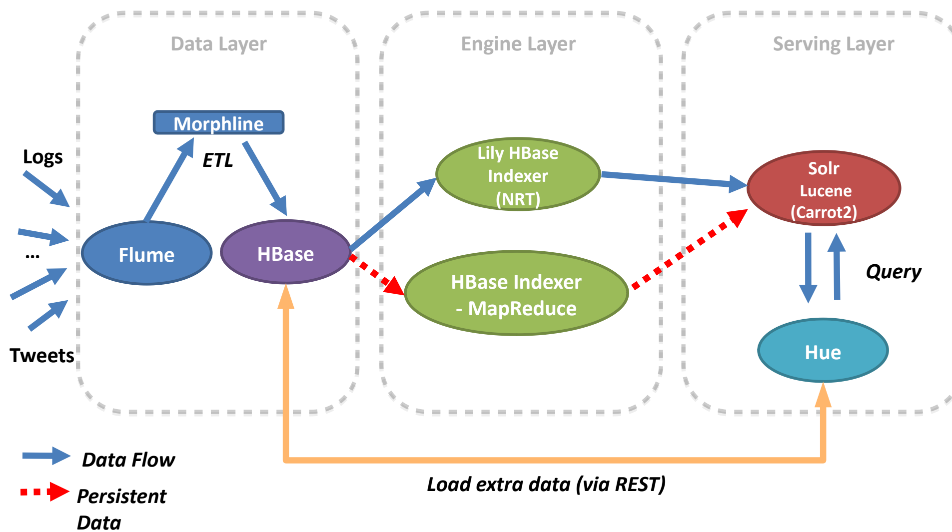


FIGURE 3.7: Big Data Search Architecture

Hadoop environment: it is a special purpose tool for sending data into HDFS and HBase and therefore it requires less configuration time than other tools.

Once the data have been intercepted by Flume, these are stored on the data warehouse created with HBase and through a project known as Morphline, ETL procedures are applied "on the fly", which clean up input data and perform mapping with the columns structure of the datastore. The choice to use a column family oriented datastore like HBase is necessary as in this architecture we need to make an index on some parts of data content and to avoid to create an inverted index on all fields of a column we need to use a row id to retrieve not-indexed fields contents. Also the columnar structure allows to put in the same column-family the content that need to be indexed speeding up the indexing phase. Once data arrived on HBase, there are two possible situations in the Engine Layer. In the case in which data are already present in data warehouse, it is desirable to perform a static indexing operation. To this end it is possible to use the Map Reduce framework, through which it is possible to exploit all the resources present in the cluster and thus complete the operation as quickly as possible. For this purpose HBase Map Reduce Indexer library has been used and will be discussed later. If instead you only need to index new data entered in the data warehouse, then you can take advantage

of a project called Lily Indexer [12]. The data flows are thus sent on HBase, or indexed and stored on Solr.

Apache Hadoop works well for off-line analysis, as in the case of data indexing. However in case of full-text search, for the implementation of the Serving Layer, it is necessary to refer to another project created specifically for this purpose: Apache Solr[13]. It allows users to search for arbitrary keywords in any field that has been previously indexed and allows to quickly view a number of documents that satisfy a specific query. All thanks to the well-known mechanism of inverted index. The set of operations that can be performed via Solr is really extensive and includes field facets, arbitrary range queries and pivot facets, all operations that have much in common with the classic OLAP operators (slice, dice, drill-down, roll-up, pivoting) and that make Solr an excellent real-time analysis engine for text documents.

As a toy example, we can think to a website that collects log files and additional information on user behavior. This type of data can be indexed by Solr in order to be able to perform queries on specific time ranges and keywords or to derive graphs on aggregate information, such as the growth over time of the number of registered users or operations performed of a certain type. In this context, the Carrot2[14] module, a Solr system plugin, makes search even more effective, allowing real-time clustering on search results. The system therefore allows to obtain new dimensions of analysis with some limitations, including the maximum number of results that can be analyzed. Increasing this limit too much would make the entire functionality unusable, due to the consequent waiting times. To display search results, the Hue[15] project has been used; it is a software that can be natively integrated with other components of Apache Hadoop ecosystem. From a single and simple interface, it is possible to offer different functionalities to all possible types of users, including end users and domain experts. In particular, end user can only be interested to search in a simplified way among the data indexed on Solr, while for domain expert it is also possible to display/modify the dimensions available in the data warehouse and to build categories starting from search results that can be grouped.

[12]https://ngdata.github.io/hbase-indexer
[13]http://lucene.apache.org/solr
[14]https://search.carrot2.org
[15]https://gethue.com

A relevant aspect of the proposed solution is to avoid the transfer of whole data warehouse inside Solr. There is a distinction between data that are only indexed and usable only to search for documents and data also stored on Solr. It is possible to access the last type directly, without going through the source from which the data were extracted. In this regard, we keep the minimum amount of information possible on Solr and make the other information accessible through REST API Server.

In the proposed architecture, as can be seen in Figure 3.7, two different data management methods have been provided: one for persistent data and the other for data flows. The blue arrows indicate the components that are used in data flows processing, while with red arrows those oriented to persistent data. It is a general architecture that can be used in different contexts.

In summary, this implementation of Sigma Architecture is able to offer:

- Full-text search, interactive, scalable and with a flexible indexing system.
- Possibility of discovering some new dimensions of analysis (e.g. *"similar results"*) starting from results obtained by search
- Data Analysis through facets in a similar way to OLAP analysis

A convenient faceted navigation is available, which can be of two types:

- Static facet navigation: these are classic categories on attributes that are defined a priori.
- Dynamic faceted navigation: in this case, categories are generated starting from the search results and without defining anything a priori, using clustering algorithms.

3.2.4 System Description and Integration

The proposed architecture implementation is based on HBase datastore as this system provide the possibility of storing data in a highly scalable and fault-tolerant way. A main feature of this datastore is the lack of rigid schemes in the dataset structure, in this way it is possible to add new attributes at any time, without modifying the already existing data. HBase is certainly one of the

most suitable solutions when you have an enormous amount of data to analyze and you want to give priority to reading operations. Unfortunately, without the support of an indexing system, the response time to full-text search, would remain too high, making the service unusable by end user. This is just one of the reasons why it was decided to integrate the Solr project into the proposed solution. Among the most interesting features, we can find, in fact, those related to the generation of facets through which the user can not only search through the data, but navigate it easily in a similar way like classic OLAP tools or even be guided by the system itself through suggestions of similar results to those searched. It is important to highlight the fact that starting from the last Solr release (SolrCloud), the index management becomes distributed, because it no longer refers to the local file system but to HDFS for the storaging and to Zookeeper system for communication, the same services to which the HBase project is based. Another aspect not to be underestimated in the integration of the two systems is the full support for *multivalue attributes*; Solr allow to memorize them natively (it has a specific multivalued type), on HBase instead one of the possible ways to manage them, consists in adding a suffix to interested columns. Supposing, for example, to have in the data warehouse an attribute that refers to various email contacts of a receptive structure, we could think of modeling the storage in the following way on HBase:

```
columnFamily_registry [email_1:<value1>, email_2:<value2>, etc..]
```

while on Solr we could define a field of this type:

```
<field name="email" type="string" indexed="true" multivalued="true"
stored=’’true’’/>
```

To make a whole indexing of HBase content, the Map Reduce framework can be exploited. The advantages of this choice consist in the possibility of executing the entire operation of generation/construction of the index in batch mode, in order to commit all resources available to cluster. For the other cases, the Lily project represents a more suitable solution, as it allows NRT indexing (Near Real Time). It is through this module and its compatibility with data extraction sources such as the Flume project, that the latest data entered in the data warehouse become searchable by the user.

The architecture includes a simple and intuitive search interface made by the Hue project. In Figure 3.8 it is reported the Hue-based search interface.

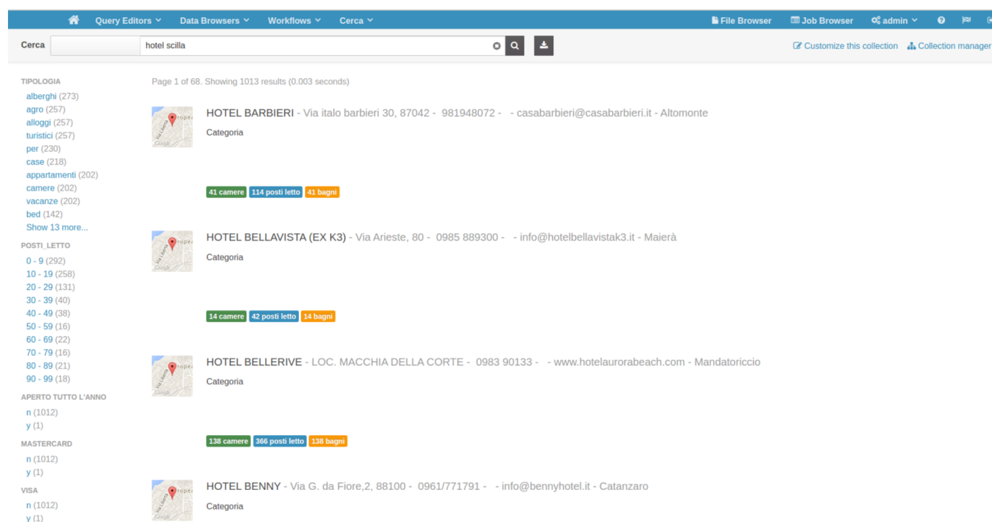


FIGURE 3.8: Hue-Based Search Interface

Data Indexing

Data indexing on Solr can be done in two distinct ways, in a static manner, usually when you need to work on a huge dataset available on a system, or dynamically when the system is in production and it is necessary to index new data coming from outside. In the first case HBase Map Reduce Indexer is used, while in the second case Lily HBase Indexer.

HBase Map Reduce Indexer is a Java library able to facilitate the installation of a Map Reduce job on a Hadoop cluster. The index generation/construction operation, once started, is performed in batch mode. As first operation, all tuples on HBase are converted into <key, value> pairs, using Map type tasks. Therefore in Reduce tasks *SolrInputDocument* objects are generated which constitute the storage unit on Solr. Figure 3.9 shows a diagram with the entire indexing process:

In Reduce phase it is possible to perform the duplication of existing index or merge them. However, if autocommit mode is configured on Solr, once the indexes have been created and loaded it is possible immediately to use them to perform searches.

To start this type of indexing it is necessary:

- Populate a HBase table

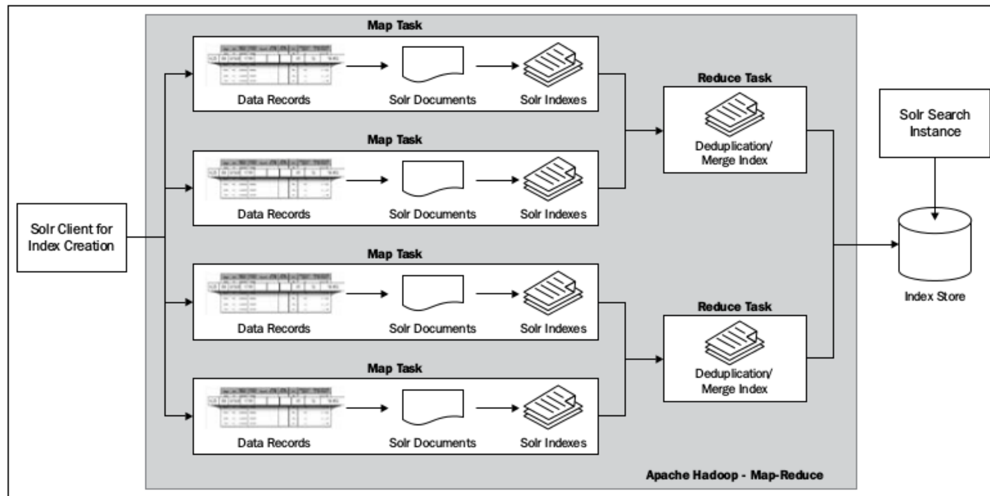


FIGURE 3.9: Solr Indexes Generation Starting From HBase

- Create a corresponding collection on SolrCloud
- Create a configuration file for Lily HBase Indexer
- Create a Morphline configuration file (for ETL)
- Run HBaseMapReduceIndexerTool

The advantages of this type of solution are that a high level of parallelism is reached in the creation of index through Map phase and that merging operations with pre-existing indexes are possible through reduce phase. The disadvantage lies in the fact that it is not possible to give priority to a given type of input data rather than to others because all the pairs <value key> are received with equal weight/importance in reduce phase

Lily HBase Indexer is an Open Source project distributed by NGDATA company. This system allows to integrate capabilities of Apache HBase and Solr projects, indexing data stored in the first system, in order to allow their use in real time. In practice, Lily processes data as soon as it is inserted on HBase using a sort of trigger. The entire operation takes place in a perfectly asynchronous manner, so it does not impact performance of HBase, but requires the enabling of data replication system. In practice, Lily indexer daemon works like a normal replication process for HBase. When a writing operations occur

on datastore, these are managed in a distributed way by HBase Region Servers through Zookeeper project. The data is then "replicated" asynchronously by several processes and indexer simply captures and analyzes change events generated by HBase, and in case of insert operations, the corresponding documents are created on Solr. These documents contains sufficient information to uniquely identify HBase row on which they are based, allowing Solr to be used to search for content stored in HBase. The performances of HBase are not affected, because Lily does not perform read operations directly on datastore, but refers to HBase log file, the same one on which Region Servers work. This register contains all information necessary for index, moreover since the events generated by HBase replication system are released in batches, the indexer is able to exploit them to avoid performing unnecessary operations, for example in the case where the same line is updated several times in a short space of time. This batch mode not only reduces the number of updates to Solr, but also offers important performance improvements. Lily represents a distributed system and offers a high level of horizontal scalability thanks to Zookeeper project, that is used for the configuration storage. Therefore new indexing processes can be added to the cluster at any time and automatically.

3.2.5 Open Source Tools For The System

In this section we describe some open source tools used in the proposed architecture. These are, mostly, projects that were born to be used in different contexts than Big Data, but have been extended later to adapt to this new paradigm (as Solr/Solrcloud) or have been integrated into our architecture so that they can work with huge amounts of data using parallelism or multiple sequential execution on partitioned data (as Carrot2 and Hue).

Apache Solr

Solr is a search engine used by many famous sites including CNet, SourceForge and Netflix; it is a project written in Java that uses HTTP protocol and XML for communication and offers several features such as highlighting results, spelling correction of queries, automatic suggestions for similar documents and most importantly the organization as facet of results. Solr uses

the Java and Open Source Apache Lucene [16] library to build and manage an "inverted indexes", a specialized structure for managing queries on text documents.

A relevant aspect that distinguishes a query on Lucene (hence on Solr) from a query on a relational database is the fact that results are sorted according to their relevance to the query itself, while on a relational database at most they can be ordered based on one or more columns of a table. The Figure 3.10 shows the structure of an inverted index, which considers offers of houses for sale as input documents.

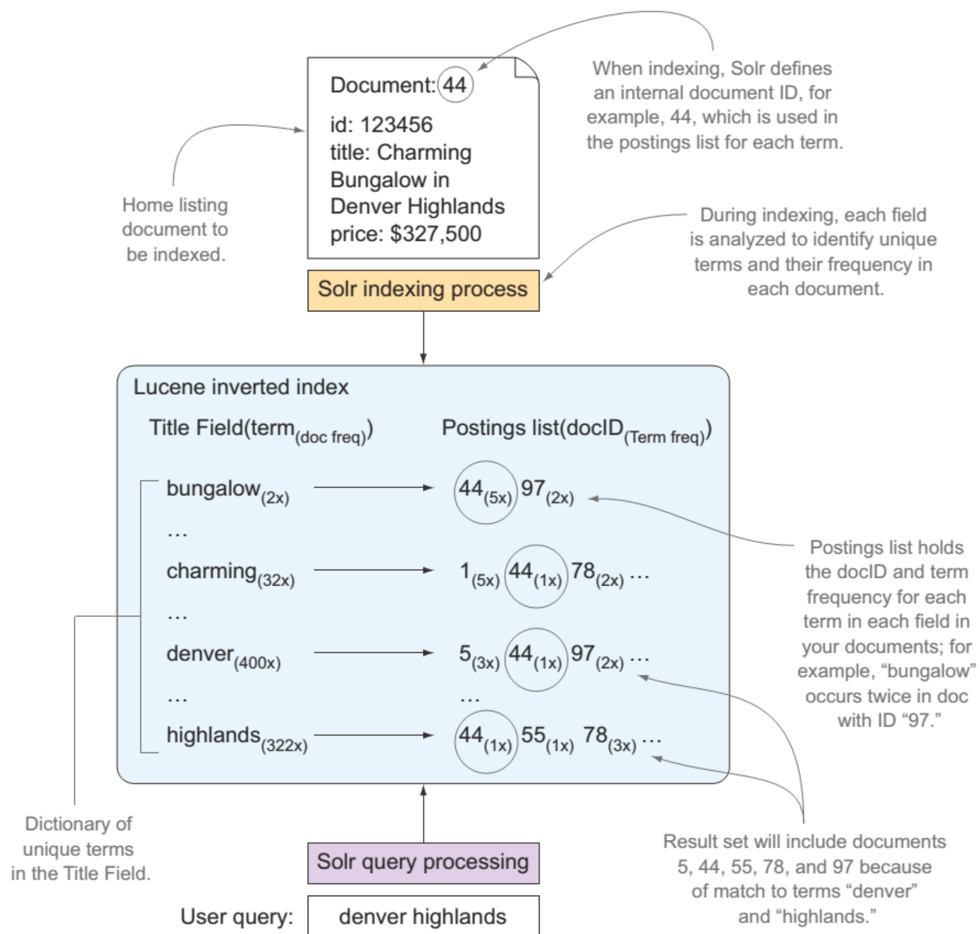


FIGURE 3.10: Inverted Index Example

[16] https://lucene.apache.org

The construction of these inverted indexes can take place in a distributed and simply way using Map Reduce framework (which is exactly what Google does). In this case the Map phase produces a unique term and ID pair of document in which the term was found, while in Reduce phase the list of documents in which the term is found is created. At this point Reduce will count the relative frequencies in documents for each term and construct the inverted index. Using Solr rather than Lucene directly leads to many advantages, including the ability to define the structure of indexes without writing lines of code but directly through XML files. Some of the most important features of Lucene are:

- Storing a reverse index for quick retrieval of documents by keywords
- Text analysis tools that transform a string of characters into a series of terms (words), which are the fundamental units for indexing and research
- A rich syntax of query language
- A flexible and customizable scoring algorithm for sorting results based on relevance.
- A spell checker for queries based on indexed data

Lucene itself is not a server and cannot communicate via XML. Solr can be seen as the implementation in form of server of this library, although additional features are introduced, such as the following:

- Indexing and querying via HTTP and XML protocol: communication with Solr occurs by sending data to the servlet URL through the POST method of http protocol
- Multiple cache memories to get results faster
- Administration interface accessible via browser that includes statistics on the use of the cache, an index query form, a data schema browser and details of the scoring algorithms and text analysis.
- Configuration file for the schema and for the server itself (in XML).
- Possibility to scale the system by distributing it to several machines.

Solr is implemented using servlets and requires a server for web applications that implement the Oracle Java Servlet specifications, such as Apache Tomcat or Jetty.

Solr allows to create a multidimensional search system, in which the fields of diagram constitute the facets (precisely the dimensions). Thanks to counting functions and ability to filter queries in a sequential manner, Solr is able to provide a counter for each indexed term, thus detecting the total number of documents that contain it and guiding navigation to subset of results more interesting to the user. Solr generates these counters together with the results of the text search and therefore both information is returned following a single query; in SQL, a similar result might require a series of separate queries.

SolrCloud

Although Solr was born as a full-text search engine in order to research and evaluate documents, over time the project has evolved to the point of managing much more. Many organizations already use this evolution, known as SolrCloud, as a classification engine, recommendation system or even directly as a NoSql datastore, but one of the areas in which it has shown to have many potential is precisely Big Data analysis. SolrCloud is optimized for a specific class of problems, full-text search on large amounts of data and the relative ordering of results based on relevance. The properties that distinguish it are the following:

- Scalability: indexing and querying operations are distributed across multiple servers in a cluster
- Search Optimization: it is a system able to give answers to complex queries in a short time (tens of milliseconds)
- Management of large volumes of documents: the system is designed to manage indexes with many millions of documents
- Optimized system for searching documents written in natural language (e.g. emails, pdf, etc.)
- Sort results by relevance

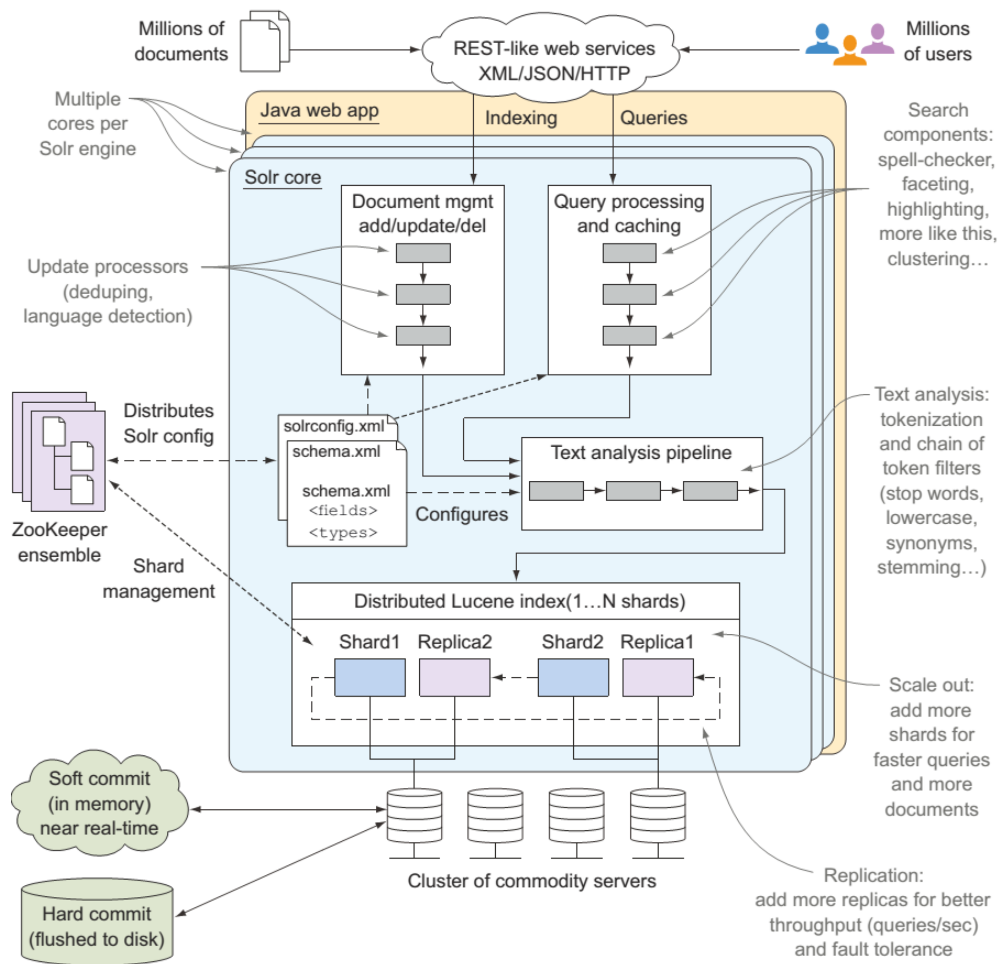


FIGURE 3.11: SolrCloud Architecture

Figure 3.11 shows the internal architecture of Solr Cloud.

As you can see from Figure 3.11, SolrCloud allows to run multiple different indexes (cores) on a server, each with its own configuration files and its own space dedicated to data storage. Furthermore, a single core can be segmented and replicated on multiple nodes of a cluster in order to distribute the index and make it fault tolerant able to handle multiple requests simultaneously. In the last version of SolrCloud these aspects are managed through Zookeeper project (component on the left of figure), which deals both with distributing the cluster configuration between the various nodes and with continuously monitoring the state of the cluster.

Carrot2

Carrot2 is a library, with a series of support applications, that can be used to build a search engine with clustered results. This engine organizes search results in topic in a totally automatic way without external knowledge such as taxonometers or pre-classified contents. The project is released with two proprietary algorithms, both created specifically for the clustering of search results: Suffix Tree Clustering (Zamir and Etzioni, 1998) and Lingo (Osinski and Weiss, 2005) as well as a version of k-means. Furthermore, there are components for retrieving documents from search engines, such as Bing or Google, and data sources such as Lucene and Solr are supported. In the latter case the Carrot2 library is used as a search component within a pipeline, so the result is processed by various Solr modules.

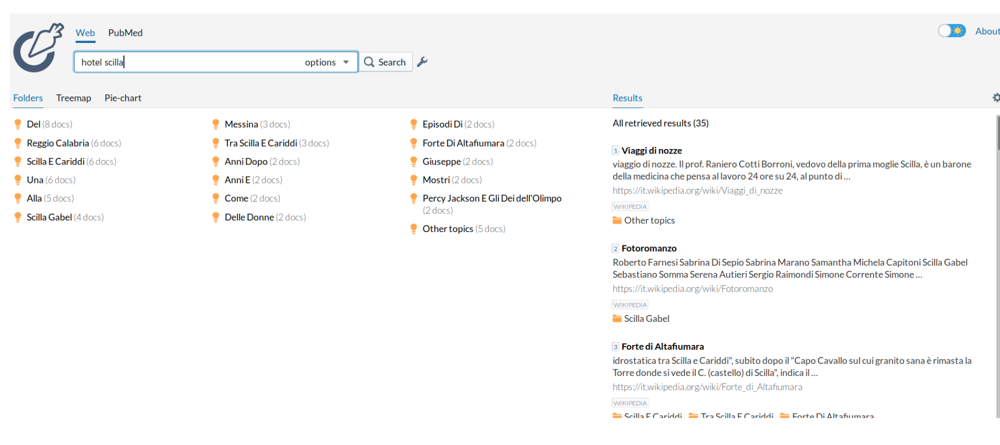


FIGURE 3.12: Carrot2 Web Interface

Carrot2 is distributed in different forms. It is possible to directly use the tools in standalone version, the API to quickly get results from Java and C# programs or delegate all operations to a remote service. Carrot2 distribution contains the following elements:

- **Carrot2 Document Clustering Workbench** which is a standalone GUI application you can use to experiment with Carrot2 clustering on data from common search engines or your own data.

- **Carrot2 Document Clustering Workbench** which is a standalone GUI application you can use to experiment with Carrot2 clustering on data from common search engines or your own data.
- **Carrot2 Java API** for calling Carrot2 document clustering from your Java code.
- **Carrot2 C# API** for calling Carrot2 document clustering from your C# or .NET code.
- **Carrot2 Document Clustering Server** which exposes Carrot2 clustering as a REST service.
- **Carrot2 Command Line Interface** applications which allow invoking Carrot2 clustering from command line.
- **Carrot2 Web Application** which exposes Carrot2 clustering as a web application for end users.

Clustering in Carrot2 is based on a pipeline of components. The two main types of components are Document Sources and Clustering Algorithm. The first provide data processing functions. In a typical scenario this component could retrieve search results from external search engine or from a Lucene/Solr index or an XML file. Clustering algorithms allow to organize documents provided in significant groups. Algorithms available in Carrot2 are Lingo, STC and a version of k-means.

Hue

Hue is a easy to use web application that facilitates the interaction of the end user with the Hadoop ecosystem. This system allows to avoid the use of command line for interactions with the Hadoop ecosystem services.

The main features of this tool are following:

- HDFS File browser
- Graphic query editor for Hive
- HBase Browser to view and edit HBase Tables

- Metastore Browser to view and edit Hive metadata
- Job Browser to monitor Map Reduce jobs
- ZooKeeper Monitor

FIGURE 3.13: Hue Interface

3.3 OLAP Based Analysis Architecture

In this section an OLAP tools designed for supporting the analysis of Big Data gathered from several sources will be described. In particular, an end-to-end framework will be presented which assist decision makers in all phases of data warehousing from the pre-elaboration process (that could reveal really hard for heterogeneous sources) till the analysis steps.

3.3.1 The Pentaho Data Warehouse Analysis Tools

Due to Big Data features, a system to manage a data warehouse need to take care of several requirements. In order to guarantee the maximum flexibility

to a system, it is possible to exploit the Pentaho BI Suite [17]. This is a suite that provides services of data integration, OLAP, reporting, dashboard, data mining and ETL. The Pentaho suite offers two products: the Community Edition (CE) and the Enterprise Edition (EE). The CE is the open source solution offered by this suite and, even if it is a free version, it contains a series of products that offer a valid alternative to other commercial BI solutions. The EE version provides additional components and programs that make the product more powerful and more competitive even for medium-large companies. The enterprise version is obtained through an annual subscription which also includes additional assistance services. The amount of the license varies according to the services requested by the individual companies, therefore it is difficult to estimate. In this work we will refer only to the open source CE version.

Pentaho Data Integration

A crucial activity for data warehousing is the Extraction, Transformation and Loading (ETL) of raw data being collected from several sources. In this respect, Pentaho provides a suite for performing this task, namely Pentaho Data Integration (PDI) (usually referred as Kettle - Kettle Extraction Transforming Transportation Loading Environment-). It is a powerful tool with a metadata-oriented graphic development environment as shown in Figure 3.14. Thanks to this tool it is possible to cross data from multiple sources, update them in real time, migrate data between different systems and so on.

PDI provides the following tools:

- ***Spoon.*** It is the graphical user interface for editing the options provided by PDI. The main features provided concern the possibility of extracting and storing data from a large number of sources (databases, spreadsheets, text files, etc.), data manipulation with the possibility of using tasks with predefined functions already present in the environment and execute tasks defined by the user through Java and JavaScript code. This tool allows to aggregate data and store it in the Data Warehouse.

[17] https://www.hitachivantara.com/en-us/products/data-management-analytics/pentaho-platform.html

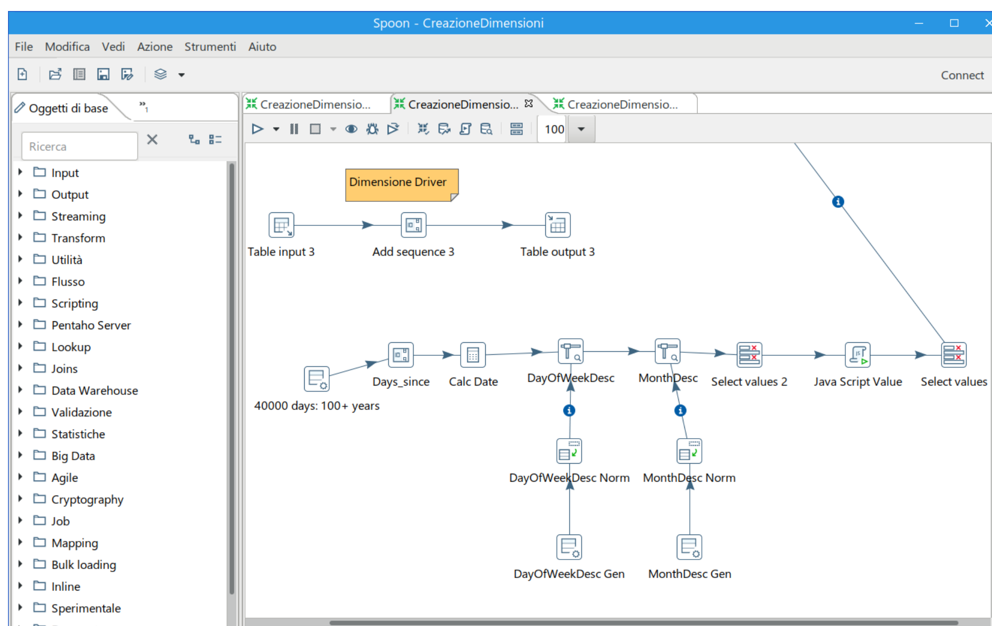


FIGURE 3.14: Pentaho Data Integration Interface

- ***Pan.*** It is a tool that allows to perform the transformations designed with PDI. Usually the transformations are scheduled in batch mode to be performed automatically at regular intervals.
- ***Kitchen.*** It allows to execute the jobs defined with Spoon from command line.
- ***Carte.*** It is an HTTP server for remote execution of transformations and jobs. It runs in clusters with other active instances in order to distribute the execution load.

Pentaho Business Analytics Platform

Commonly referred to as the BI Platform, and recently renamed Business Analytics Platform (BA Platform), it makes up the core software piece that hosts content created both in the server itself through plug-ins or files published to the server from the desktop applications. It includes features for managing security, running reports, displaying dashboards, report bursting, scripted business rules, OLAP analysis and scheduling out of the box. The Pentaho BA

Platform runs in the Apache Tomcat Java Application Server, so it provides a useful web interface as shown in Figure 3.15 and It can be embedded into other Java Application Servers. This platform contains and is based on *Mondrian*

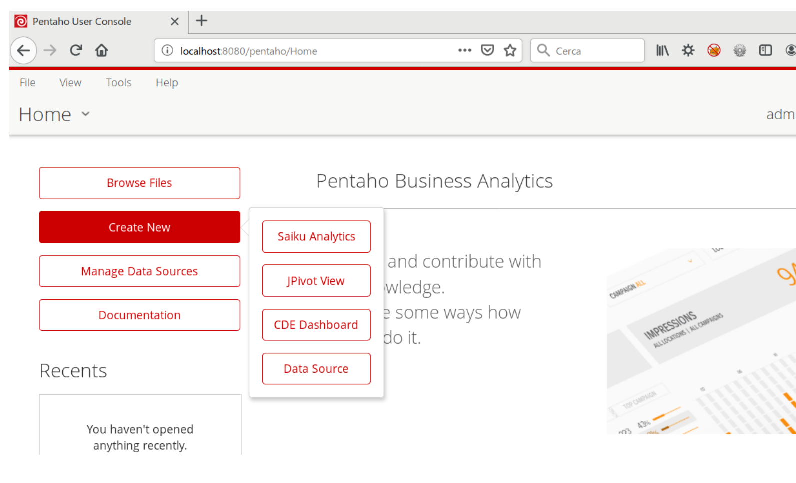


FIGURE 3.15: Pentaho BA Server Interface

server which is an open-source OLAP (online analytical processing) server, written in Java. It supports the MDX (multidimensional expressions) query language and the XML for Analysis and olap4j interface specifications. It is based on ROLAP technology, thus it translates MDX queries to SQL based on a user defined XML multidimensional model. It accesses information stored in the data repository and perform aggregation operations that are then cached. It also makes extensive use of materialized views to optimize the speed of response. Mondrian can be run separately from the Pentaho BA Platform, but is always bundled with the platform itself in both EE and CE versions.

Schema Workbench

Schema Workbench (SW) is a graphical tool for data cubes creation. The schema file is generated using a graphical user interface. Schema Workbench produces as output an XML file containing the definition of the cube structure for OLAP analysis that will be performed by Mondrian. It is not necessary to use Schema Workbench to create the schema, but it is often helpful for beginners and even experts who need go inspect a cube visually and come up to speed with how

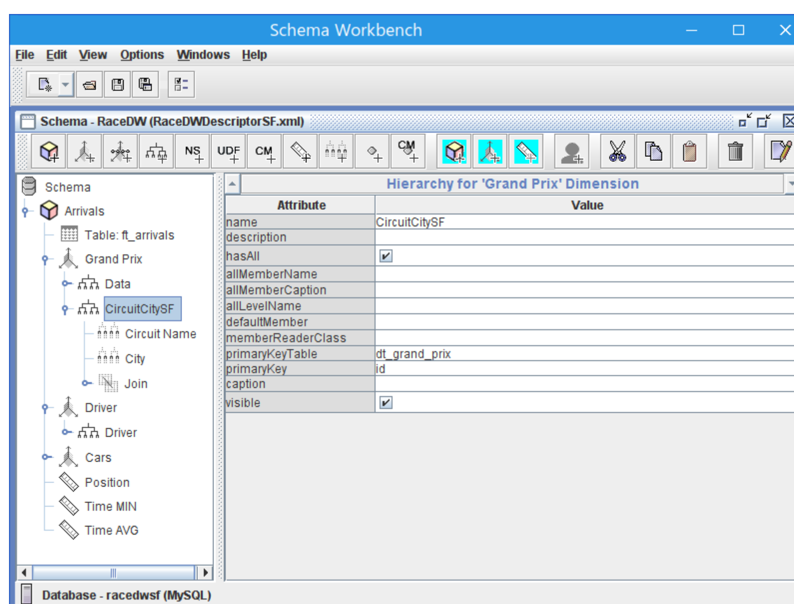


FIGURE 3.16: Schema Workbench Interface

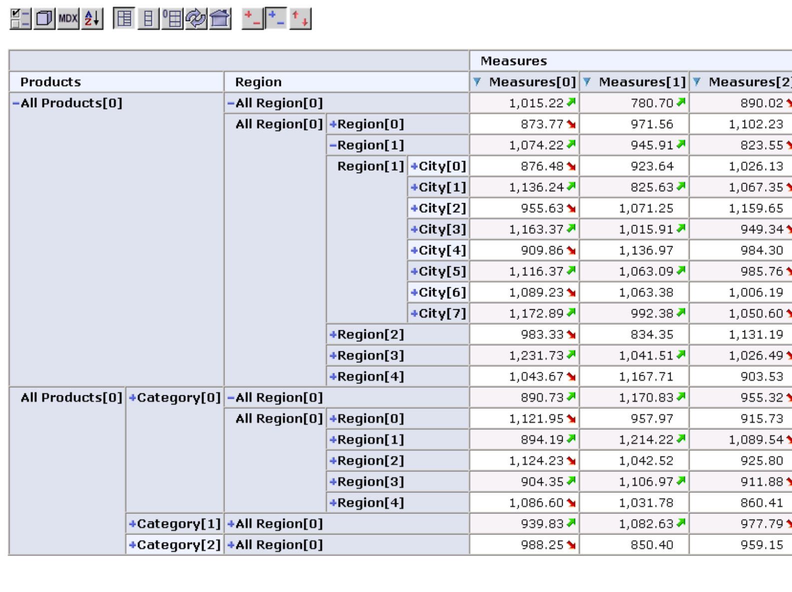
to maintain or extend it. Schema Workbench provides an integrated environment in order to validate the specified schema based on the source data specified during configuration. Figure 3.16 shows the interface of this tool.

Display Tools

By default Pentaho BA Server provides JPivot tools to display data but it is possible to extends the platform downloading other GUI in form of plugins. A better tool for display BA data is Saiku, a plugin that offers several advantages with respect to JPivot as it allows simple drag and drop and deals with HTML5 and CSS in a flexible way.

JPivot

JPivot is an open source Java library, therefore independent of the Penthao platform, consisting of custom JSP tags, i.e. it allows to display an OLAP table and to execute on it the typical operations of navigation, such as slice/dice, drill-down and roll-up. JPivot uses Mondrian as an OLAP engine in this



Products		Region			Measures[0]	Measures[1]	Measures[2]
-All Products[0]		-All Region[0]			1,015.22	780.70	890.02
		All Region[0]	+Region[0]		873.77	971.56	1,102.23
			-Region[1]		1,074.22	945.91	823.55
			Region[1]	+City[0]	876.48	923.64	1,026.13
				+City[1]	1,136.24	825.63	1,067.35
				+City[2]	955.63	1,071.25	1,159.65
				+City[3]	1,163.37	1,015.91	949.34
				+City[4]	909.86	1,136.97	984.30
				+City[5]	1,116.37	1,063.09	985.76
				+City[6]	1,089.23	1,063.38	1,006.19
				+City[7]	1,172.89	992.38	1,050.60
			+Region[2]		983.33	834.35	1,131.19
			+Region[3]		1,231.73	1,041.51	1,026.49
			+Region[4]		1,043.67	1,167.71	903.53
All Products[0]	+Category[0]	-All Region[0]			890.73	1,170.83	955.32
		All Region[0]	+Region[0]		1,121.95	957.97	915.73
			+Region[1]		894.19	1,214.22	1,089.54
			+Region[2]		1,124.23	1,042.52	925.80
			+Region[3]		904.35	1,106.97	911.88
			+Region[4]		1,086.60	1,031.78	860.41
	+Category[1]	+All Region[0]			939.83	1,082.63	977.79
	+Category[2]	+All Region[0]			988.25	850.40	959.15

FIGURE 3.17: JPivot Interface

framework, but it can also interact with XMLA data sources (XML for Analysis), it is based on the WCF library (Web Component Framework) for rendering the graphic objects of the user interface and on the well-known JFreeChart package for tracking graphics. This software represents the graphic component for browsing the datacubes, and in addition to converting the result of MDX queries into an HTML graphic format that can be understood by the user, it allows users simple navigation. In Figure 3.17 it is possible to see the format of a classic data navigation table built by JPivot.

Saiku

Saiku represents a better alternative to JPivot, it is an independent open source project that offers the possibility of performing OLAP analysis by connecting to an OLAP server like Mondrian. It is provided in the form of plugin that can be easily integrated into the Pentaho platform and available through the market of the Suite. Unlike JPivot, the graphical interface for executing queries is based on the most intuitive drag and drop mode. The plugin uses HTML 5, Javascript and CSS.

This plug-in also allows the direct creation of reports from the results of MDX queries that can be exported to the most common graphic file formats.

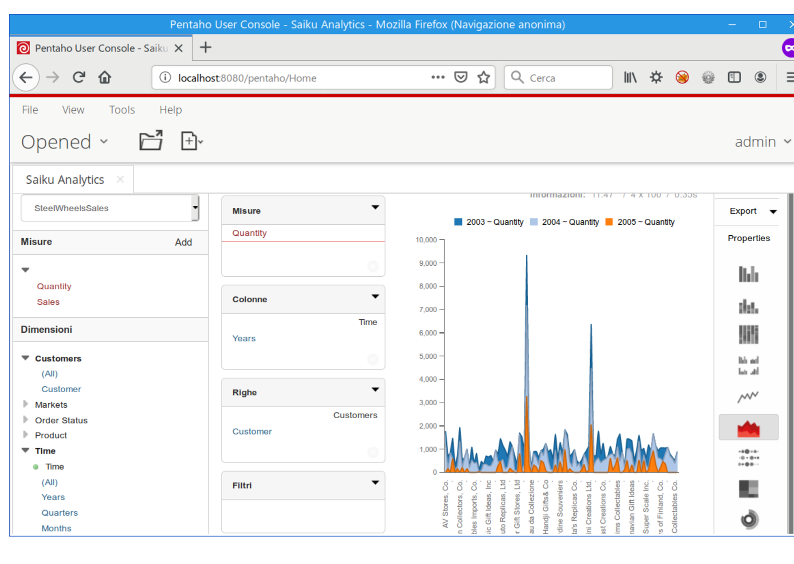


FIGURE 3.18: Saiku Interface

It also allows the export of results in different formats such as Excel, PDF or CSV. Figure 3.18 shows the Saiku Web Interface.

By harnessing the power of OLAP, Saiku allows users to choose the measures and dimensions they need to analyse and "slice and dice" the data and drill into the detail to uncover relationships, opportunities and issues. The intuitive user interface lets users drill down and up, filter, pivot, sort, and chart against OLAP and In-Memory engines.

Pentaho Analysis Operation

The core system of Pentaho Suite consists of the Mondrian Server. This is a component capable of transforming MDX queries into SQL format using a Data Warehouse descriptor XML file for mapping. This file, called cube descriptor usually is generated via Schema Workbench tool. Schema Workbench allows the writing of this file in a guided way, allowing connection to the Date Warehouse for which the descriptor must be written. At a logical level it is possible to define indifferently a star or snowflake schema inside this descriptor. The making of MDX queries usually occurs through visual procedures on client software like JPivot or Saiku that simplify this step using drag-and-drop

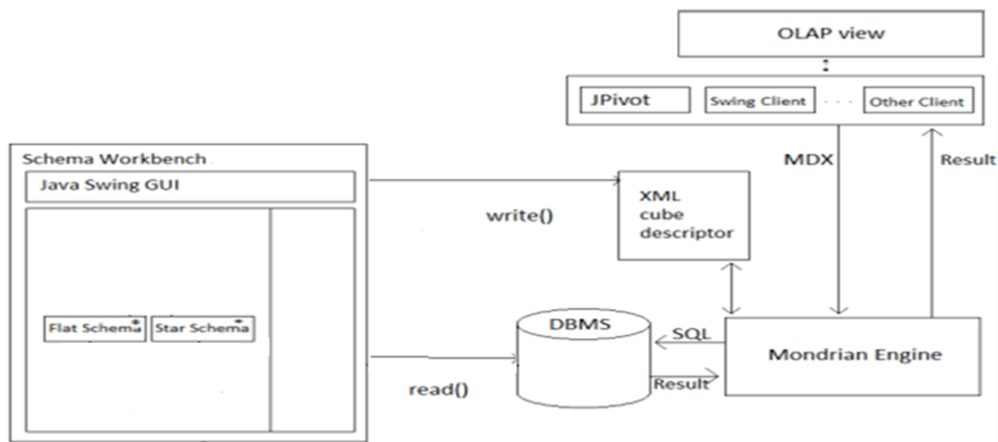


FIGURE 3.19: Pentaho System Interactions

or selection procedures. Once an MDX query is created, it is sent to the Mondrian server by a web client. The Mondrian server transforms the MDX query into SQL using the XML descriptor and forwards it to the SQL server. At this point the SQL server executes the query and returns the result in tabular form to the Mondrian server which in turn forwards it to the web client which will show the results to the user. Figure 3.19 shows this execution cycle.

3.3.2 Big Data OLAP System Architecture

This section presents a Big Data analysis system that uses Pentaho Server as Business Intelligence suite and Hive tool (see section 2.7.4) for distributed access to data contained in the Data Warehouse. As storage layer, this system exploits a NoSQL solution based on HBase, which allows to obtain in a natural way greater efficiency on reading operations compared to standard solutions. The Pentaho suite allows access to the information stored in the data archive and to perform data aggregation operations, whose results for efficiency reasons are then stored in cache.

Figure 3.20 shows the system architecture, which is composed of three macro modules:

1. Mondrian, as OLAP Server

2. Hive, as Query Executor on Hadoop Map Reduce

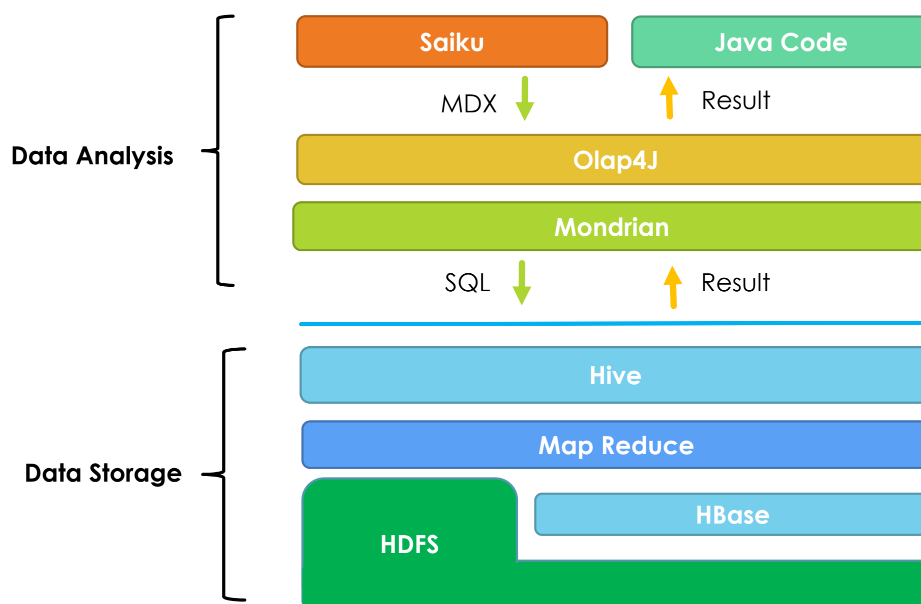


FIGURE 3.20: OLAP Analysis Architecture for Big Data

3. HBase, as NoSQL Data Storage

This architecture combine Mondrian Server and Hive in order to guarantee the distributed processing of queries on different nodes of a server cluster. In this way it is possible to obtain various advantages, including the possibility of exploiting SQL as a query language, towards a sub-system that does not support it natively. The combined use of HBase and Hive, allows to overcome some limitations of Hive like slowness of data access and interrogation. In fact, among the main features of HBase, we have vertical partitioning in Column Families, horizontal partitioning in Regions, replication and indexing, (as shown in section 2.7.2) all of which should ensure Near Real Time access to data.

The analysis process is the following:

- The user specifies a query on predefined multidimensional cube through a graphical interface (JPivot or Saiku)

- The interface module creates the MDX expression associated with the user query
- Mondrian translates MDX into SQL, as long as the query result is not already in cache
- Hive create a Map Reduce type job, which accesses data stored on distributed datastore, using a generated map from SQL commands to Hadoop ecosystem
- The final result is shown to the user, going back through previous phases

Figure 3.21 shows the user interactions with the system. It provides two types of users: "Basic User" and "Domain Expert". The users of type "Basic User" interacts with the system by generating reports and applying the classic OLAP operators using GUI client like Saiku or JPivot. They cannot define any new cube or measure but only can use those already present in the system. The users of type "Domain Expert" can use all the features of the "Basic User" and in addition they can define new cubes or measures using Schema Workbench.

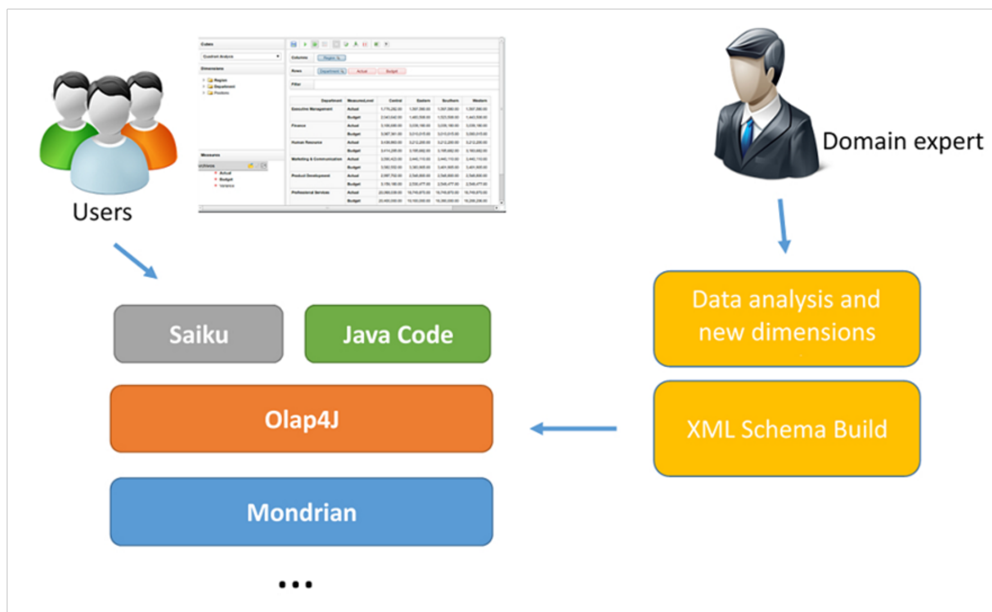


FIGURE 3.21: System User Interactions

Chapter 4

High Performance Computing On Peer to Peer Network

4.1 Background

The advances in computer science allow many problems to be solved in a quite effective way both in terms of computational time and resource usage. However, several problems still require an amount of computational resources that goes far beyond the power of a single device or a single user. In order to solve these kind of problems, a new computing paradigm has born: *crowdsourcing*. This new paradigm is based on the idea of gathering the resources needed to complete a task from the crowd in order to parallelize its execution.

Many attempts have been made to properly define the key features of crowdsourcing systems; however, the answer to this apparently trivial question is not straightforward, since there exist many different crowdsourcing system based on different models and assumptions. If we try to find the common features shared among all the successful crowdsourcing systems (e.g., Wikipedia, Yahoo! Answers, Amazon Mechanical Turk) we can clearly realize that they rely on some assumptions:

- They should be able to involve project contributors
- Each contributor should solve a specific task
- It is mandatory to effectively evaluate single contributions
- They should properly react to possible misconducts

As we can see from the examples above many systems are based on an explicit collaboration of many different people sharing the will to build a long-lasting product that can be used by the whole community. Aside from this human-centric view, there exist a plethora of systems that leverage implicit co-operation among users (e.g. multiplayer games) or tools that are not devoted to the production of a tangible object (e.g., Amazon Mechanical Turk).

A well known open source framework that is widely used (mainly) for scientific purposes is BOINC (Berkeley Open Infrastructure for Network Computing) (*BOINC website*). It allows volunteers to contribute to a wide variety of projects. Their contribution is rewarded by credits used to climb a leaderboard. In recent years a new category of collaborative approaches for cryptocurrency mining is emerging, such as Bitcoin (Nakamoto, 2008) implemented by blockchain technology. Users aiming at mining new Bitcoins contribute in solving a decoding task and are rewarded with a portion of the gathered money, proportional to the effort put in the mining task. The latter approach is gaining a lot of attention from the end users just because it allows them to earn a tangible reward.

In this chapter a system named *Coremuniti*^{TM} (that stands for *Community of Cores*) is presented. This system is inspired by the collaborative model used in BOINC while implementing an ad hoc rewarding strategy similar to Bitcoin mining. The system does not require in principle any specific user skills but users can join the network simply providing their under used computational resources. Therefore the *Coremuniti* approach can be seen as an hybrid crowd as tasks can be solved by computer-based resources(Peng, 2015).

More in detail, the novelty of this innovative project is the design of a peer to peer framework able to provide services at much lower prices compared to centralized center farms, by exploiting idle computational resources from the users joining the network.

This system represents an alternative to Map Reduce based systems (like Google Map Reduce or Hadoop) described in 2.7.3 or similar systems like Spark described in 2.7.5. However the basic approach is different: while our system was thought to operate in a extra-cluster scenario (with a lot of different devices located around the world), the map reduce systems were designed to work usually in the same cluster of the same organization.

These differences make it necessary to work on some additional issues including:

- **Correctness of the results**. Since the system involves different users, these could return incomplete or incorrect results to maximize their credits.
- **Data and results movement**. Working on one or more clusters of the same organization allows to obtain network speeds that are much higher than peers that are located around the world and with different connection speeds.
- **Security of communications**. Extra-cluster communication need to be secured to avoid that can be intercepted and manipulated by third parties.

The Coremuniti system can be used in several application scenarios, e.g. computer simulation and advanced data analysis and it is well suited for vertical implementation of computing intensive tasks, representing a trans-disciplinary opportunity. More specifically, this approach can be a valid alternative to traditional solutions, such as buying or renting expensive dedicated servers. Furthermore, in many cases, even using powerful dedicated servers, the time needed to solve a problem is still too high, because the subtasks composing the problem are not parallelized at all. Coremuniti approach is based on an high performance Peer to Peer (P2P) network composed by computational resources shared by the users of the network itself. Each node of the network (i.e. users in the crowd) can set the amount of their resources to share. When a peer needs to execute a high resource consuming task, he can ask the necessary computational power to the network. The process can be executed in few clicks thus making this software quite user friendly. In order to assess the effectiveness of this solution, a the 3D rendering scenario has been analyzed that turns to be a severe test bench for this technology. A specialized plugin named *Mozaiko*^{TM} (this name was chosen as basic approach splits a complex task in several sub-tasks that will be re-assembled like mosaic tiles) has been developed to allows to render Blender 3D models on this distributed network. The rendering process typically engages user's computers for a considerable amount of time. The experimental analysis shows that existing solutions are slower and more expensive than the proposed approach.

Moreover, this system does not require to frequently purchase new hardware, since users continuously provide (up to date) computational power. Finally, the better re-use of already powered resources could induce a beneficial systemic effect by reducing the overall energy consumption for complex tasks execution.

4.1.1 Coremuniti in a nutshell

P2P networks feature a common goal: the resources of many users and computers can be used in a collaborative way in order to significantly increase the computing power available for the users and parallelizing task executions. In "full" P2P networks, each computer communicates directly to each other thus allowing better bandwidth use. However, in many cases, there are some inherent drawbacks to P2P solutions and some functionality needs to be centralized. Those systems, that can be used both for data sharing(Yang and Yang, 2010) and distributed computation(Yang and Garcia-Molina, 2001), are denoted as "hybrid" P2P. Coremuniti falls in the latter category and aims to build a P2P network where users can share their unexploited computing resources. Figure 4.1 shows a possible usage scenario for the platform when using Mozaiko plugin. However, it must be said that this platform is general purpose, thus can be used for solving any complex problem that is parallelizable.

In order to join this network a user has to download the *Coremuniti Server* (platform independent) software. By running this software the user becomes a node server of network, denoted in Figure 4.1 as NSA (*Node Server Agent*). This software does not interfere with other applications running on the computer and the user can easily set the amount of CPU that wants to share with the network, so that Coremuniti Server can be easily adapted to everyone's needs. On the opposite side, users who need additional computational power, in order to complete computing intensive tasks for example, can install the specific software that is denoted in Fig. 4.1 as NCA (*Node Client Agent*) (i.e., *Mozaiko* for the case study). To start a new task, they simply issue a request to the network in order to gather the required resources. The submission of a new task in the network will cost to the user a number of credits proportional to the complexity of the task itself (see 4.3). Since each node of the network can act both as a server and as a client, when submitting a new task two cases may occur:

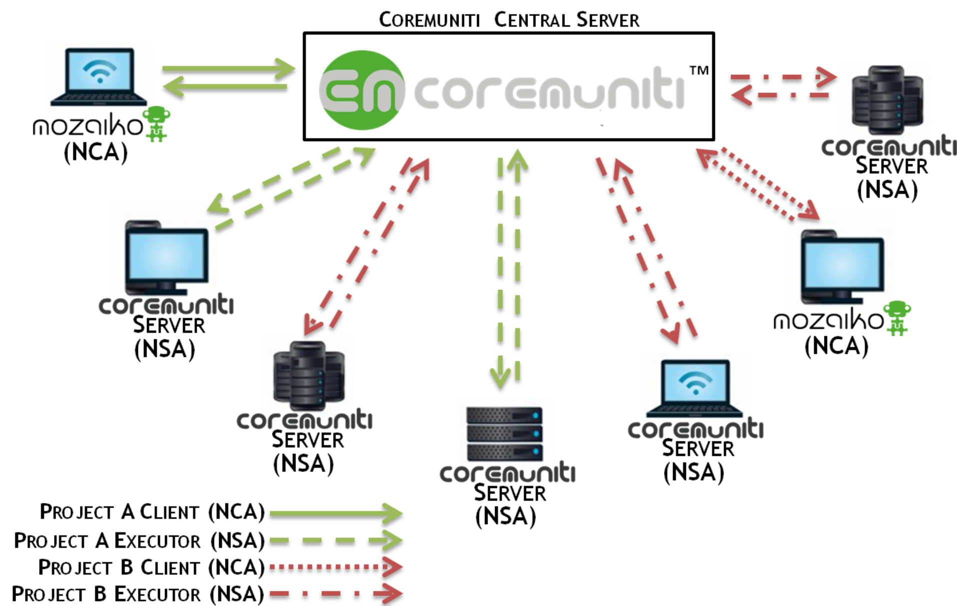


FIGURE 4.1: Coremuniti System at Work

1. the user has previously earned (a portion of) the required credits for running the task (e.g., because they acted as servers)

2. they bought the required credits

In order to guarantee a high level of service, the central server is responsible of performing the subtask assignment. More in detail, to fully take advantage from the capability of this network, the (possibly huge) initial task is partitioned in an adequate number of (much smaller) subtasks that can be quickly executed by the server peers. Moreover, an internal company network of 80 peers was created to be used when the number of available public peers is not sufficient to guarantee proper execution of user tasks.

As soon as subtasks are completed, the system checks their correctness and it rewards the participating peers. The model for subtask assignment, that will be described in 4.3, guarantees efficient execution for clients and gave to all peers (even if they have limited computational power) the possibility to be rewarded. Interesting enough, even users that are not going to ask for task execution have the chance to earn credits that can be redeemed by coins or gadgets. The latter feature makes Coremuniti a more convenient choice w.r.t.

other collaborative systems such a cryptocurrency mining (as shown in the experimental evaluation paragraph).

Main Contribution. To summarize, the major contributions of this work are the following:

- An hybrid P2P infrastructure named Coremuniti has been designed and implemented which allows collaboration among users by sharing unexploited computational resources. In particular, by running this software, users can join the Coremuniti network and they can either provide or request computational resources;
- A robust model for task partitioning, assignment and rewarding to network users has been designed. More in detail, users that are available for task execution are assigned with a suitable set of (sub-)tasks. When the execution is completed, the system reward users with credits that can be later redeemed for asking computational resources, coins or gifts;
- To prove the effectiveness of the proposed approach, several experiments have been performed on a real 3D rendering scenario. To this end the system performances have been measured against those of commercial popular 3D rendering farms and cloud services. Moreover, the system performances have been compared w.r.t. other systems that reward users for their effort in solving computational expensive tasks.

4.2 Coremuniti Architecture

The goal of Coremuniti Network is to build a reliable infrastructure that allows to share computational resources in an easy and secure way. This framework has to be robust against attacks from malicious users, analogously to every distributed computing systems (Bhatia, 2013) or distributed storage systems (Firdhous, 2012). More in detail, the system need to guarantee secure communication between clients and server, trusted software for remote execution and privacy for the intermediate computation. To this end, the system rely on an hybrid P2P framework (Franklin et al., 2011) where some functionality is still centralized, since it aims at guaranteeing continuous service availability.

Figure 4.2 shows the interactions among system components. Herein: *Client Node* refers to those users who want to execute a high computing demanding

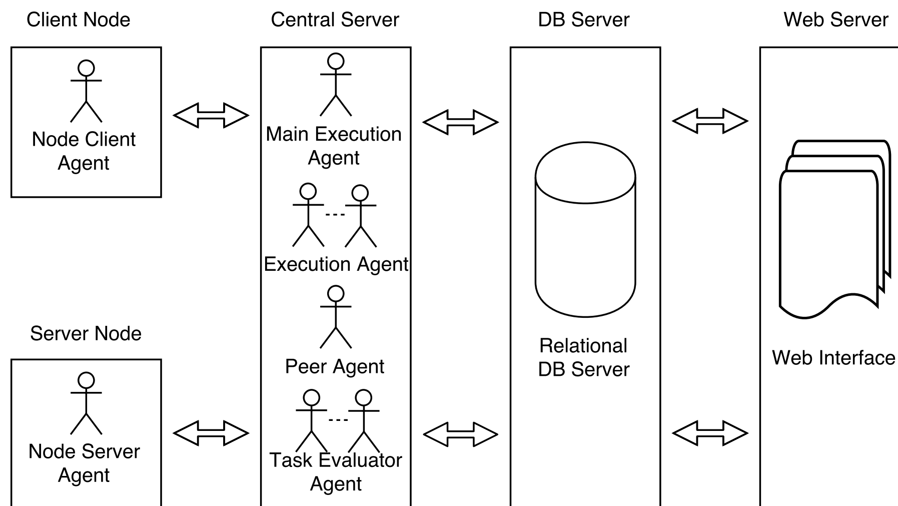


FIGURE 4.2: System Architecture

task (also referred to as *project* in the following) using Coremuniti network. *Server Node* refers to those users who will to share with other Coremuniti network users the computational resources that are not fully employed on their devices. Finally, *Central Server* is the core of the system. It is in charge of the allocation of the nodes, the displacement of the messages and the scheduling of tasks issued from client nodes that have to be distributed among server nodes.

The system is general purpose, every specific case study is implemented as a new dedicated plugin. Thanks to the use of software agents the system is able to be independent from low level processes and/or threads that are specific to the task being executed on the network. Figure 4.2 depicts the overall system architecture where we can see how for each component several agents are deployed.

Aside from the Client and Server Nodes, two new components complete the architecture: the DB Server and the Web Server. The first consists of a classical relational DB which stores information about the network status, the users and the devices. The web server, instead, allows users to interact with the system via web interface.

In what follows, the main components of Coremuniti architecture will be described.

Definition 1 (Client Node) *The machines of all users who want to submit new projects to the Coremuniti Network run a Client Node. It allows the execution of the* NodeClientAgent *(NCA) whose role is to create projects to be submitted.*

More in detail, a project is submitted to an agent residing on Central Server component. Thanks to the *NodeClientAgent*, users, after authentication, can perform the following actions:

- *Create and send new project to Central Server* that allows the agent to create a new project by specifying the execution data and parameters.
- *List all active project*
- *Stop and resume the remote execution of a project*
- *Delete a project*
- *Download partial updates*, i.e., a user can download the results of the computation even if the main task is still not completed. Thus, if the output of the project is too big, it can be downloaded while it is still running. This feature is very useful for Rendering Type tasks where the output can be very large (typically in the order of Gigabytes).

Definition 2 (Server Nodes) *Users willing to share their computing resources making them available to other users act as* resource providers *by running a Server Node.*

It allows the execution of the *NodeServerAgent* (NSA), whose job is to execute subtasks received from the Central Server. The agent is able to retrieve the technical features of the machine on which it is currently running and offers the opportunity to set the amount of computational resources to share with the network. Currently it is possible to share CPU or GPU power, RAM and disk space. The central server chooses the server nodes for a specific computation taking into account several aspects related to the complexity of the task and the availability of computational resources.

The *Central Server* component implements a clustered architecture and manages the execution of tasks submitted to the Coremuniti Network. This is the component that centralizes some core functionality of the peer to peer network and allows the concurrent execution of different types of agents.

The *MainExecutionAgent* (MEA) receives the remote execution requests performed by NodeClientAgent agents. For each request an *ExecutionAgent* (EA) is created, with the goal of handling the remote execution.

The primary role of MEA is to manage the load balancing of internal servers by properly assigning the EA to different machines.

ExecutionAgent takes care of managing remote execution of a single project. It performs the following jobs:

- *Subtask execution coordination for a single project.* This activity includes the initial assignment of subtasks to NSA (Algorithm 1 reported in Section 4.3). During execution of the subtasks the EA receives updates about the execution status and once the computation ends, it receives the final subtask results from each NSA. In addition to that, the EA periodically allocates new NSA to subtasks if the assigned NSA are slow (Algorithm 2 reported in Section 4.3).

- *Result Quality Control.* Specifically, a subtask is assigned to more than one resource provider. It is crucial for the EA to validate the obtained results. Since system protocol can be used in different scenarios by developing specialized plugins, the validation step is strongly tied to the process being executed. Generally, in order to perform the validation, different results are compared using a specific task evaluation metric. The comparison details can be implemented by using the *TaskValidation* API, that provides a fully customizable and easy way to manage different types of tasks in the Coremuniti network (as it will be shown for the rendering case study described in Section 4.4).

- *Credits Management.* When at least three NSA complete the execution of the assigned subtask *st* the EA assigns credits to all NSA working on *st* as described in Section 4.3.

- *Data Transfer Management (Upload/Download project/subtask data).* The system uses an internal file server that deals with data transfer among peers and central server.

The PA keeps track, using a database, of the information related to the execution of the tasks from the resource providers. It constantly monitors the

availability of resource providers that can be assigned to new tasks and updates a pool of available resources that can be requested by EA.

Since the primary goal of this framework is to minimize the overall completion time of a given problem, it is mandatory to estimate this total time in order to perform an effective division of the main task that guarantees good results in terms of execution time. The estimation of the overall completion time is performed by the *TaskEvaluatorAgent* (TEA). This component is also in charge of splitting the initial task into multiple subtasks with a similar expected duration.

Each TEA runs on a single machine (denoted as reference machine). The reference machine adopted in the current implementation has the following technical specifications:

- CPU: Single core - Single thread 4.0 GHz Intel Processor
- RAM: 32 Gb DDR3 1600 Mhz
- HDD: SATA SSD 1 Tb 550 Mb/s peak

4.2.1 Communication Protocols

This section describes the communication protocols between agents carrying out the main functions on the Coremuniti Network. Each project (task) Pj is a triple $\langle type, D, pms \rangle$, where *type* is the type of the task to be performed, D are the data on which the execution should be carried on and *pms* are the execution parameters. Each message M exchanged on the network has the form $\langle mtype, content \rangle$ where *mtype* is a string that defines the type of message and *content* is a generic object that specifies the content of the message.

In the following, the details of two communication protocols implemented in Coremuniti is reported in order to enable the communication between the Central Server and Client/Server nodes: *Adding New Project Protocol* and *Task Assignment and Execution Protocol*. These protocols are open specification in order to allow the developer community to eventually design new Client/Server node implementations.

Main agents involved in the submission of a new task to the system are:

- ***NodeClientAgent***: It creates the project P_j by choosing its *type*, sending D and setting the values of one or more *pms* for the selected project *type*;

- ***MainExecutionAgent***: It listens for remote execution requests, creates a new ExecutionAgent and deploys it to a selected machine to balance the system load;

- ***ExecutionAgent***: it manages the project evaluation interacting with the TaskEvaluatorAgent, the PeerAgent and the NodeServerAgents;

- ***TaskEvaluatorAgent***: Receives projects from ExecutionAgent and estimates the Pj total cost splitting it in subtasks of uniform cost.

In order to add new project, a NodeClientAgent sends to a MainExecutionAgent a message of type $RemoteExecutionRequest$. The MainExecutionAgent creates and deploys a new instance of ExecutionAgent trying to balance the system load. After the new ExecutionAgent is created, a new $ExecutionAgentAddress$ message is sent to NodeClientAgent with a $content$ containing the following information:

- Address and port of the ExecutionAgent
- Address and port of a data server
- Credentials for the data server

At this point the NodeClientAgent talks directly to ExecutionAgent that will handle both the creation and the remote execution of project. Each ExecutionAgent has an internal data Server in order to receive the data D. The user agent uploads D to ExecutionAgent via data server using credentials received with the previous message, (in the 3D rendering use case this data is the scene file for the project with $type$: *Rendering*). After upload is completed, NodeClientAgent can send a cost calculation request through a message with $mtype$ $CalculateCostExecution$ and with $content$ embedding the values of project parameters *pms*. In the example use case, the *pms* for *Rendering* type projects are the numbers of first and last frame to render. Obviously, the value of parameters can change overall total execution time and therefore the cost of project. For this reason, this message can be sent multiple times with different parameters in order to let user choose preferred parameters based on total project cost. When first message with $mtype$: $CalculateCostExecution$ comes is received by

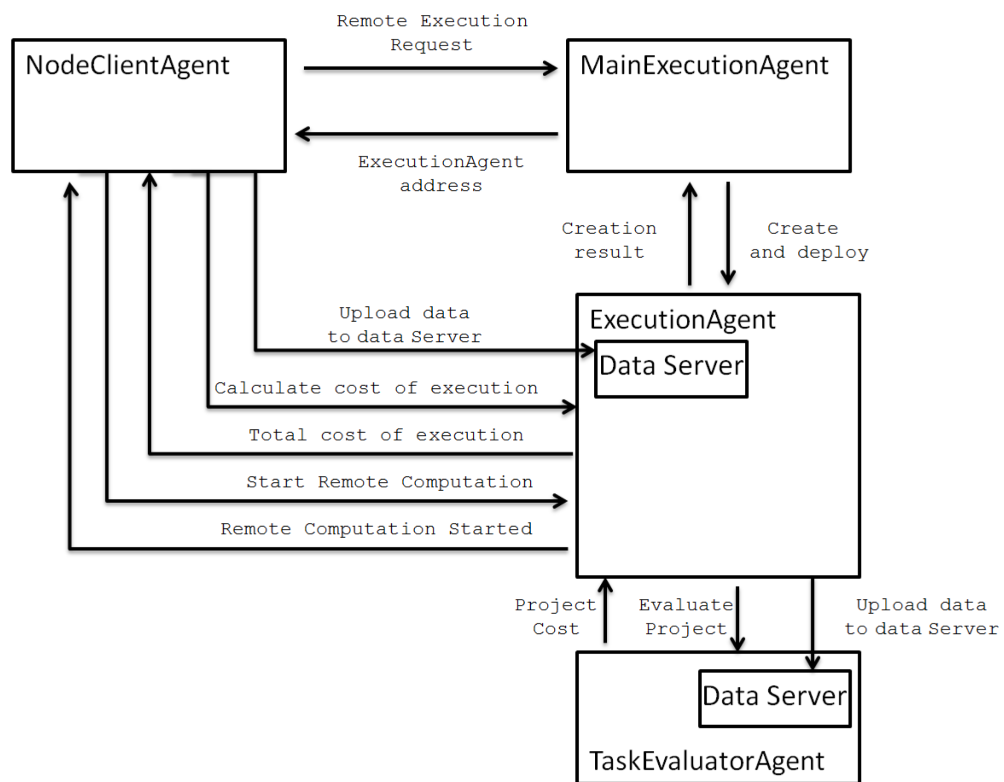


FIGURE 4.3: Adding New Project Protocol

ExecutionAgent, the D is uploaded to data server and *pms* are sent via message to TaskEvaluatorAgent to estimate the total cost of Pj. After the evaluation is completed a new message is sent from TaskEvaluatorAgent to ExecutionAgent with the total cost and number of subtasks of project. It is important to underline that an ExecutionAgent executes only one Pj, therefore a triple $\langle ExecutionAgent, Pj, pms \rangle$ uniquely identifies a request of Pj evaluation. As soon as ExecutionAgent receives the message containing total execution cost, it sends a new message to NodeClientAgent notifying total cost for the Pj with selected *pms*. At this point NodeClientAgent can start remote execution sending a new message to ExecutionAgent with *mtype StartRemoteComputation* and *content* the *pms* selected. If user on NodeClientAgent has enough credits she can request the task execution: the ExecutionAgent writes on DB Server the project data and remote computation can start; otherwise an error message is returned to NodeClientAgent. In order to save resources on the machines where ExecutionAgents are deployed a session timeout is set. If data D is not sent before timeout, the agent is closed and resource released. Furthermore, after D is uploaded another timeout is used to allow users to select preferred *pms* (via *CalculateCostExecution* messages) and start remote computation. Again if this timeout expires ExecutionAgent is closed.

Since each Coremuniti project Pj must be executed in a parallel way among nodes of the network, Pj is splitted into several subtask $(st_1, st_2, \ldots, st_n)$. In this section, the interaction between Server Nodes (*resource providers*) and Central Server that occur when a subtask of a project Pj needs to be retrieved and executed will be described. The main actors involved in this phase are:

- ***NodeServerAgent***: Receives the subtask to execute, downloads the data and uploads the results to the ExecutionAgent,
- ***PeerAgent***: Provides a pool of NodeServerAgent to the ExecutionAgent,
- ***ExecutionAgent***: Manages the remote execution of Pj.

When a NodeServerAgent is available, it send a message to PeerAgent with *mtype NodeHeartBeat* to notify its presence and to ask for tasks to perform. The *content* of this message, shown in table 4.1, is a list of information about the architecture and the status of the device on which the agent is running.

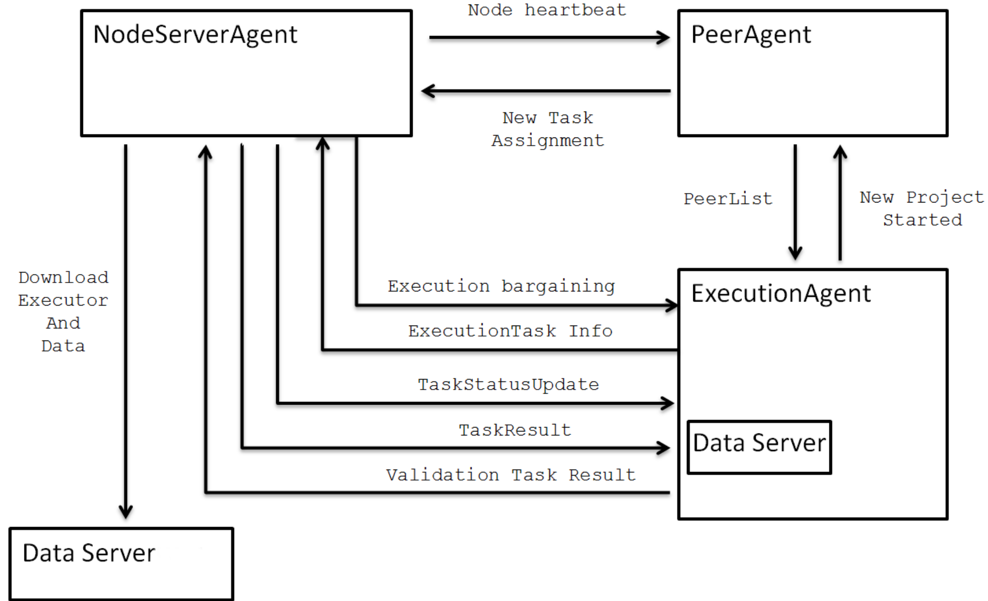


FIGURE 4.4: Task Assignment and Execution Protocol

TABLE 4.1: NodeServerAgent Beat Message Content

Field	**Description**
`Device name`	Name of the computational device
`Device type`	Type of the device. Currently supported types are *CPU* and *GPU*
`Core Number`	Number of cores of the device (optional)
`Available RAM (Mb)`	Total amount of RAM of the device
`Available HD (Mb)`	Total amount of disk space of the device
`Currently Used RAM (Mb)`	Currently used RAM of the device
`Currently Used Device Power (%)`	Percentage of device power currently used
`Currently Used HD (Mb)`	Amount of currently used disk space of the device
`Max Shared RAM (Mb)`	Maximum amount of RAM the user wants to share
`Max Shared Device Power (%)`	Maximum amount of device power the user wants to share
`Max Shared HD (Mb)`	Maximum amount of disk space the user wants to share
`Public signature key`	A public key used to sign the results sent to the central server

This message is cyclically sent to PeerAgent to notify peer availability. A node is considered alive if

$$LTU < CST - \Delta t$$

where LTU is the timestamp of $NodeHeartBeat$ message, CST is the timestamp of current server time and Δt is a fixed time interval (currently 60 seconds). In the actual state, in terms of GPUs, we only support Nvidia cards with CUDA-Enabled (Nickolls et al., 2008; Harris, 2008).

When an ExecutionAgent starts a remote project, it sends a message containing $mtype$: $NewProjectStarted$ to the PeerAgent. This agent selects a pool of available NodeServerAgents that can be assigned to the project. Afterwards it sends a message to the ExecutionAgent with $mtype$: $PeerList$ and $content$ made of a list of nodes ids that are authorized to work on Pj. Subsequently, a message with $mtype$: $NewTaskAssignment$ is sent by the ExecutionAgent to all NodeServerAgent that have been chosen. This message contains the ExecutionAgent address and port. At this point, NodeServerAgent contacts ExecutionAgent using a message of the form $mtype$: $ExecutionBargaining$ and, if authorized, it will receive an $ExecutionTaskInfo$ message. This message contains all information needed by the agent in order to execute subtask and to send results to ExecutionAgent:

- ***Executable Links***: Links to download the executable file for the assigned task. This is a list of several links, each for a different system architecture and Operating System. Currently OS supported are Windows (x86 and x64), Mac Os (x64) and Linux (x86 and x64).

- ***Params***: Parameter to pass to the executable. For the Rendering type projects are the coordinates and the number of frames to be rendered.

- ***Upload Server Data***: These are the information needed to connect to the in internal ExecutionAgent data Server: address, port, username and password.

As soon as $ExecutionTaskInfo$ message has been received, the NodeServerAgent downloads data D and starts execution of assigned subtask. Furthermore, cyclically it sends a message with $mtype$: $TaskStatusUpdate$ to notify the

local subtask execution progress. The *content* field of this message contains a couple $\langle ES, Perc \rangle$ where *ES* is the Execution Status and *Perc* is a percentage representing current state of completion of subtask. The *ES* variable can be one of the values: *DOWNLOAD*,*EXECUTION* or *UPLOAD*. When the execution of the subtask ends, the agent uploads the results (typically a file) to the Execution Agent data Server (e.g., in the 3D rendering case study, the result is an encrypted object containing the rendered tiles). A message with *mtype*: *TaskResult* is then sent by the NodeServerAgent in order to notify the ExecutionAgent about the completion of the task.

4.3 Subtasks Assignment and Credit Rewarding

In order to properly assign subtasks to resource providers the system rely on a mathematical model, described in this section, whose primary goals are the following:

1. it aims at minimizing the expected completion time for the overall task
2. it takes into account resource providers'revenue expectations

More in detail, as explained above, when a user submits a task to the Coremuniti network, this task is splitted in several subtasks, much easier to solve than the initial one. Every subtask can be completed using a reasonable amount of computational resources by the server nodes connected to the P2P network. The obtained results are then combined in order to produce the initial task solution. On the opposite side, users providing their computational power to the Coremuniti network wish to maximize their revenues by executing as much tasks as they can, in order to gain as many credits as possible. Obviously enough, the assignment should take into account the computing capability of each node in order to give to every node the chance to gain credits for their subtask executions. In the following the assignment model will be described.

The model assumes the presence of a set of available resource providers $\mathcal{RP} = \{rp_1, rp_2, \ldots, rp_n\}$, and of a function $\lambda_c : \mathcal{RP} \rightarrow N \times N$, that assigns to every resource provider rp a pair $\langle tmin, tmax \rangle$, where $tmin$ (resp. $tmax$) is the minimum (resp. maximum) execution time required by the resource node to complete a subtask.

Furthermore, the system leverage a credit assignment function which is based on the "usefulness" of the results yielded by the resource providers. Specifically, let st be a subtask whose execution was assigned c credits and which was assigned to the resource providers $rp_{i_1}, rp_{i_2}, \ldots, rp_{i_x}$. Let the resource providers $rp_{i_1}, rp_{i_2}, \ldots, rp_{i_x}$ sorted ascending w.r.t. their completion times and store them in a sequence RP_{st}. Resource providers which did not terminate their subtask are put at the end of the sequence RP_{st} ordered according to the percentage of the subtask they completed.

When at least three resource providers completed their work, the system assigns $\frac{3c}{10}$ of the total credits, paid by the client for running the subtask on the network, to the first three resource providers in RP_{st}, i.e., the resource providers which "step up on the podium" for returning st result. Moreover, he distributes the remaining $\frac{c}{10}$ credits among the other resource providers in RP_{st} as follows. For each $j \in [4 \ldots x]$, the system assigns to $RP_{st}[j]$ the credits given by the following formula

$$\frac{c \cdot Compl(RP_{st}[j])}{10 \cdot \sum_{k=4}^{x} Compl(RP_{st}[k])},$$

where $Compl(rp)$ is the percentage of the subtask st which was completed by rp.

During the task assignment phase the system aims at finding an optimal distribution of the subtask among resource providers which minimize the expected completion time while guaranteeing that:

1. to every resource provider is assigned at most a subtask,
2. it is possible, for every resource provider with a subtask assigned, to be one of the first three resource providers completing the subtask.

Subtask assignment proceeds in two phases. First, an initial assignment of the various subtasks to resource is yielded (Algorithm 1). Next, at regular intervals, the systems checks for the overall completion of the project and assigns resource providers that are available to new subtasks (Algorithm 2).

Let st be a subtask and $RP = \{rp_1, rp_2, \ldots, rp_k\}$ be the set of resource providers assigned to st. The expected completion time of st given RP is denoted as $EC_{st,RP}$ and is computed as follows. Let $\vec{t_\downarrow}$ and $\vec{t_\uparrow}$ be two vectors

reporting, respectively, the minimum and maximum completion times of the resource providers in RP in increasing order. Let t' denote the value $\vec{t_\downarrow}[2]$ and t'' denote the value $\vec{t_\uparrow}[2]$. Let $vect = [t_0 = t', \ldots, t_k = t'']$ be a vector containing all the values in $\vec{t_\downarrow}$ and $\vec{t_\uparrow}$ which lie in the interval $[t', t'']$. Hence, denoting with $F^3(x)$ the probability that at least three resource providers complete their job before time limit x, we have that:

$$
\begin{aligned}
EC_{st,RP} = \ & \int_0^\infty (1 - F^3(x))dx = t' + \int_{t'}^{t''} (1 - F^3(x))dx = \\
& t' + \sum_{i=0}^{k-1} \int_{t_i}^{t_{i+1}} (1 - F_i^3(x))dx \ = \\
& t' + \sum_{i=0}^{k-1} FF_i^3(t_i, t_{i+1}),
\end{aligned}
$$

where $F_i^3(x)$ is the probability that at least three resource providers complete their work before x time limit given that $t_i \leq x \leq t_{i+1}$ and $FF_i^3(t_i, t_{i+1})$ is equal to $\int_{t_i}^{t_{i+1}} (1 - F_i^3(x))dx$. It is easy to see that $FF_i^3(t_i, t_{i+1})$ can be easy derived from the minimum and maximum completion times associated to every resource provider by the function λ_c.

Example 1 *Consider the case that a set of four resource providers* $RP = \{rp_1, rp_2, rp_3, rp_4\}$ *was assigned to st where* $\lambda_c(rp_1) = \langle 1,5 \rangle, \lambda_c(rp_2) = \langle 2,7 \rangle, \lambda_c(rp_3) = \langle 6,7 \rangle$ *and* $\lambda_c(rp_4) = \langle 4,8 \rangle$. *In this case,* $t' = 4$ *and* $t'' = 7$ *and*

$$
\begin{gathered}
EC_{st,RP} = \\
4 + \int_4^5 (1 - F_0^3(x))dx + \int_5^6 (1 - F_1^3(x))dx + \int_6^7 (1 - F_2^3(x))dx = \\
4 + \int_4^5 (1 - \tfrac{x-1}{5-1} \tfrac{x-2}{7-2} \tfrac{x-4}{8-4})dx \ + \int_5^6 (1 - \tfrac{x-2}{7-2} \tfrac{x-4}{8-4})dx + \\
\int_6^7 (1 - (\tfrac{x-2}{7-2} \tfrac{x-6}{7-6} + \tfrac{x-2}{7-2} (1 - \tfrac{x-6}{7-6}) \tfrac{x-4}{8-4} + (1 - \tfrac{x-2}{7-2}) \tfrac{x-6}{7-6} \tfrac{x-4}{8-4})) \, dx = \\
4 + \int_4^5 (1 - \tfrac{x-1}{4} \tfrac{x-2}{5} \tfrac{x-4}{4})dx \ + \int_5^6 (1 - \tfrac{x-2}{5} \tfrac{x-4}{4})dx + \\
\int_6^7 (1 - (\tfrac{x-2}{5} \tfrac{x-6}{1} + \tfrac{x-2}{5} (1 - \tfrac{x-6}{1}) \tfrac{x-4}{4} + (1 - \tfrac{x-2}{5}) \tfrac{x-6}{1} \tfrac{x-4}{4})) \, dx = \\
4 + \left[\tfrac{11x}{10} - \tfrac{7x^2}{80} + \tfrac{7x^3}{240} - \tfrac{x^4}{320} \right]_4^5 + \left[\tfrac{3x}{5} + \tfrac{3x^2}{20} - \tfrac{x^3}{60} \right]_5^6 \\
+ \left[\tfrac{1}{10} (-126x + 44x^2 - \tfrac{17x^3}{3} + \tfrac{x^4}{4}) \right]_6^7 = \\
4 + \tfrac{901}{960} + \tfrac{11}{15} + \tfrac{31}{120} \approx 5.93
\end{gathered}
$$

Before describing the details of subtask assignment Algorithms, some additional notations will be defined. A subtask assignment $\mathcal{SA}$ is a set of pairs of the form $\langle st, rp \rangle$, where st is a subtask and rp is a resource provider. Moreover given a subtask assignment $\mathcal{SA}$ and a subtask st, we denote with $\mathcal{SA}(st)$ the

set of the resource providers assigned to st in $\mathcal{SA}$, i.e., $\mathcal{SA}(st) = \{rp|\langle st, rp\rangle \in \mathcal{SA}\}$, and we denote as $\mathcal{ST}(\mathcal{SA})$ the set of subtasks mentioned in $\mathcal{SA}$, i.e., $\mathcal{ST}(\mathcal{SA}) = \{st|\langle st, rp\rangle \in \mathcal{SA}\}$. Furthermore, given a subtask assignment $\mathcal{SA}$ we define as $MaxECP(\mathcal{SA})$ the maximum expected completion time of a subtask in $\mathcal{SA}$, i.e., $MaxECP(\mathcal{SA}) = \max_{st \in \mathcal{ST}(\mathcal{SA})}(EC_{st,\mathcal{SA}(st)})$.

The initial subtask assignment is computed by running Algorithm 1. It works in two phases. First, an initial assignment of resource providers to tasks is yielded by assigning a resource provider at a time to the every subtask paying attention to minimize the maximum expected completion time of all the subtasks. This is done considering resource providers in ascending order of their expected completion time (lines 2-8). Next, the initial assignment is revised by swapping pairs of task between resource providers. The algorithm iteratively selects a pair of assignments $\langle st', rp'\rangle, \langle st'', rp''\rangle$ which once swapped provide the greatest decrement of the maximum expected completion time (lines 9-12). Finally, the subtask assignment is returned.

Algorithm 1: Initial Assignment

Input: Resource provider sequence $\mathcal{RP}$ of size n
Input: Subtask sequence $\mathcal{ST}$ of size $k < \frac{n}{3}$
Output: Tasks assignment $\mathcal{SA}$

1: $\mathcal{SA} = \emptyset$
2: $\mathcal{RP} = order(\mathcal{RP})$
3: **for** $i = 1$ to n **do**
4: $st = selectBestST(\mathcal{ST}, \mathcal{SA}, \mathcal{RP}[i])$
5: **if** $st \neq$ `null` **then**
6: $\mathcal{SA} = \mathcal{SA} \cup \langle st, \mathcal{RP}[i]\rangle$
7: **end if**
8: **end for**
9: **repeat**
10: $(\langle st', rp'\rangle, \langle st'', rp''\rangle) = selectBestCP(\mathcal{SA})$
11: $\mathcal{SA} = \mathcal{SA} - \{\langle st', rp'\rangle, \langle st'', rp''\rangle\} \cup \{\langle st', rp''\rangle, \langle st'', rp'\rangle\}$
12: **until** $existsCP(\mathcal{SA})$
13: **return** $\mathcal{SA}$

Function *order* receives as input a sequence of resource providers $\mathcal{RP}$ and returns $\mathcal{RP}$ sorted ascending w.r.t. the expected resource provider completion time. Function *selectBestST* returns, given a resource provider rp and a subtask assignment $\mathcal{SA}$, the subtask $st \in ST$ such that $MaxECP(\mathcal{SA} \cup \{\langle st, rp\rangle\})$

is the minimum, i.e., $st = \arg\min_{st \in \mathcal{ST}}(MaxECP(\mathcal{SA} \cup \{\langle st, rp \rangle\}))$, such that $MaxECP(\mathcal{SA} \cup \{\langle st, rp \rangle\}) < MaxECP(\mathcal{SA})$, `null` otherwise. Since in the initial step of the algorithm there could be some subtasks st in $\mathcal{ST}$ such that $|\mathcal{SA}(st)| = x < 3$, for each subtask st satisfying this constraint, when computing $MaxECT$ function *selectBestST* assumes that $3 - x$ fake resource providers have been assigned to st, where the minimum and maximum completion times of these fake resource providers are $max - 1$ and max, respectively, where max is a constant value (much) greater than the maximum completion time of every the resource provider in $\mathcal{RP}$.

Function *selectBestCP* returns a pair of assignments $\langle st, rp \rangle$ and $\langle st', rp' \rangle$ in $\mathcal{SA}$ such that there do not exists an assignment $\langle st'', rp'' \rangle$ and $\langle st^*, rp^* \rangle$ in $\mathcal{SA}$ such that
$MaxECP(\mathcal{SA} - \{\langle st, rp \rangle, \langle st', rp' \rangle\} \cup \{\langle st, rp' \rangle, \langle st', rp \rangle\}) >$
$MaxECP(\mathcal{SA} - \{\langle st'', rp'' \rangle, \langle st^*, rp^* \rangle\} cup \{\langle st'', rp^* \rangle, \langle st^*, rp'' \rangle\})$.
Function *existsCP* returns true if there is a pair of assignments $\langle st, rp \rangle$ and $\langle st', rp' \rangle$ in $\mathcal{SA}$ such that
$MaxECP(\mathcal{SA} - \{\langle st, rp \rangle, \langle st', rp' \rangle\} \cup \{\langle st, rp' \rangle, \langle st', rp \rangle\}) < MaxECP(\mathcal{SA})$.
Moreover, both functions *selectBestCP* and *existsCP* consider only pairs of assignments $\langle st, rp \rangle$ and $\langle st', rp' \rangle$ in $\mathcal{SA}$ such that swapping rp and rp' guarantees that the probability that rp (resp. rp') is among the first three resource providers assigned to st' (resp. st) completing st is greater than zero.

The following proposition describe the behavior of Algorithm 1.

Proposition 1 Let $\mathcal{RP}$ be a sequence of resource providers of size n and $\mathcal{ST}$ be a sequence of tasks of size $k < \frac{n}{3}$. Algorithm 1 returns a subtask assignment $\mathcal{SA}$ in polynomial time w.r.t. $|\mathcal{RP}|$ and $|\mathcal{ST}|$ such that

1. for each $st \in \mathcal{ST}$ $|\mathcal{SA}[st]| \geq 3$, and
2. for each $rp \in \mathcal{RP}$ if $\langle st, rp \rangle \in \mathcal{SA}$ the probability that rp is among the first three resource providers that complete st is greater than zero.

Proof (Sketch). The first property follows from the hypothesis that $k < \frac{n}{3}$ and the fact that the initial assignment of resource providers to tasks (lines 2-8 is done by selecting the subtask st that provides the largest decrement of the maximum expected completion time. Therefore, it is easy to see that, as in the case that a subtask st have been assigned less than three resource providers,

at least one fake resource provider is considered in the computation of *st* expected completion time and the minimum completion time of a fake resource provider is much larger than the maximum completion time of any (real) resource provider. Furthermore, it cannot happen that more than three resource providers are assigned to a subtask before that all the other tasks have been assigned at least three resource providers.

As regards the second property, it straightforwardly follows from the definition of functions *selectBestST* and *selectBestCP*.

Finally, it can be show that the algorithm runs in polynomial time w.r.t. $|\mathcal{RP}|$ and $|\mathcal{ST}|$ by proving that in the case that a pair of assignments is returned by function *selectBestCP* at least one of the resource providers that will be swapped, will never be returned by subsequent invocation of function *selectBestCP*. □

Coremuniti central server runs Algorithm 2 periodically, in order to check for the overall completion of the project and to assign resource providers, that have completed the assigned tasks, to new tasks. The statistics of the resource providers involved in the computation of the subtasks are updated before running this algorithm, so that the system deals with up to date values for minimum and maximum completion times. Essentially Algorithm 2 first sorts the available resource providers ascending w.r.t. their expected completion times and next iteratively assigns each resource provider rp to the subtask *st* selected by invoking function *selectBest*. Function *selectBest* returns, given a resource provider rp and a subtask assignment $\mathcal{SA}$, the subtask $st \in ST$ such that $MaxECP(\mathcal{SA} \cup \langle st, rp \rangle)$ is the minimum, i.e.,
$st = \arg\min_{st \in \mathcal{ST}}(MaxECP(\mathcal{SA} \cup \langle st, rp \rangle))$ and is such that
$MaxECP(\mathcal{SA} \cup \{\langle st, rp \rangle\}) < MaxECP(\mathcal{SA})$, `null` otherwise.

4.4 Case study: Efficient and Effective 3D rendering

Many real life application requires huge computing resources in order to properly execute. As an example we mention here physical science simulation, mathematical simulation for insurance companies, biology simulation and cryptography. In this section, the Coremuniti^{TM} based solution will be described for a quite interesting scenario, i.e., 3D Professional Rendering. More in detail, rendering is the process of converting a graphical model into an high quality

Algorithm 2: Incremental Assignment

Input: Available resource providers sequence $\mathcal{RP}$ of size n
Input: Subtask sequence $\mathcal{ST}$
Input: Initial Subtasks assignment $\mathcal{SA}$
Input: Subtasks and Resource Providers constraints STC
Output: Revised Subtasks assignment $\mathcal{SA}$

1: $\mathcal{RP} = order(\mathcal{RP})$
2: **for** $i = 1$ to n **do**
3: $st = selectBest(\mathcal{ST}, \mathcal{SA}, \mathcal{RP}[i])$
4: **if** $st \neq$ `null` **then**
5: $\mathcal{SA} = \mathcal{SA} \cup \langle st, \mathcal{RP}[i] \rangle$
6: **end if**
7: **end for**
8: **return** $\mathcal{SA}$

image by means of a computer program. The input of the rendering process is the *model* (also called *scene*), composed of texture, lighting, shading, viewpoint and geometry information. Starting from these information, many rendering algorithms have been implemented in order to obtain the final image. The most important concept used by these algorithms is *light tracing*. A naive solution to the rendering problem could be obtained by tracing every particle of light from every source to every object in the scene. This solution is (obviously) impractical because it calls for massive amount of computational time. Moreover, it is useless because some portion of the scene being rendered will not be visible in the final image in the majority of real life rendering task. In particular, only a space region of the model may appear as a part of the rendered image, it is referred as *view frustum*. The exact shape of this region varies depending on many factors such as the viewing direction and the camera being used.

Based on this assumption a more efficient light transport modeling techniques have been defined, such as:

- *Ray casting*: it runs the rendering process starting from a point of view and using basic reflection laws and elementary geometry computes the rendering output. The obtained image quality could be improved by using *Monte Carlo* based correction techniques;

- *Ray tracing*: it works analogously to Ray Casting, but it is based on more sophisticated optical laws (Glassner, 1989).

The use of the above mentioned approaches allows to limit the light tracing execution only to those pixels that will be visible in the final image, nevertheless, the rendering process still requires high performance resources for running to completion if the input scene is (slightly) complex.

Although many advances in computing device technology led to affordable computational power (thus ameliorating the curse of huge execution time for professional use), the always growing demand of photorealistic results and high resolution artistic products, make the rendering problem more and more challenging. Coremuniti provides a nice solution to this problem by taking advantage of the resources provided by users joining the P2P network. Figure 4.5, shows the Coremuniti Server status running a rendering task. In particular, resource providers can set the amount of resources they are willing to share both as a percentage of the the total CPU usage (in Fig. 4.5 it has been set to 80%) and the maximum amount of memory (in Fig. 4.5 it has been set to 1 GB).

It is straightforward to note that the rendering task is highly parallelizable as frames and tiles (composing each frame) can be rendered independently. Parallel rendering drastically improves the speed of rendering operations. If we take into account ray tracing we can divide the whole view frustum into portions. Each piece of the frustum can be rendered on different nodes using the appropriate rays in order to obtain the corresponding portion of the rendered image. It is easy to see that single machine execution requires to send rays to all the pixels in the frustum slowing down the overall rendering time.

Mozaiko natively provides the opportunity to parallelize the rendering process on Coremuniti network. In the first implementation the software *Blender* [1] is used to render models. Blender is a professional computer graphic software available for many platforms that is actually considered one of the best tool for professional rendering and movie making. As a default add-on, Blender provides *Cycles* render engine that is based on ray-tracing. An interesting feature provided by Cycles is the *Render Border* function. Border rendering works on a smaller piece of the original render model. It is typically used to refine the rendering of an image portion that requires longer execution times w.r.t.

[1] https://www.blender.org

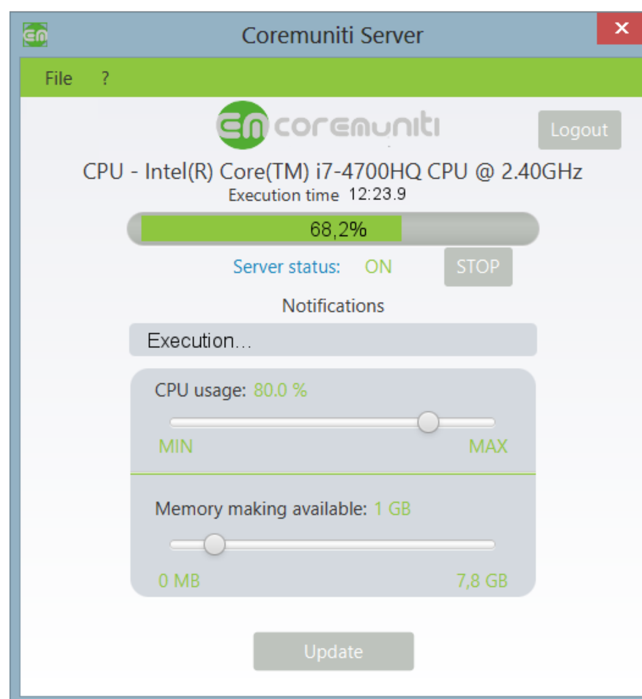


FIGURE 4.5: Coremuniti Server At Work

the rest of the image or for previewing purposes. The images are then easily combined in order to obtain the final rendered image.

More in detail, the rendering process starts from the pixel that will be visible in the final image and sends a "ray" into the scene to identify the exact colour for that pixel. Each ray will bounce around the scene from object to object to light source based on rendering settings set to calculate the final result. While the render border will reduce the pixels used as the starting point for each ray, it does not reduce the objects or lights in the scene that each ray may encounter on its way. Each ray crossing the scene will still "see" every visible object and light effects in the scene that can influence the final result for each pixel. Using border rendering, then, does not introduce any degradation on the quality of the resulting output image. Mozaiko uses border render function to split the original model to be rendered into rectangular regions that are sent to the computing nodes for parallel rendering. As the intermediate results from each node are gathered, the rendered regions are merged into the final image. Fig. 4.6 depicts the Mozaiko client. It allows to check the completion status of the tasks that have been launched and to see the preview of the

FIGURE 4.6: Mozaiko Rendering View

rendering output. It is also possible to control the execution flow and to check the credits needed to perform the rendering.

Rendering Task Validation. A crucial requirement for Coremuniti architecture is the detection of incorrect task execution. Since the system renders the models using distributed resources, it is mandatory to implement a checking strategy to determine if the rendering task solved by peers is correct. In particular, the system needs to output properly rendered files to requesting users and identify possible malicious actions (e.g., a user submitting a wrong image with the intent of damaging the network). As explained in section 4.3, the system asks several peers to solve a specific task and waits for their answers. Since these results may slightly differ each other, the system needs to find a way to evaluate them in order to output the best one w.r.t. the requesting user viewpoint. As this process involves complex neurological mechanism that are beyond the scope of this work, we simply mention here that a method based on visual perception metrics (Witzel, Burnham, and Onley, 1973) was applied in order to determine the similarity threshold for the pixels that differs among rendered images. If the difference is higher than the obtained threshold, the

system randomly chooses a small subset of pixels to render on the internal specialized backup server in order to determine the best result to be returned to the requesting client. As a final note, we point out that, it is mandatory to prevent the injection of malicious script in the model to be rendered (e.g., Blender allows to execute custom python script in a 3D model). Into the Coremuniti software, some security actions are performed. For example the execution of unnecessary scripts are disabled and in case that the script is claimed by the client to be crucial for the rendering completion, the system runs it in a secured sandbox (Payer, Hartmann, and Gross, 2012).

4.5 Experimental Evaluation

This section is devoted to experimentally evaluate the Coremuniti performances from two standpoints:

1. Efficiency and effectiveness evaluation of the Coremuniti approach by comparing his performance against the state of the art solutions available;

2. Comparison of the revenue that users can get by joining Coremuniti network w.r.t. other collaborative approaches that reward users (e.g., cryptocurrency mining).

We focus our attention on two different kind of tasks: 3D rendering and matrix multiplication (implemented by the highly parallelizable Strassen algorithm described in (Li, Ranka, and Sahni, 2011)) since it is one of the basic operation behind most of computer simulations. For the latter we used the dataset of the University of Florida Sparse Matrix Collection [2]

The experimental evaluation was carried out on a testnet network that involve 40 peers that have been classified according to the computational power of devices. The network features are reported in Table 4.2.

As explained above, the Coremuniti system works in parallel with the normal user activities. This is a crucial assumption as traditional approaches for high performance computing relies on dedicated resources. We measured the running time of the rendering tasks by considering several usage scenarios for resource providers:

[2] Available at https://sparse.tamu.edu/

Type	Equipment	Peer Percentage
BASIC	Intel i5-6440HQ - 8GB RAM - GPU Nvidia GeForce GTX 950M	60
INTERMEDIATE	Intel i7-4790K 4GHz - 32GB RAM - GPU Nvidia 980 Ti 6 Gb RAM	25
ADVANCED	Xeon E5-26700 2.60GHz - 128GB RAM - GPU NVIDIA GK110GL	15

TABLE 4.2: Computation Power of Resource Providers in Test-net Network

- **Low-Use:** Users who are performing activities that do not require huge amount of computing resources such as browsing simple pages and/or editing text (this is the system best case)
- **Medium-Use:** Users who are playing multimedia files (this is the system medium case)
- **Intensive-Use:** Users who are playing video games (this is the system worst case)

These 3 scenarios were simulated using ad-hoc software solutions installed on the peers of the test network. For Low-Use simulation a system that open and interact with with productivity tools (e.g. Libreoffice Writer, Calc, Base ecc.) has been developed. For Medium-Use, films or playlist of soundtrack have been performed on the peers and for Intensive-Use, some 3D games benchmark were performed on network nodes.

For each possible scenario, three task categories (each category being composed of 10 tasks) were performed. For each category a set of models have been chosen so that can be partitioned respectively in 10, 20 and 30 subtasks but it must be said that there is in principle no fixed upper bound for the number of subtastks.

4.5.1 System Performances

In order to validate the framework a comparison was conducted against some of the most popular specialized rendering server farms in the case of 3D rendering (i.e. RenderStreet[3] and RebusFarm [4]) and some clusters on well known cloud service platform in the case of matrix multiplication. Below the used configurations on cloud services:

[3]https://render.st
[4]https://us.rebusfarm.net/en

- *Microsoft Azure.*[5] 8 Virtual Machines D5 V2 having 2.4 GHz Intel Xeon® E5-2673 v3 - 16 Core (Virtualized) CPU, 56 Gb RAM, 800 GB HD - Linux Centos OS.
- *Amazon EC2.*[6] 8 Virtual Machine m5.4xlarge (On-Demand plan) having 3,1 GHz Intel Xeon® P 8175 - 16 Core (Virtualized), 60 Gb RAM 800 GB HD (via ABS - Amazon Block Store) - Linux Centos OS.
- *DigitalOcean.*[7] 8 Virtual Machine having Intel Xeon® Skylake (2.7 GHz, 3.7 GHz turbo) - 16 Core (Virtualized), 64 Gb RAM 200 GB HD - Linux Centos OS.
- *Google Cloud*[8] 8 Virtual Machine n1-standard-16 having Intel Xeon® Skylake (2.7 GHz, 3.7 GHz turbo) - 16 Core (Virtualized), 60 Gb RAM 200 GB HD - Linux Centos OS.
- *IBM Cloud*[9] 8 Virtual Machine B1.16x64 having Intel Xeon® CPU - 16 Core (Virtualized), 64 Gb RAM 25 GB HD - Linux Centos OS.

Time Performances

Figures 4.7 and 4.8 show the execution times of the Coremuniti solution by running a first series of tests using the software to simulate the above mentioned usage scenarios. The benchmark used for experiments of the Figure 4.7 is the 3D Rendering while for the Figure 4.8 is the matrix multiplication. In order to extremely stress this approach, we gave to competitors the advantage of discarding the (eventual) set-up time for the cloud and the server farm devices (marked in Fig. 4.7 as $*$ and $**$). More in detail, both Render Farm and Render Street require users to sign up and choose a profile type, if the user chooses the cheapest one, her tasks will be categorized as low priority. This significantly slows down the overall execution time since the task could start hours later. As for the cloud cluster, in addition to the cluster set-up times, advanced skills are needed to be able to create an intra-cluster network and let the various machines communicate with each other.

[5] https://azure.microsoft.com
[6] https://aws.amazon.com
[7] https://www.digitalocean.com
[8] https://cloud.google.com/
[9] https://www.ibm.com/cloud

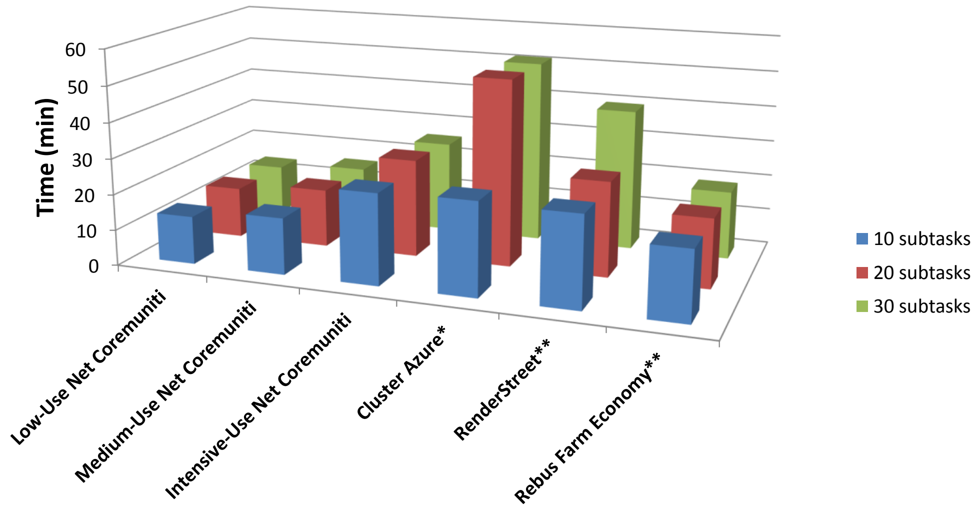


FIGURE 4.7: Execution Times Comparison - 3D Rendering

It is easy to see that, except for the system worst case scenario (i.e., when the service providers are all intensively busy with other activities) the Coremuniti execution times are always better than the competitors. Even in the intensive use case, the system times are slightly higher than RenderFarm. However, we point out that the set up time have been discarded, so if this is summed up to the execution time, is likely to be higher than Coremuniti execution in the system worst case. Finally, it is worth noticing that the execution times of the cloud based solution is almost double between the 10 and the 20 subtask runs, while they remain almost the same between the 20 and the 30 subtask. This is due to the fixed number of nodes and to the higher number of scene frames with respect to node number that fail to execute all the frames simultaneously. A similar issue occurs on RenderStreet between the 20 and the 30 subtask runs.

As regards to the matrix multiplication scenarios, it is easy to see that Coremuniti performances are still better than the cloud services we compare to in runs with 20 and 30 subtasks with only exception of Intensive-Use of 20 subtasks w.r.t. DigitalOcean and Google Cloud where execution times are slightly higher. For short tasks, for the cases of Medium-Use and Intensive-Use, the execution times of cloud cluster are low, while for Low-Use times are very

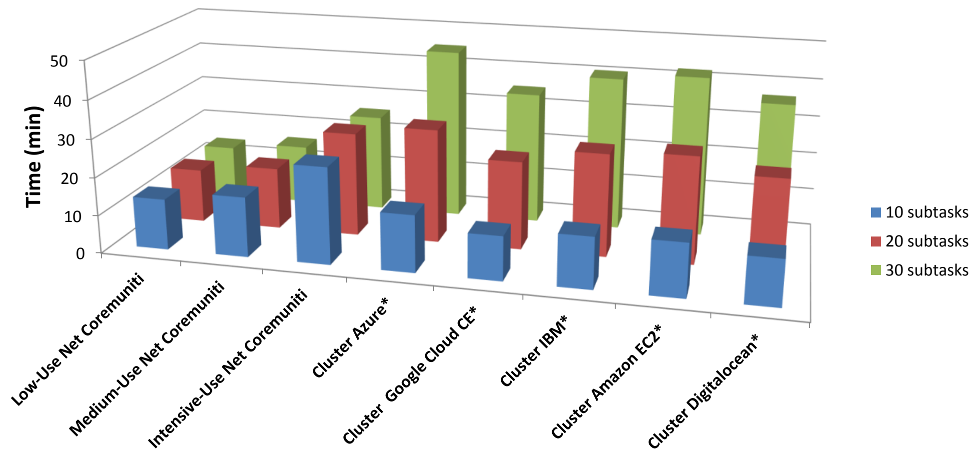


FIGURE 4.8: Execution Times Comparison - Matrix multiplication

similar. As already mentioned, in this case the cluster set-up and virtual machine startup times are not considered so if this is summed up to the execution time, is likely to be higher than Coremuniti execution also in 10 subtask run.

Cost Evaluation

The Coremuniti strongest advantage w.r.t. the competitors is the money that users can save using this network. Due to the system peculiar model it does not suffer from fixed cost deriving from continuous update of hardware infrastructure.

Indeed, users joining the network are mainly professionals and technical faculty university students that own up to date computing devices, thus the network do not suffer of technical obsolescence. This competitive advantage, reflects on Coremuniti cost model, where it is used a fixed price for each task (in this use-case scenario is 0.12 Euro).

As shown in Figure 4.9, the total cost paid by users executing their tasks on Coremuniti network is always lower than the cost paid if using render farm or cloud services in case of Rendering. Moreover, as for the time performances,

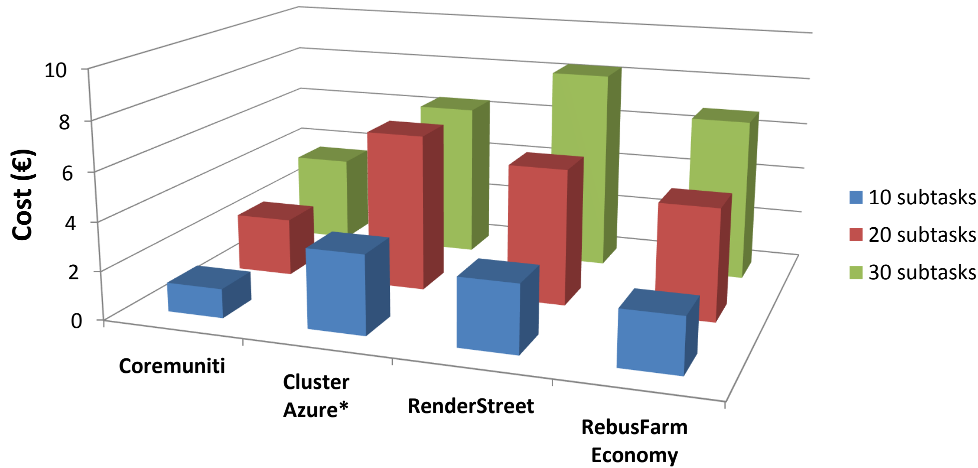


FIGURE 4.9: Rendering Cost Comparison

we gave to the competitor some advantage in the calculus, i.e., we do not account the cost for signing to their service, while users can join the Coremuniti network for free.

Figure 4.10 show the cost comparison of Coremuniti w.r.t. cloud cluster. In this case Coremuniti cost are lower than Azure Cluster, while they are similar to other cloud providers. However, in this comparison, only the execution costs were considered. In the use of cloud clusters other hidden costs are present as the cost of set-up the cluster, the cost of storage and the cost of network.

Furthermore, we do not account also the possibly accumulated credits by users that prior to ask for additional resources may had provided their own computational time to the network. Finally, the system redistributes the 80% of the paid cost to reward the resource providers. Next paragraph is devoted to discuss this issue.

4.5.2 User Revenues Evaluation

As mentioned above the system redistribute to resource providers a great portion of the money paid by users requesting computational power to the Coremuniti network. In order to compare the possible outcome for joining Coremuniti network with respect to other collaborative systems that rewards their

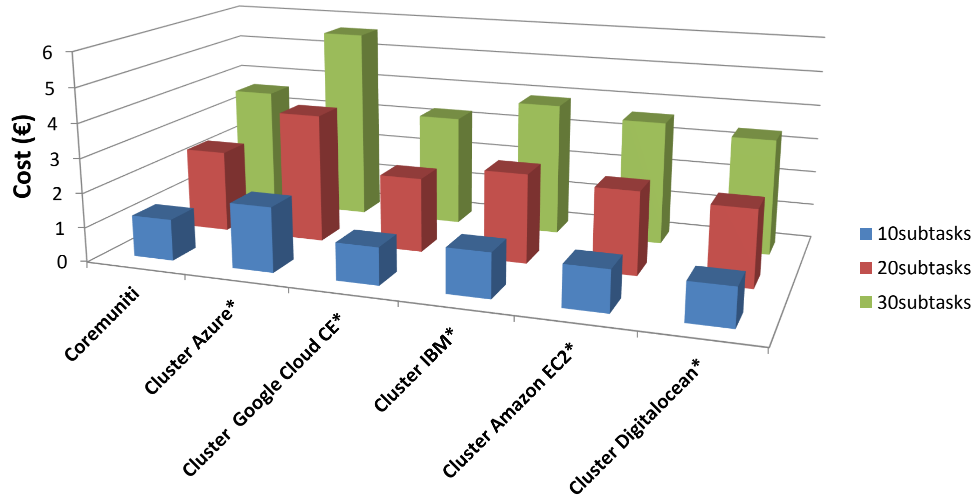


FIGURE 4.10: Matrix Multiplication Cost Comparison

users we compare the system performances to the most popular services currently available using their own metrics, i.e. *Bitcoin* mining [10], *Ethereum* smart contracts [11] and *Monero* mining [12] and . First of all, it is necessary to clarify that those services require intensive-use advanced devices in order to give users a chance to be rewarded (according to publicly available statistics). Table 4.3 shows more details on the key parameters for actual system comparison. The parameter *Network Difficulty* is peculiar of cryptocurrency mining while it is not applicable to Coremuniti network. We considered a resource provider holding a video card *Nvidia GTX 980 Ti (EVGA)* fully dedicated to task execution for the considered networks. We were forced to choose this device because the basic and intermediate settings are not powerful enough to get neither Bitcoin or Ethereum or Monero revenues. The device is able to complete a Coremuniti task in 5 minutes and 30 seconds. This means that it is able to complete 10 tasks per hour while for mining the other coins it can decode the number of hashes reported in Table 4.3. Considering that the Coremuniti system redistribute to the network 0.12 Euro for each accumulated credit, by applying the mathematical task assignment model (as shown in 4.3) the user

[10]https://alloscomp.com/bitcoin/calculator
[11]https://badmofo.github.io/ethereum-mining-calculator
[12]https://www.cryptocompare.com/mining/calculator/xmr

TABLE 4.3: Gain Comparison

Network	Max Effort	Network Difficulty	Coin Gain	Current Rate	Daily Gain (Euro)
Coremuniti	10 subtask/hour	-	21,6	0,12	2,07
Bitcoin	240 Mhash/s	7,4094 T	0,00000001	10195	0,000090
Ethereum	18,5 Mhash/s	159418 T	0,001932	295,75	0,51
Monero	600 Hash/s	37,1689G	0,00384	109,72	0,37

can get up to 2.07 Euros per day. In this calculation we suppose that the user that the user arrives in the top 3 positions at least 50% of the time. Also, another main difference between cryptocurrency mining and Coremuniti is that the first always returns a portion of coin to the miners during the 24 hours, while for the Coremuniti system it is necessary that there are some users who launches tasks in the system to be able to distribute credits. For this motivation in the calculation was assumed a 12-hour execution time a day for Coremuniti peers. As shown in the table 4.3 also with this assumption and limitation, the Coremuniti gain is 4 times higher than revenues obtained by Ethereum (which turns out to be the most profitable cryptocoin for the used devices)[13].

[13] We considered the exchange rates available on 21st Jun 2019 on Poloniex Exchange (https://www.poloniex.com)

Chapter 5

Enhancing Big Data Information Search

5.1 Introduction

The issue of devising novel solutions for analyzing Big Data, coming both from heterogeneous information sources and from logs of user interactions and behaviors, is becoming more and more compelling in the construction of *Intelligent Information Systems* (IIS) to assist end user in the search of relevant information and in the interaction with services in the net. To this end, several research issues need to be dealt with.

First of all, data gathered by social network and search engines available on the web are inherently non structured; therefore, a data exchange task has to be performed for moving source data into a target "structured" database enabling an effective analysis of user behavior. In the proposed framework, source data should be also enriched with additional background information and knowledge patterns on the application domain, derived by previous analysis of experiences from both experts and simple users. In this chapter, the following challenge will be investigated: *devising a data exchange setting for enhancing the information content of source data*. As an example, given raw data about restaurants and post reviews, the system will have to classify the restaurants according to the various "dimensions" (i.e., categorized properties or other features) that have been singled out and evaluated in the reviews.

A relevant issue is to adapt the content of target data to the strategy followed by the users when seeking for useful information. In particular, the user may be driven by some predefined faceted features (*browsing*) or may simply

formulate a query using “free" keywords (*searching*). Most of the present systems mainly follow one of the two mentioned paradigms and only few systems offer a mix of the two of them. A new frontier for IIS is to combine searching and browsing by using features that are not a-priori predefined but selected for and adapted to the search context. The challenge explored in this chapter is: *detecting on-the-fly features that are relevant in the search context and tailored to the user behavior.* As an example, consider the case of a user expressing an enthusiastic comment on a recently visited city (e.g., by a Facebook “like" and/or post). Most likely, his/her friends could be interested to utilize the positive feedback on that city when deciding their next travel destination. In this respect, enabling users to exploit some useful information in the search for a specific destination can be done by suggesting some categorized high level information (e.g., “funny cities for young people"), which can be eventually further refined (e.g., by adding specific search keywords).

The relevance of user behavioral features makes necessary to investigate the following two additional issues: *How to allow relevant information about user search preferences propagate over the network? How to measure the possible information spread*? The two issues are preliminary to the activity of discovering new dimensions to be added to the information sources and to be later used to support a search mixed with tailored browsing.

In sum, in this chapter we focus on (i) the analysis of user’s searching and comment posting activities in order to identify potentially interesting suggestions about user searches, (ii) the proposal of a novel data exchange setting that exploits them for enhancing the information contents of the source databases and (iii) the definition of a search strategy based on tailored faceted features. This approach is motivated by the observation that both performed searches and posted comments define a quite accurate profiling of user wishes and feedbacks that can be exploited to construct background information and knowledge in an application domain for supporting advanced further searches. The final result of this work is a user behavior oriented search framework for implementing new generation IIS.

Clearly enough, exploring Big Data in IIS has several implications aside the technical ones. In particular, the merge of data collected from distinct information sources may cause the so-called private information leakage. Therefore, a IIS should be aware of possible privacy threats as discussed in (Narayanan

and Shmatikov, 2008). Indeed, that paper has shown that, by using the Internet Movie Database as the source of background knowledge, it is possible to successfully identify the Netflix records of known users, uncovering their apparent political preferences and other potentially sensitive information. This scenario gives an intuition of the relevant privacy problems that arise when users interact with IIS. This drawback has been mentioned for the sake of completeness, although dealing with privacy issues is beyond the scope of this work.

The selected application scenario is tourism recommendation. The ratio behind the choice of this scenario is that, users (i.e., tourists) do an extensive search activity when traveling and they post comments and suggestions, thus contributing to the definition of their behavior and needs and to the construction of a background knowledge ground to assist them in further searches.

Example 2 .

Consider the scenario depicted in Figure 5.1. Each node of the network represents a user interacting with a social network[1]. When interacting with the social network, users usually issue several queries, post comments and upload (tagged) files. For example a user may pose the following query: *Find a restaurant in Milan*. Traditional search engines will provide user results ranked on the basis of their default criteria. However, this ranking could be ineffective as users may not be satisfied by query answers as they are mainly based on proximity search and some fixed categorization (e.g. stars, price). In order to overcome this limitation, the proposed system performs a data pre-processing by clustering user comments stored in the system. As a result, a set of comments groups are obtained that may contain some new possible search categories (referred in the following as search *dimensions*) previously hidden in the data. It could therefore happen that the clustering algorithm suggests to add a new dimension to the source information that classifies the quality of dishes served at each restaurant, while matching the user query for a restaurant located in Milan area. The new dimension could be *Food Quality*. As users interact with the system and new enquiries and comments are made, some other additional dimensions could arise. For instance, the *Food Quality* could be refined by an additional dimension *fresh fish* with suitable values: *bad/good/excellent*.

[1]In the prototype Facebook, Yelp and Twitter are used but the same reasoning applies to every social network

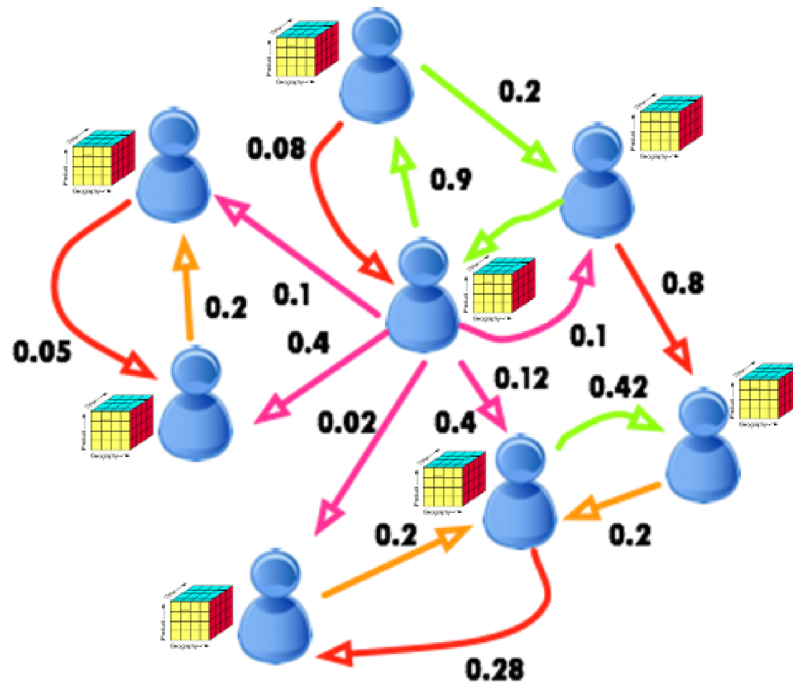


FIGURE 5.1: Influence Model for User Search

Furthermore, as users may be *influenced* by their social contacts, the new dimensions might spread in the network so that an increasing number of queries include them. Therefore, to understand how the discovered dimensions spread over the network, we need to take into account the probability that a user could be influenced by her/his social contacts. By simply considering how the new dimensions spread in the network, we can eventually detect the influence spreading on the network and update the user connection graph consequently. More in detail, by analyzing the social environment of each user and her/his search activity log over time, it is possible to label each edge with a value representing the influence currently exerted by a user to its connections – see the labeled edges in Figure 5.1. The influence plays a crucial role for filtering dimensions that the clustering algorithm will suggest for a tailored search.

Data posting is used both to enrich the source raw data with the discovered dimensions and to personalize them for the current user so that the enriched information may be added to the search toolbar as soon as search keywords (e.g., “starters” or “main courses”) are typed for which the dimensions are pertinent, thus enabling a faceted browsing – this functionality is represented

as a cube aside the user symbol in Figure 5.1. □

5.1.1 The proposed framework in a nutshell.

In this paragraph, an high-level description of proposed framework for implementing user search task described is provided in the example above. Figure 5.2 depicts the overall process of information analysis, enrichment and delivery to the final user.

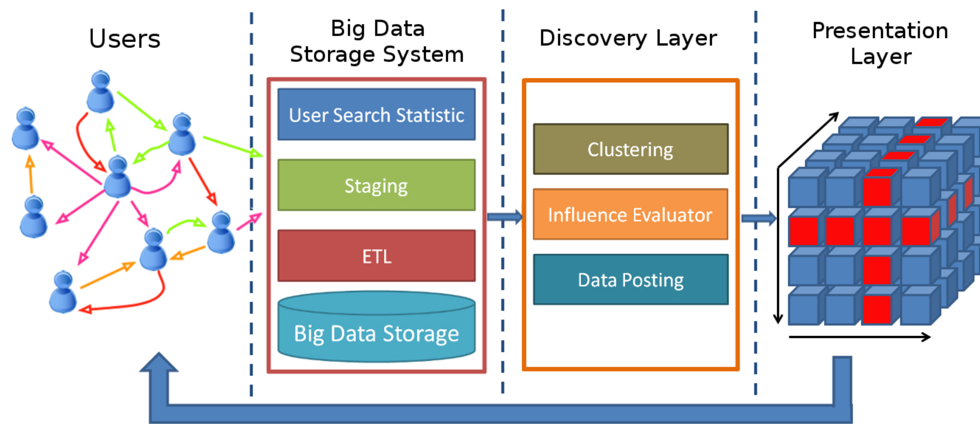


FIGURE 5.2: Overview of Framework

Data about user activities and interactions with a social network are collected by *Big Data Storage System*, which performs the following preliminary operations: i) it first computes some high level statistics about user search (e.g. most frequent search keywords) that will be refined during the information analysis and enrichment steps; ii) it stores all the collected data in a staging area to overcome potential problems due to different speed, size and format of incoming data; iii) it performs the proper extraction, transformation and loading operations for making data well suited for efficient storage (space saving) and analysis (fast execution time); iv) finally, it stores the pre-processed data in a structure tailored for Big Data.

After data pre-elaboration takes place, data goes through the *Discovery Layer*, which includes three modules. The first one is based on a suite of clustering algorithms that are devoted to the extraction of unsupervised information hidden in the collected data. The obtained clusters, are filtered by the

influence evaluator module that identifies the dimensions that have the potential to spread across the network. After this filtering step, high level information will be used by the data posting module to enrich information sources. We stress that data posting differs from classical data exchange because, while moving data, the contents are enriched by supplying additional pieces of information. To this end, a source database is linked with additional tables, called *domain relations*, that store the dimensions to be added into the target database and aggregate data dependencies (in particular, count constraints) are used to select dimension values that better characterize user behaviors. We observe that the dimension-based structure of a target database can be effectively supported by a column-based Big Data storage system – actually, in this system implementation a distribution of HBase was used.

Finally the *Presentation Layer* is used to personalize the enriched information for the current user query and to present portions of it (dimensions) into the search toolbar that are pertinent with entered keywords for enabling a faceted browsing.

5.1.2 Improving Faceted Navigation by Clustering

The faceted search pattern can be enhanced by exploiting a data mining approach for information enrichment. Among the plethora of data mining algorithms proposed in literature, we focused on Clustering. The rationale for this choice is described in the following. When users interact with web based systems, either for information browsing and searching or for posting comments and suggestions, they provide useful information about their behaviors. This information can be exploited for an accurate user profiling that is the basis for designing better user-oriented services. Unfortunately, no information about a possible classification of user features is easily available as no labeled examples can be collected. To overcome this limitation, a clustering solution has been used due to its unsupervised features. The addressed clustering problem can be formalized as follows: Given a set $\mathcal{O} = \{o_1, \ldots, o_n\}$ containing n objects[2], cluster analysis aims at producing a partitioning[3] $\mathcal{C} = \{C_1, \cdots, C_k\}$

[2]For the sake of generalization we do not distinguish between query text and post as both of them can be considered as plain text objects

[3]In this chapter we refer to the hard clustering problem, where every data point belongs to exactly one cluster.

of the objects in O, such that objects in the same set C_i are maximally similar and objects in different sets are minimally similar, according to some similarity function. Consequently, each object $o \in \mathcal{O}$ is contained in exactly one set C_i. These sets $C_i \in \mathcal{C}$ are called clusters.

More in detail, given an input dataset the following steps are performed:

- Given the initial set $\mathcal{O}$ of objects, a partition $\mathcal{C}$ of $\mathcal{O}$ is provided. The feature set to be used for representing objects is derived by the data source (e.g. timestamps, location, etc.);

- A technique for discovering cluster labels which is based on the notion of *discriminative cluster patterns* is performed. Discriminative cluster patterns highlight all the characteristics of a given cluster, since they are expected to lie in a specific cluster and at the same time not to lie in any other cluster;

- The initial partition is incrementally updated according to a (possibly infinite) stream $\{o_{n+1}, \ldots, o_{n+k}, \ldots\}$ of new incoming objects. In this respect, each object o_i (may) induce a new partition $\mathcal{P}_i$ that could contain a different number of discriminative features.

A detailed description of the clustering strategy is beyond the scope of this work, however, we briefly recall some basic concepts that have been exploited in the system.

When dealing with data containing textual information, a major issue is the selection of the set of relevant terms, or *index terms*, i.e., the terms capable of best representing the topics associated with a given textual content. In order to achieve this, some standard text processing operations are used (Moens, 2000; Baeza-Yates and Ribeiro-Neto, 1999), such as *lexical analysis*, *removal of stopwords*, *stemming*, *lemmatisation*.

Terms have different discriminating power, i.e., their relevance in the context where they are used. To weight term relevance, a common approach is to assign high significance to terms occurring frequently within a document, but rarely with respect to the remaining documents of the collection. The weight of a term is hence computed as a combination of its frequency within a document (*term frequency - TF*) and its rarity across the whole collection (*inverse document frequency - IDF*). We denote by $tf(w_j, m_i)$ the number of occurrences of term w_j

within message m_i, and by $df(w_j, \mathcal{M})$ the number of messages (within a given message collection $\mathcal{M}$) containing w_j. A term w_j is denoted as an *index term* for $\mathcal{M}$ if $l \leq df(w_j, \mathcal{M}) \leq u$, where l and u represent default threshold values. The ratio here is that terms appearing in a few documents, as well as terms appearing in most documents, are less significant, and hence they should be discarded.

A widely used representation model is the vector-space model (Baeza-Yates and Ribeiro-Neto, 1999). Each message m_i is represented as an m-dimensional vector $\mathbf{w}_i$, where m is the number of index terms and each component $\mathbf{w}_i[j]$ is the (normalized) *TF.IDF* weight associated with a term w_j:

$$\mathbf{w}_i[j] = \frac{tf(w_j, m_i) \cdot \log(N/df(w_j, \mathcal{M}))}{\sqrt{\sum_{p=1}^{m}[tf(w_p, m_i) \cdot \log(N/df(w_p, \mathcal{M}))]^2}}$$

After the pre-elaboration steps have been performed, many algorithms can be exploited for text clustering. For the sake of conciseness, we report here only the pseudo-code for *Lingo* algorithm as there exists an efficient implementation provided by *Carrot*[2] [4], a suite exploited in the system implementation.

As the cluster partition is obtained, the system checks if the discriminative terms that characterize the clusters may cause new dimensions to arise. To this end, a novel approach, referred as Data Posting is used, which, starting from raw data and existing ontologies, can add new dimensions induced by clustering. More in detail, the system stores query result as a materialized data cube to be exploited for further search. These data will be used as training set for further clustering refinement that will group query results in a unsupervised way. The obtained clustering will be used for extracting features relevant to the query, that have not been neither specified by the user nor considered for building the query result. As an example consider a user searching for a restaurant in Milan. S/he will type the query “restaurant in Milan" (also many search engines will suggest this statement). Traditional search results will include restaurants located in the city along with their rank. By exploiting the proposed approach, instead, the system is able to suggest users a further interesting parameter (i.e., analysis dimension) as the rank of appetizers, main courses and sweets, allowing a more focused search.

[4] http://project.carrot2.org/

Algorithm 3: The clustering Algorithm

Input: A set of text objects $\mathcal{O} = \{o_1, \ldots, o_n\}$
Output: A set of clusters $\mathcal{C} = \{c_1, \cdots, c_k\}$ of the objects in O

1: $\mathcal{C} \leftarrow \emptyset$
2: **for** $o \in \mathcal{O}$ **do**
3: $\quad$ *text filtering;*
4: $\quad$ *identify the document's language;*
5: $\quad$ *apply stemming;*
6: $\quad$ *mark stop words;*
7: **end for**
8: *discover frequent terms and phrases;*
9: *discover abstract concepts* $\mathcal{AC} = \{ac_1, \cdots, ac_j\}$;
10: **for** $ac \in \mathcal{AC}$ **do**
11: $\quad$ *find best − matching phrase;*
12: **end for**
13: *create cluster labels* $\mathcal{CL} = \{cl_1, \cdots, cl_z\}$;
14: *prune similar cluster labels in* $\mathcal{CL}$;
15: **for** $cl \in \mathcal{CL}$ **do**
16: $\quad$ *create new cluster* c_k *in* $\mathcal{C}$;
17: $\quad$ *determine the cluster contents of* c_k;
18: **end for**
19: *calculate* $\mathcal{C}$ *clusters scores;*
20: *apply cluster merging for* $\mathcal{C}$;
21: **return** $\mathcal{C}$

5.2 Data Exchange and Count Constraints

Data exchange (Fagin, Kolaitis, and Popa, 2005; Arenas et al., 2004) is the problem of migrating a data instance from a source schema to a target schema such that the materialized data on the target schema satisfies a number of given integrity constraints (mainly inclusion and functional dependencies). The integrity constraints are specified by: TGDs (Tuple Generating Dependencies), which are universal quantified formulas with additional existential quantifiers, and EGDs (Equality Generating Dependencies), which are universal quantified formulas enforcing the equality of two variables.

The classical data exchange setting is: $(S, T, \Sigma_{st}, \Sigma_t)$, where S is the source relational database schema, T is the target schema, Σ_t are dependencies on the target scheme T and Σ_{st} are source-to-target dependencies.

The dependencies in Σ_{st} map data from the source to the target schema and are TGDs, which have the following format: $\forall \mathbf{X}(\ \phi_S(\mathbf{X}) \rightarrow \exists \mathbf{Y}\ \psi_T(\mathbf{X}, \mathbf{Y})\ )$, where $\phi_S(\mathbf{X})$ and $\psi_T(\mathbf{X}, \mathbf{Y})$ are conjunctions of literals on S and T, respectively,

and $\mathbf{X}, \mathbf{Y}$ are lists of variables.

Dependencies in Σ_t specify constraints on the target schema and can be either TGDs or EGDs – the latter ones have the form $\forall \mathbf{X}(\psi_T(\mathbf{X}) \rightarrow x_1 = x_2)$, where x_1 and x_2 are variables in $\mathbf{X}$.

Example 3

As an example, consider a source schema S with three relations:

1. $\mathtt{R}(\mathtt{N},\mathtt{P})$ (*Restaurant*) with attributes N (Restaurant Name) and P (Average Price),
2. $\mathtt{P}(\mathtt{I},\mathtt{U},\mathtt{N},\mathtt{E})$ (*User Review Post*) with attributes I (Post Identifier), U (User), N (Restaurant Name) and E (Evaluation) and
3. $\mathtt{D}_{\mathtt{PEC}}(\mathtt{P},\mathtt{E},\mathtt{C})$ (*Restaurant Category*) with attributes P (Average Price), E (Evaluation) and C (Category Value).

The target schema T has two relations:

1. $\mathtt{CR}(\mathtt{N},\mathtt{C})$ (*Classified Restaurant*) with two attributes N (Restaurant Name) and C (Category Value), and
2. $\mathtt{CP}(\mathtt{I},\mathtt{U},\mathtt{N},\mathtt{C})$ (*Classified Post*) with attributes I (Post identifier), U (User), N (Restaurant Name) and C (Classified Review Evaluation).

We want that each restaurant be classified by choosing one of the evaluation reviews given for it:

$$\begin{aligned}
\Sigma_{st} = \{ & \mathtt{R}(\mathtt{n},\mathtt{p}) \rightarrow \exists\, \mathtt{C}\; \mathtt{CR}(\mathtt{n},\mathtt{C}); \\
& \mathtt{P}(\mathtt{i},\mathtt{u},\mathtt{n},\mathtt{e}) \wedge \mathtt{R}(\mathtt{n},\mathtt{p}) \wedge \mathcal{D}_{\mathtt{PEC}}(\mathtt{p},\mathtt{e},\mathtt{c}) \rightarrow \mathtt{CP}(\mathtt{i},\mathtt{u},\mathtt{n},\mathtt{c}) \} \\
\Sigma_t = \{ & \mathtt{CR}(\mathtt{n},\mathtt{c}) \rightarrow \exists\, \mathtt{I},\mathtt{U}\; \mathtt{CP}(\mathtt{I},\mathtt{U},\mathtt{n},\mathtt{c}); \\
& \mathtt{CR}(\mathtt{n},\mathtt{c}_1) \wedge \mathtt{CR}(\mathtt{n},\mathtt{c}_2) \rightarrow \mathtt{c}_1 = \mathtt{c}_2 \}
\end{aligned}$$

where all low-case letter variables are universally quantified. The constraints in Σ_{st} move restaurant names and user reviews into the target database; in addition, every review evaluation is replaced by the category value associated to the pair (average price, evaluation) by the relation $\mathtt{D}_{\mathtt{PEC}}$ and every restaurant is

classified with a value that is non specified but only declared by the existentially qualified C. The first constraint in Σ_t is a TGD and enforces that every restaurant be classified using any of the classified review evaluations issued for it. The second constraint is an EGD that admits at most one classification for a restaurant. □

The target schema typically contains some new attributes that are defined using existentially quantified variables and the main issue of Data Exchange is to reduce arbitrariness in selecting such variable values. Therefore a data exchange solution is required to be "universal" in the sense that homomorphisms exists into every possible solution, i.e., a universal solution holds a sort of "minimal arbitrariness" property. Indeed, a main goal of data exchange is to single out situations for which a universal solution exists and can be computed in polynomial time. A universal solution has the benefit that the query semantics is independent from any specific solution that may be selected as target database, so that it can support certain answers, that is, the answers that occur in the intersection of a query over all "possible" target databases. In the above example, a universal solution does not bound a restaurant classification to one of its review but it generates a new review tuple for each restaurant in order to respect the principle of "minimal arbitrariness".

In the proposed framework, the issue of "minimal arbitrariness" is not crucial for our goal, which consists in finding a "specific" solution that enriches the knowledge content of the target database instead. In the example, a specific solution can be obtained by choosing one of the issued reviews to classify a restaurant. But an arbitrary choice is not really a great achievement: the data exchange setting must provide mechanisms for making "intelligent" choices. A major step forward in this direction is to extend the data exchange setting with a new type of data dependency, called *count constraint* (an extension of cardinality constraint), first proposed in (Saccà, Serra, and Guzzo, 2012), which prescribes the result of a given count operation on a relation to be within a certain range. Count constraints use a *set term* that is either a constant set term or a *formula term*, defined as $\{\mathbf{X} : \exists \mathbf{Y} \psi\}$, where $\mathbf{X}$ and $\mathbf{Y}$ are disjoint list of variables, and ψ is conjunction of literals in which variables in $\mathbf{X}$ occur free (similar notation for set terms and aggregate predicates has been used in the dlv system (Faber et al., 2008)). There is an interpreted function symbol *count* (denoted by #) that can be applied to a set term T to return the number of tuples in T (i.e.,

the cardinality of the table represented by T).

The following example is devoted to clarify how count constraints can be used to enlarge the perspective of Data Exchange.

Example 4

Consider the data exchange problem that has been modeled in Example 3. We now enrich the criteria for restaurant classification by requiring that a category can be assigned to a restaurant only if there are at least 10 reviews supporting it. If more than one category is applicable, the one which occurs more frequently in the reviews posted by distinct users is chosen. In absence of an applicable category, a restaurant gets the classification value “NA" (not applicable).

The new mapping is defined by keeping the rules in Σ_{st} and modifying the ones in Σ_t as shown next. As usual, lower-case and upper-case letters denote variables that are respectively universally and existentially quantified – in addition, dotted letters denote free variables used for defining sets.

(1): $\mathtt{c} \neq \text{“}\mathtt{NA}\text{”} \land \mathtt{CR}(\mathtt{n},\mathtt{c}) \rightarrow \#(\{\,\ddot{\mathtt{I}} : \mathtt{CP}(\ddot{\mathtt{I}},\mathtt{U},\mathtt{n},\mathtt{c})\}) \geq 10.$

(2): $\mathtt{CR}(\mathtt{n},\text{“}\mathtt{NA}\text{”}) \land \mathtt{CP}(_,_,\mathtt{n},\mathtt{c}) \rightarrow \#(\{\,\ddot{\mathtt{I}} : \mathtt{CP}(\ddot{\mathtt{I}},\mathtt{U},\mathtt{n},\mathtt{c})\}) < 10.$

(3): $\mathtt{c} \neq \text{“}\mathtt{NA}\text{”} \land \mathtt{CR}(\mathtt{n},\mathtt{c}) \land \mathtt{CP}(_,_,\mathtt{n},\hat{\mathtt{c}}) \rightarrow \#(\{\,\ddot{\mathtt{U}} : \mathtt{CP}(\mathtt{I},\ddot{\mathtt{U}},\mathtt{n},\mathtt{c})\}) \geq$
$\#(\{\,\ddot{\mathtt{U}} : \mathtt{CP}(\mathtt{I},\ddot{\mathtt{U}},\mathtt{n},\hat{\mathtt{c}})\}).$

(4): $\mathtt{CR}(\mathtt{n},_) \rightarrow \#(\{\,\ddot{\mathtt{C}} : \mathtt{CR}(\mathtt{n},\ddot{\mathtt{C}})\}) = 1.$

All the four rules in Σ_t are count constraints. Constraint (1) states that any restaurant classification value c different from “NA" must be substantiated by at least 10 distinct users posting a review that classifies the restaurant with the value c – such reviews are collected by means of the set term with free variable $\ddot{\mathtt{I}}$. Constraint (2) states that if a restaurant is classified with “NA", then any classification posted for must violate the previous constraint. Among the applicable categories for a restaurant, the constraint (3) choices the one with highest frequency in the restaurant reviews posted by distinct users – note that in this case, as the set term is defined by the variable $\ddot{\mathtt{U}}$ rather than by $\ddot{\mathtt{I}}$, reviews posted by a same user are counted only once. Constraint (4) implements the functional dependency $\mathtt{N} \rightarrow \mathtt{C}$ in the relation $\mathtt{CR}$ so that, in case of a tie in a

restaurant classification, any of the values satisfying the constraint (3) is to be chosen.

As mentioned above, # is an interpreted function symbol for computing the cardinality of a set. We point out that existentially quantified variables are local in a set term, e.g, $\{\ddot{\mathtt{I}} : \mathtt{CP}(\ddot{\mathtt{I}}, \mathtt{U}, \mathtt{n}, \mathtt{c})\}$ stands for $\{\ddot{\mathtt{I}} : \exists \mathtt{U}\ \mathtt{CP}(\ddot{\mathtt{I}}, \mathtt{U}, \mathtt{n}, \mathtt{c})\}$; in addition, anonymous variables, denoted by underscore, are used to define a relation projection, e.g., $\mathtt{CP}(_, _, \mathtt{n}, \mathtt{c})$ stands for the projection of CP on N and C. □

The approach of using count constraints for Data Exchange has an evident drawback: the lack of a universal solution in most cases. Indeed, a universal solution is achievable only when the upper bound of a count constraints is 1, as it happens for functional dependencies. Nevertheless, as pointed out before, the goal is finding a solution that enriches data while exchanging them, rather than preserving the correspondence with the source database in order to support certain answers. More specifically, the proposed approach is aimed at enabling the selection of suitable values for existentially quantified variables, whereas the classical data exchange setting leave them undistinguished, except for the cases functional dependencies have to be satisfied. Count constraints are powerful formal tools to define "intelligent" value selection. The relationship of count constraints with value selection mechanisms has been discussed in (Saccà and Serra, 2012), where an extension of Datalog with a choice construct is presented to provide a logic programming formalization of the data exchange problem.

The introduction of count constraints allows to enrich source data with new features (i.e., additional attributes [5] reflecting properties discovered during the process of data exchange), in order to construct big data tables that can be effectively queried by end users. Thus, the new setting can be used for a new declination of data exchange for posting existing data with additional patterns so that the end user is enabled to extract additional information and knowledge from existing data while receiving suggestions and guidelines for making more comprehensive queries. This approach can be thought of

[5] Also in OLAP analysis, attributes used to highlight properties of raw data (mainly, by categorization and grouping) are called *dimensions* – we recall that an OLAP system is characterized by multidimensional data cubes that enable manipulation and analysis of data stored in a source database from multiple perspectives (see for instance (Chaudhuri and Dayal, 1997)).

as a theoretical contribution to support the so-called “faceted" navigation, described in Section 3.2.1.

We point out that data posting as the process of enriching content while publishing them is nowadays very popular in social networks – in particular is the main strength of the Twitter’s success. A tweet is a short message typically composed by a URL (a reference to existing data or news) and a number of hashtags (key words adding values to referenced data). Thus, tweeting can be thought of as a social network example of data posting.

5.3 Modeling discovered dimensions diffusion

Social network users links each other in several ways. They can simply establish a friendship relation (e.g., Facebook friends) or they can follow other users (e.g., Twitter followers). As user interactions take place, some users could become more influential than others, i.e. they are able to drive their friends (or followers) choices in many context such as food, entertainment and travels. Users influence can be also exploited for providing personalized search suggestions to their social connection. As explained above, the proposed system derives new knowledge about user search preferences by clustering. However, only a subset of the obtained dimensions can be actually considered in order to provide useful suggestions. To this end, an influence evaluation model tailored for dimension ranking is proposed such that only the most promising ones will be suggested to the network users. This ranking can be computed by exploiting the possible *influence* exerted by users to their social links. The ratio behind this choice is that, when users need to take a decision, they are largely influenced by their social environment. Moreover, users performing complex searches are usually sensitive to suggestions about related searches performed by other users. In order to clarify this statement, an example is reported.

Example 5

Consider a social network user u_1 who is searching for a good restaurant in Naples. The clustering algorithm suggests two novel dimensions, d_2 (e.g. good fish) and d_3 (e.g. good fried appetizers), that have already been exploited, respectively, by two u_1 friends, u_2 and u_3. Suppose that u_1 refines its search by choosing d_2, i.e., it searches for good fish dishes. In a sense, the latter implies that u_1 has been influenced by u_2. Based on u_1 choice it is likely that u_2

dimensions have an higher probability to be exploited by u_1 for his/her future queries. Such a dimension diffusion process is not sharp, but it can be modeled by assigning a value to the edge connecting u_1 and u_2 in the social graph. This value represents the *influence weight* that a user exerts on other user choices. Obviously, this weight is not fixed, but it varies according to the actual user searches. Indeed, as u_1 makes new inquiries, it might happen that s/he decides to follow u_3 suggestion, searching for fried appetizers, causing an increase of the probability that u_3 will influence u_1 in the future. □

As a result, identifying for each user his/her most influential friends and ranking the dimensions in decreasing order of their diffusion probability will allow to improve both users interaction with the (specialized) search engine and their query experience. To this end, in the following, the model exploited to emulate the dimensions diffusion across the network and the heuristic to compute their diffusion probability will be described. Moreover, the mathematical basis of influence evaluation that have been exploited for updating the influence weight associated to each edge of the social graph will be introduced.

5.3.1 The Algorithm for Influence Evaluation

In this section, the algorithm for modeling the diffusion of dimensions across the network will be described. Let $G = (V, E)$ be a directed graph representing a social network, where V is the set of nodes (i.e., users) and E the set of edges (i.e., social links between users). We denote with $N^{in}(v)$ (resp. $N^{out}(v)$) the set of in-neighbors (resp. out-neighbors) of an user v, i.e., $N^{in}(v) = \{u \in V | \exists(u, v) \in E\}$ (resp. $N^{out}(v) = \{u \in V | \exists(v, u) \in E\}$). Let D^v be the set of dimensions exploited by user v for issuing his/her last query, and let $\mathbf{D} = \{D^1, \cdots, D^n\}$, with $n = |V|$, be the current assignment of search dimensions for each user in the graph.

A formal mathematics model for information diffusion is firstly introduced by (Granovetter, 1978). Different groups of models types are described in (Singh et al., 2019) :

- *Threshold Models*: They use some thresholds values to differentiate behaviour prediction of user.

- *Cascading Models*: This models work in a interactive manner and are used mainly in viral marketing.
- *Epidemic Models*: Used to model various news and rumours transmission and virus infection.
- *Competitive Models*: Differently by previous, they concentrate not only on single cascade of diffusion, but on multiple cascade competition.

In order to evaluate the spread of a set of dimensions $\mathcal{D}^*$ catching their diffusion process, we consider a Threshold Type Model: the *Linear Threshold* model (LT) (Kempe, Kleinberg, and Tardos, 2003) as it fit well to collective behaviour applications and is one of the most widely-studied diffusion models. In the LT model, the diffusion process evolves in discrete time and involves a set S of initial active users (e.g., users who have exploited some suggested dimensions). Moreover, a node v is influenced by its neighbor u according to the weight $b_{u,v}$ labeling the edge between u and v. More in detail, at the beginning of the process, each node v chooses a threshold ϑ_v uniformly at random from the interval $[0, 1]$. Then, starting from the set S, each inactive node v is activated if the total weight of its active neighbors is at least ϑ_v. As the sum of in-neighbors weights for each node must not exceed 1, ϑ_v represents the weighted fraction of v's in-neighbors that must become active[6] in order to activate v. The process runs until no more activations are possible.

In order to simulate the diffusion process of $\mathcal{D}^*$ through the LT model, we need to learn its parameters. To this end, we initialize $b_{u,v}$ by applying the Bernoulli model proposed in (Goyal, Bonchi, and Lakshmanan, 2010) to the available social network activity logs. Thus, for each edge $(u, v) \in E$, the weight $b_{u,v}$ is computed as the ratio between the number of actions performed by both u and v if v's actions took place after u's, and the total number of actions performed by v. This weight is next updated by the influence evaluator every time v poses a new query as follows:

$$b_{u,v} = \begin{cases} \dfrac{|D^u \cap D^v| + b_{u,v}}{|D^u \cup D^v| + 1} & if\ |D^u| \neq 0 \\ b_{u,v} & otherwise \end{cases} \tag{5.1}$$

[6]In the application scenario, a user u is active for a given dimension d if u has exploited d for issuing its last query, i.e., $d \in D^u$

According to Eq. 5.1, $b_{u,v}$ will be updated only in the case that u exploits some of the dimensions suggested by the system. Clearly enough, the higher is the number of dimensions shared by u and v, the greater will be the increase of $b_{u,v}$. The same reasoning applies if the number of shared dimensions decrease thus causing a lower value for $b_{u,v}$ to be set. As the sum of the in-weights for a node could exceed 1, a normalization of these values dividing each $b_{u,v}$ by their sum is performed. The latter allows to accomplish the constraint imposed by the LT model.

Once assigned the weights to graph edges, the solution proposed in (Goyal, Lu, and Lakshmanan, 2011) is exploited to estimate the diffusion probabilities of $\mathcal{D}^*$. In the LT model, the probability of a path $\pi = [v_0, v_1, \ldots, v_l]$ $(l > 0)$ is defined as $Pr[\pi] = \sum_{i=0}^{l-1} b_{v_i,v_{i+1}}$. In (Goyal, Lu, and Lakshmanan, 2011), it has been shown that the overall activation probability of a node u in the LT model, given a set of initial spreader S, can be computed by summing up, for each node $s \in S$, the path probability of each path from s to u that does not cross any other node in S.

The algorithm for influence diffusion analysis is inspired by the study presented in (Goyal, Lu, and Lakshmanan, 2011). Algorithm 4 shows the proposed heuristic, namely *DIM-DIFF*, that, given a dimensions assignment $\mathbf{D}$, computes the diffusion probabilities $\mathbf{A}$ of a set of dimensions $\mathcal{D}^*$ by considering the possible paths along which they can spread across the network.

DIM-DIFF first computes the set S of initial active nodes for $\mathcal{D}^*$. Since a user can be active only for a subset d^* of dimensions in $\mathcal{D}^*$, it is necessary to keep track of this subset during the diffusion process (steps 2-6). Once built the set S, the algorithm proceeds by considering all the paths whose source nodes belongs to S and computing the path probability by multiplying the weight of the edges which compose the path. To this end, a recursive implementation of the depth first visit from an active node at a time is provided by the function *computeProbability*. The latter takes as input the last node u of the path currently considered, the set d^* of dimensions that u might propagate to its neighbors, the path probability pp and a vector which keeps track of the nodes already visited, and tries to extend the path by considering u's out-neighbors (line 14). Once the probability of the new path which includes edge (u, v) is obtained (step 15), *DIM-DIFF* updates the set of dimensions that can spread through node v by removing those for which v is already active

Algorithm 4: *DIM-DIFF*

Input: Weighted graph $G = (V, E, b)$, dimensions assignment $\mathbf{D} = \{D^1, \cdots, D^n\}$ with $n = |V|$, set $\mathcal{D}^*$ of dimensions to spread, pruning threshold δ
Output: Diffusion probability $\mathbf{A}$ for $\mathcal{D}^*$

1: $S \leftarrow \emptyset$; $\mathbf{A} \leftarrow \emptyset$
2: **for** $u \in V$ **do**
3:   **if** $\mathcal{D}^* \cap D^u \neq \emptyset$ **then**
4:     $S \leftarrow \langle u, \mathcal{D}^* \cap D^u \rangle$ *// identifies nodes that are active for at least one dimension in* $\mathcal{D}^*$
5:   **end if**
6: **end for**
7: **while** $S \neq \emptyset$ **do**
8:   $\langle u, d^* \rangle \leftarrow S.extract()$
9:   $visited\,[u] = true$
10:   $computeProbability(u, d^*, 1, visited)$ *// starts the depth first visit from each active node*
11: **end while**
12: **return** $\mathbf{A}$;

13: **Function** ***computeProbability***$(u, d^*, pp, visited)$
14: **for** $v \in N^{out}(u) \wedge\, !visited\,[v]$ **do**
15:   $pp \leftarrow pp \times b_{u,v}$ *// updates the path probability*
16:   $d^* \leftarrow d^* \setminus D^v$ *// removes the dimensions node v is currently activated for*
17:   **if** $d^* \neq \emptyset \ \wedge \ pp \geq \delta$ **then**
18:     $A^v \leftarrow \mathbf{A}\,[v]$
19:     **for** $d \in d^*$ **do**
20:       $A^v(d) = A^v(d) + pp$ *// updates the diffusion probability of dimension d for node v*
21:     **end for**
22:     $visited\,[v] = true$
23:     $computeProbability(v, d^*, pp, visited)$
24:     $visited\,[v] = false$
25:   **end if**
26: **end for**

(step 16). By step 20, the diffusion probability of the dimensions not yet used by node v is updated. Clearly, since the path probability rapidly diminishes as the path gets longer, a threshold δ is used to consider only the contributions of paths with a reasonable probability (step 17).

To better understand how the algorithm works, an execution trace will be shown. Consider the graph in Figure 5.3 (left), with nodes identifying users of a social network and the current influence weight for each edge highlighted

in blue. Users a, b and e have already exploited some of the dimensions suggested by the system (see Figure 5.3 (right) that depicts the current dimension assignment **D**). Assume we want to evaluate the diffusion process for $\mathcal{D}^* = \{d_1, d_3\}$ when $\delta = 0$. Thus, the set S consists of the nodes highlighted in red, i.e., a and e.

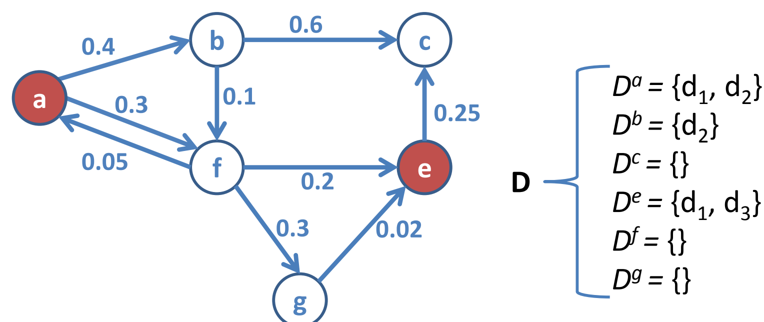


FIGURE 5.3: Diffusion Graph (left) and Dimensions Assignment (right)

Once defined the set S of initial spreaders, the algorithm starts visiting the graph from each active node. Figure 5.4 (left) highlights in red the edges considered by *DIM-DIFF* during the visit from node a. The box beside the reached nodes contains the diffusion probability of $\mathcal{D}^*$ to that node. As node a is active only for dimension d_1, each box contains only the values for dimension d_1. Moreover, it is worth noting that, despite there exist several paths connecting a to e, these are never considered since e is already active for dimension d_1. In next steps, *DIM-DIFF* repeats the visit starting from node e (Figure 5.4 (right)), which is active for both dimensions in $\mathcal{D}^*$, causing the update of the diffusion probability for node c.

The algorithm described above can be used for defining a strategy for dimension recommendation based on the potential influence that can be excerpted by a user. We recall that, in this system, dimensions can arise, disappear and/or reappear according to the users needs. Thus, main interest is not focused in maximizing the spread of a dimension in the network. However, the system can be easily exploited to detect the most influential users, as it allows to keep track of users who lead to the largest cascade of dimension diffusion. In the experimental sections, we will show that the algorithm exhibits good performances in real scenarios.

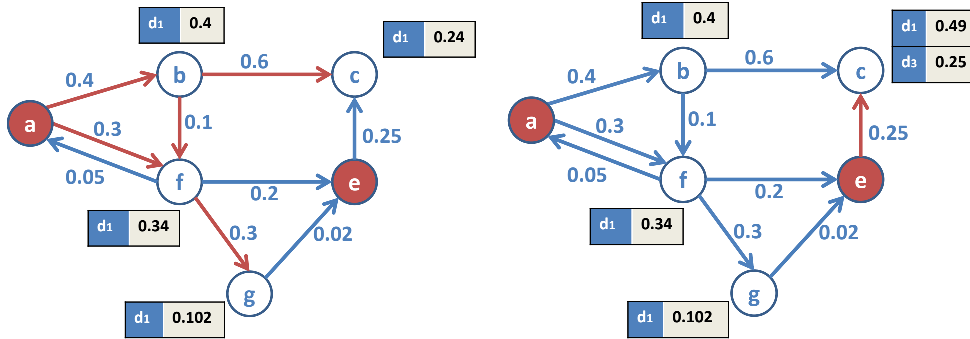


FIGURE 5.4: Diffusion process of Dimensions d_1 and d_3 Starting From Node *a* (left) and From Node *e* (right)

5.4 System Architecture

In this section, the overall architecture of the system for big data search will be described. This architecture exploits the techniques described above and sketched in Figure 5.2. Main goal is to provide users a flexible tool for assisted text search, that is interactive, scalable and dynamic. To this end, several indexing and data management strategies are exploited, which allow us to cope with high volume, heterogeneous and burst information. Figure 5.5 shows the system architecture describing in detail each high level module depicted in Figure 5.2.

Proposed architecture is based on the *Sigma Architecture* described in section 3.2.2 where the Layers have been renamed to match the overall framework terminology as showed in Figure 5.2. As users interact with this system, new data are collected by the storage layer. Based on data arrival rate, the system schedules offline clustering of the whole dataset (e.g. after a burst of 1.000.000 tuples is collected). The cluster partitioning is then refined by evaluating the potential influence diffusion of the emerging dimensions by exploiting influence evaluator described in Section 5.3. Finally, a data posting task is executed to enrich the dimensions migrating data from a source schema of clustering results to the target schema of HBase storage (as reported in section 5.2). These new dimensions can be fruitfully exploited for guiding users through an assisted search task.

For Presentation Layer two customized views is provided for different user categories, one for *end-users* and the other for *domain experts*. In particular,

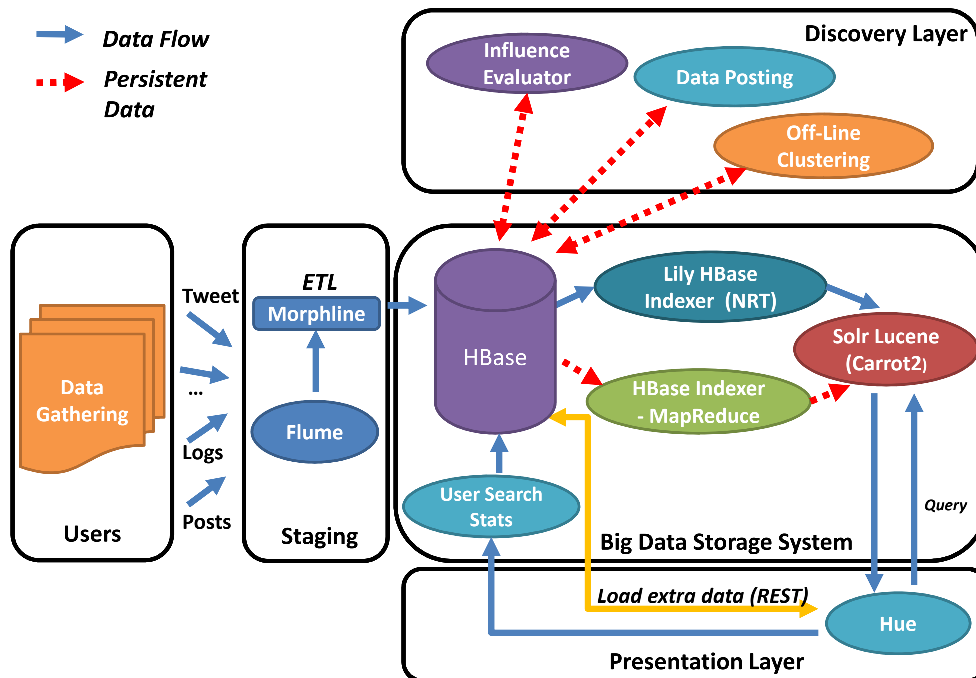


FIGURE 5.5: System Architecture for Big Data Search

end-users are able to search only data indexed by *Solr*, while domain experts can also view/query data available in the data warehouse (including those automatically detected by the system and suggested by the influence diffusion and data posting tasks) and (eventually) add new dimensions. The rationale behind this choice is the following: end-users are interested to data pertaining their own search, thus they are not interested in additional details. On the opposite side, domain experts need a comprehensive view of the interactions taking place among users. We recall that, as end-users interact with the system, usage statistics are collected and stored for better evaluating how dimensions spread among users.

Finally, for the sake of efficiency, the system prevent the reversal of the entire data warehouse by *Solr*. More in detail, there is a distinction between indexed data used to search for documents and data stored on *HBase*. The latter can be accessed directly, avoiding the additional overhead due to raw data access. Moreover, in order to improve system performances, the minimum amount of information is maintained on *Solr* index and the access to the

complete information is allowed by REST API service[7].

5.5 Experimental results

In order to validate the system, several experiments have been performed on a real dataset. Some anonymized tourist data, with their comments and search logs, have been provided by ACI Informatica. The overall dataset consists of 1.000.000.000 tuples, most of them obtained by crawling additional data from Yelp, Twitter and Facebook. More in detail, the system works with three kinds of tuples:

- **Action tuples.** Posts and comments from the same user have been collected in a single tuple in order to exploit the column family store features. Each action tuple has the form $\langle id_u, C, DL \rangle$, where:

  - id_u is a user identifier;
  - C is a pair $\langle SN, count \rangle$, where SN is the social network from which this tuple has been collected and *count* is the number of interactions the user has with SN (e.g. the number of posts or comments);
  - DL is a set of triples $\langle c_i d, content, \tau \rangle$, where $c_i d$ is an object identifier, *content* is the object text (objects here can be either comments, posts or suggestions) and τ is the post timestamp;

- **Graph tuples.** These tuples keep track of the user connections. Indeed, each graph tuple consists of two users identifiers and a numerical value representing the influence weight. For data retrieved by Facebook and Twitter the connections among users can be easily derived through friendship and followers/followee relationships. As regards Yelp, two users are considered as connected if it posted a comment on the same company (e.g. a restaurant). As concern the influence weight, its initial value is set according to the Bernoulli model presented in (Lu et al., 2013), which is based on the number of actions performed by the users.

- **Dimension tuples.** The system keep track for each user who poses queries of a *dimension tuple* which has the form $\langle id_u, D^{id_u}, A^{id_u} \rangle$, where:

[7]This feature is implemented by HBase Rest Server

– id_u is the user identifier;
– D^{id_u} is a list of pairs $\langle d, t \rangle$ where d is the dimension being used/exploited by user id_u and t is the timestamp of the last use of this dimension;
– A^{id_u} is a list of pair $\langle$ *dimension*, $pp\rangle$ containing the output of the *DIM-DIFF* algorithm for that user;

In what follows, the detailed description of data gathering strategy is provided.

5.5.1 Data Gathering.

In order to perform an effective data analysis, data are collected by implementing some wrappers specifically tailored for tourist data scenario. The crawling strategy have been designed by exploiting some basic tourist information obtained by ACI Informatica.

Step 1: Yelp. The augmentation strategy starts from *Yelp*. Data regarding companies whose activity is of potential interest for tourists (e.g. restaurants and bars) and users posting information about them have been downloaded from Yelp. In order to implement a proper querying strategy that complies with the social network policy, several accounts have been created and the API provided by Yelp, namely *search API* and *Business API* have been used. The search API allows to issue REST requests to the web site for retrieving information based on geographical location of the companies, their category and meaningful keywords. Results are ordered by their relevance w.r.t. the issued query and their ranking. In principle, it is also possible to identify companies offering special discounts but we did not use this feature as it is not relevant for our purposes. Each query corresponds to 40 results, thus the requests were iterated till the maximum per day limit [8]. It was chosen to search for Italian companies by iterating the requests on the Italian city list and Yelp category list (e.g.the following request was issued for each city and category: *http://api.yelp.com/v2/search?category_filter='Category X'&location='city Y')*. For each collected companies id, an excerpt of user reviews about each company (and the corresponding user id) has been collected As we got the companies id, we collected an excerpt of user by querying the

[8]The maximum number of requests per day is 10.000

business API[9]. The reviews pertaining a company consist of a JSON object, which is cleaned and sent to the Hbase datastore.

Step 2: Twitter. Once data extraction task from Yelp has been completed, additional data by Twitter has been added. Also in this case several user accounts have been used. In order to work properly, Twitter Search API requires: 1) a correct identification of the user that is requesting the data; 2) the application that is performing the data access and 3) the signature for user authorization. The API allows to gather information about tweets published in the last 7 days. The extraction procedure exploits the open source library Twitter4J [10]. The crawling strategy exploits the company names and locations previously collected in order to retrieve tweets pertaining to them. Also Twitter API poses some access restriction (e.g. 15 minutes time windows and a maximum number of 180 requests for each window) thus it was necessary to iterate the strategy according to these constraints.

Step 3: Facebook. Finally, additional information from Facebook have been collected. The core of Facebook data structures is the *Social Graph* that stores social network users and their interactions (represented as labeled nodes and edges). The graph can be queried by *Graph API* which allows to gather both single node information [11] and node interactions (e.g. friends, posts, tags) [12] once you get an *Access Token*[13]. The extraction procedure exploits the open source library Facebook4J[14] which allows to search and download Facebook contents that are matched with company name (and/or city) picked from the list built in previous steps[15]. For each query, we get up to 500 public posts. As in previous steps, there exist time window limitations, thus several iterations have been performed.

---

[9] The request process must be performed by secure authentication protocol OAuth 1.0a, xAuth

[10] http://twitter4j.org

[11] https://graph.facebook.com/$< id >$

[12] https://graph.facebook.com/$< id >$/$< connection_type >$

[13] Facebook exploits OAuth 2.0 protocol

[14] https://facebook4j.github.io

[15] In case of ambiguity due to the company name we discard the data as we are not able to automatically resolve it

5.5.2 Experimental Setup.

As concern the system response time, we analyze the impact of the data shard size on the proposed indexing strategy and the performance of read and write operations. Experiments were executed on *ALEPH* cluster having 12 computing nodes (24GB RAM each), 1 front-end node and a GPU (NVIDIA Tesla).

5.5.3 System Scalability Evaluation

Evaluating the performance of a system exploiting Big Data techniques is a non-trivial task as we need to address several issues. Moreover, it requires ad-hoc approaches due to the complexity of the architectures and the interactions of Big Data oriented environments. This section will be devoted to describe the strategy for evaluating the performance of the system from several points of view.

Map Reduce Efficiency

In the proposed system, data arrive in form of data shards and they need to be indexed for later analysis as described in Section 5.4. The key tasks at this stage are the *Map* and *Reduce* operations. The system collects data continuously, thus it is necessary to measure the performance of these tasks w.r.t. the number of shards to be processed and their size. Two set of experiments were performed:

- The number of posts for each shard has been fixed while keeping constant their arrival rate;
- The arrival rate of posts has been varied for each shard size

Several experiments have been performed, and in order to evaluate the efficiency of the proposed approach, we report here the *worst case* performances as they allow to appreciate the high efficiency of this framework even in a "stressing" condition. More in detail, Figure 5.6 shows the execution times obtained by fixing the arrival rate to 10.000 post per minute as it is the maximum arrival rate we tested which causes the slower execution times.

It is worth noticing that, the execution times reach a minimum when the number of tuples is in the range [50.000.000, 100.000.000]. After that minimum, the execution times are almost constant. Indeed, a low number of tuples causes

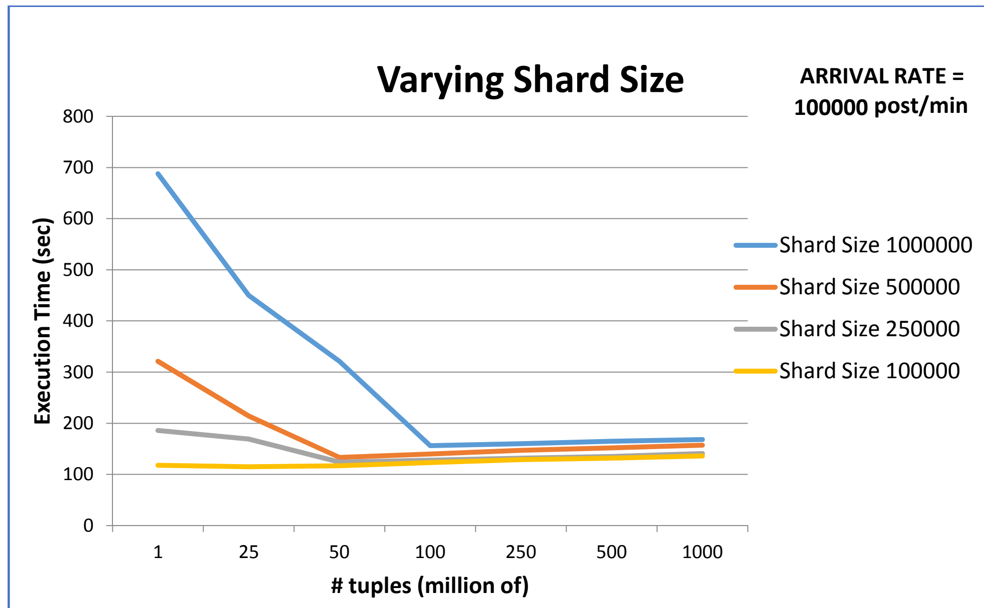


FIGURE 5.6: Execution Time vs. Data Shard Size

a large response time due to the high work that the few shards have to perform to complete the map operations. When the number of tuples is higher than the one that exhibits the minimum value, the slight increase in execution times is due to the latency for waiting all the nodes finish their task.

For the sake of completeness, Figure 5.7 shows the results obtained by varying the arrival time (i.e. post per minute) while keeping fixed the data shard size. Analogously with the results reported before, we report the results obtained for the data shard size causing the slower execution times.

Also in this case, it is possible to observe the same trend for execution times (disregarding the arrival rate) and best performances are obtained for lower arrival frequencies. The latter is caused by the increased complexity of the reduce operations when the arrival rate increases.

Read and Write Efficiency

An important parameter for assessing the efficiency of the proposed system is the *read* and *write* performance. Several test were performed, by varying the number of tuples in the Hbase storage system. For the sake of readability, we do not plot the results as they present a constant trend both for read operations

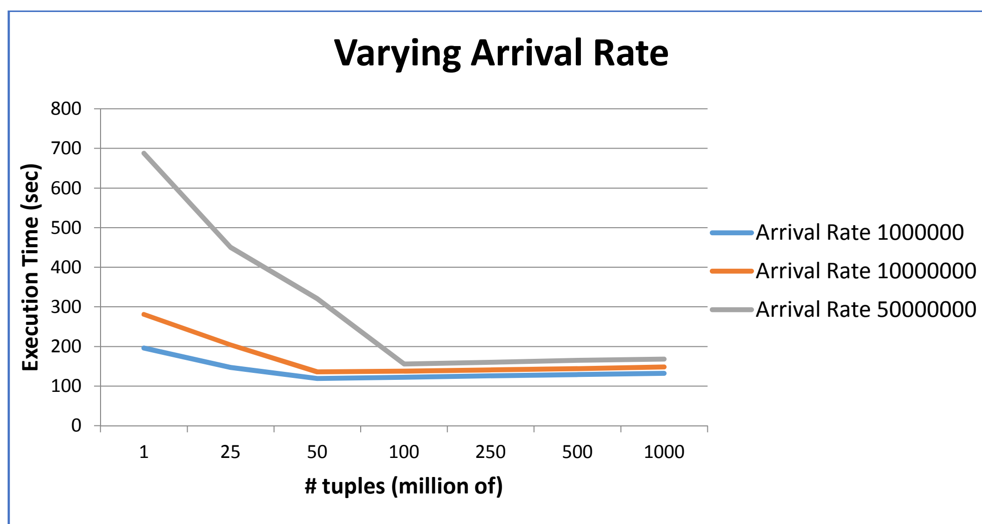


FIGURE 5.7: Execution Time vs. Arrival Rate

(1 msec average independently of the number of tuples in the data store) and write operations (6 msec average independently of the number of tuples in the data store). These results are quite interesting from several viewpoints: 1) The read operations over the HBase secondary index are performed in the order of a millisecond. This implies that the proposed knowledge discovery and enrichment algorithms can run at minimum latency, thus linking new posts on the fly without undergoing any delay that may worsen the system real-time functionalities; 2) The execution time of the read operations is stable and does not increase as the number of posts grew. This is crucial for the proposed framework, as the system is capable to produce timely answers for such high volumes of data as reported below. Table 5.1 reports the execution times obtained for dynamic searches using a varying number of keywords. For each table row it is reported the number of keywords composing the query and the time elapsed for producing the results matching the query. More in detail, for each number of keywords we performed 100 queries and in table it is reported the average response time.

The results reported in Table 5.1 is quite intriguing as they imply that the system is capable to provide really fast answers. No results are reported for queries composed by more than 5 keywords for two motivations: 1) they are quite uncommon; 2) the results are almost the same as for 5 keywords.

TABLE 5.1: Response Time vs. Number of Keywords

#TotalKeywords	*ResponseTimes* (sec)
1	< 1
2	< 1
3	1.23
4	1.41
5	1.45

Chapter 6

Conclusions

The impressive progress and development of Internet and on-line technologies has led to an increasing availability of a huge volume of data generated by heterogeneous sources at high production rates. The availability of such unprecedented large amount of information sources represents an opportunity for the analysis of human behavior, and their evolution when influenced by other people's opinion/suggestion. This type of analysis requires on one hand the availability of adequate systems for storing large amounts of data and on the other hand the availability of high performance computing systems with huge potential calculation. The main question that we attempted to answer with this work is, basically, to what extent classic Big Data Tools to analyze and extract complex and useful information in an effective way and how to design an effective high performance P2P network for resource sharing.

To this end, a user behaviour oriented search framework which implements a new generation of IIS and an hybrid peer to peer architecture for computational resource sharing named Coremuniti^{TM} have been presented. Moreover, an introduction to Big Data and NoSQL has been provided and two basic architecture for Big Data analysis have been illustrated. The first architecture, named Sigma, describes a solution for building a complete, interactive and scalable Big Data system and an implementation of it, able to offer a full-text search mechanism, has been presented. To design and develop a similar tool it was necessary to use complex data indexing and management exploiting the potential of some open source projects. The second architecture is based on an OLAP approach using the Pentaho BI Suite. In this architecture the basic data warehouse services provided by Pentaho have been extended using the Map Reduce features provided by the Hive project.

The presented user behavior oriented search framework implementing new generation IIS is based on Sigma architecture and is organized into three layers:

1. *Big Data Storage System*, which (i) computes some high level statistics about user search (e.g. most frequent search keywords) that will be refined during the information analysis and enrichment steps, (ii) stores all the collected data in a staging area to overcome potential problems due to different speed, size and format of incoming data, (iii) performs the proper extraction, transformation and loading operations for making data well suited for efficient storage (space saving) and analysis (fast execution time) and (iv) organize the pre-processed data in a structure tailored for Big Data;

2. *Discovery Layer*, which includes three modules: (i) *Clustering* devoted to the extraction of unsupervised information hidden in the collected data by means a suite of clustering algorithms, (ii) *Influence Evaluator* that filters the obtained clusters to identify properties (dimensions) of potential interest for users and evaluates how they spread across the network and (iii) *Data Posting* extending classical data exchange so that, while moving data, the contents are enriched by supplying additional pieces of information.

3. *Presentation Layer* is used to personalize the enriched information for the current user query and to present portions of it (dimensions) into the search toolbar that are pertinent with entered keywords for enabling a faceted browsing.

A prototype implementation of the search framework has been described and the various design solutions adopted for offering advanced search functionalities have been illustrated in detail. Results on the usability, efficiency and effectiveness of the system were reported as well and they confirmed the validity of the overall approach. The application scenario used for experiments was tourism recommendation. The ratio behind the choice of this scenario was that, users (i.e., tourists) do an extensive search activity when traveling and they post comments and suggestions, thus contributing to the definition of their behavior and needs and to the construction of a background knowledge useful to assist them in further searches.

As main future direction of this research line, some possible ideas are the following:

- Testing the approach on a wider set of social networks and application contexts (e.g., viral marketing, food education sentiment analysis and epidemic modeling)
- Applying more sophisticated diffusion models and influence evaluation algorithms

Moreover, in this work, an hybrid peer to peer architecture for computational resource sharing has been proposed. Users joining this network are able to share their unexploited computational resources and are rewarded by tangible credits. The computational power shared is used to solve difficult task submitted by other users. In order to guarantee the efficiency and effectiveness of the computation process, a task partitioning and assignment algorithm has been provided which reduces the execution times while allowing satisfactory revenues for resource providers. Extensive experiments were conducted in a real life scenario such as 3D rendering to assess the system performances. As shown in the experimental evaluation, the Coremuniti approach is able to improve the use of efficient digital solutions in "traditional" field like engineering and architectural design, by offering a new high speed and low cost tool for increasing their productivity.

As future work, it will be useful to develop specialized plugins for other challenging application fields such as medical and insurance simulation. Furthermore, we are in the process of evaluating the energy consumption achievable by leveraging the Coremuniti network. More in detail, the ICT industry currently accounts for about 2% of global emissions of carbon dioxide (CO2). Many international organizations recommends the sector to reduce the carbon footprint in next years. Coremuniti shows a possible way to reduce the impact of computing intensive processes by a P2P network that can induce a significant energy saving (as we do not need to power additional resources but better use *already* powered ones) compared to processes running on a single node or server farm. This reduction in energy consumption will reduce the pollution emission, characterizing our solution as an environment friendly solution for high performance computing.

Bibliography

Agrawal et al., D. (2012). “Challenges and Opportunities with Big Data. A community white paper developed by leading researchers across the United States”. In:

Arenas, Marcelo et al. (2004). “Locally Consistent Transformations and Query Answering in Data Exchange”. In: *PODS*. Ed. by Catriel Beeri and Alin Deutsch. ACM, pp. 229–240. ISBN: 1-58113-858-X.

Baeza-Yates, R. and B. Ribeiro-Neto (1999). *Modern Information Retrieval*. ACM Press Books. Addison Wesley.

Bhatia, Rashmi (2013). “Grid Computing and Security Issues”. In: *International Journal of Scientific and Research Publications (IJSRP)* 3, Issue 8, August 2013 Edition. URL: http://www.ijsrp.org/research-paper-0813/ijsrp-p2094.pdf.

BOINC website. https://boinc.berkeley.edu. Accessed: 2018-02-22.

Brewer, Eric (Jan. 2000). “Towards robust distributed systems”. In: p. 7. DOI: 10.1145/343477.343502.

C. Strozzi. Nosql relational database management system. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/HomePage. Accessed: 2019-08-22.

Chang, Fay et al. (June 2008). “Bigtable: A Distributed Storage System for Structured Data”. In: *ACM Trans. Comput. Syst.* 26.2, 4:1–4:26. ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: http://doi.acm.org/10.1145/1365815.1365816.

Chaudhuri, Surajit and Umeshwar Dayal (1997). “An Overview of Data Warehousing and OLAP Technology”. In: *SIGMOD Record* 26.1, pp. 65–74.

Dean, Jeffrey and Sanjay Ghemawat (2004). “MapReduce: Simplified Data Processing on Large Clusters”. In: *OSDI’04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, pp. 137–150.

Economist, The (2010). “Data, data everywhere”. In: *The Economist*.

Faber, Wolfgang et al. (2008). "Design and implementation of aggregate functions in the DLV system". In: *TPLP* 8.5-6, pp. 545–580.

Fagin, Ronald, Phokion G. Kolaitis, and Lucian Popa (2005). "Data Exchange: getting to the core". In: *ACM Trans. Database Syst.* 30.1, pp. 174–210.

Firdhous, Mohamed (2012). "Implementation of Security in Distributed Systems - A Comparative Study". In: *CoRR* abs/1211.2032. URL: `http://arxiv.org/abs/1211.2032`.

Franklin, Michael J. et al. (2011). "CrowdDB: Answering Queries with Crowdsourcing". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pp. 61–72.

Glassner, Andrew S (1989). *An introduction to ray tracing*. Elsevier.

Goyal, Amit, Francesco Bonchi, and Laks V. S. Lakshmanan (2010). "Learning influence probabilities in social networks". In: *Proceedings of the Third International Conference on Web Search and Web Data Mining, WSDM 2010, New York, NY, USA, February 4-6, 2010*, pp. 241–250.

Goyal, Amit, Wei Lu, and Laks V. S. Lakshmanan (2011). "SIMPATH: An Efficient Algorithm for Influence Maximization under the Linear Threshold Model". In: *11th IEEE International Conference on Data Mining, ICDM 2011, Vancouver, BC, Canada, December 11-14, 2011*, pp. 211–220.

Granovetter, Mark (1978). "Threshold Models of Collective Behavior". In: *American Journal of Sociology* 83.6, pp. 1420–1443. DOI: `10.1086/226707`. eprint: `https://doi.org/10.1086/226707`. URL: `https://doi.org/10.1086/226707`.

Harris, Mark (2008). "Many-core GPU Computing with NVIDIA CUDA". In: *Proceedings of the 22Nd Annual International Conference on Supercomputing*. ICS '08.

Kempe, David, Jon M. Kleinberg, and Éva Tardos (2003). "Maximizing the spread of influence through a social network". In: *KDD*, pp. 137–146.

Kiran, Mariam et al. (Oct. 2015). "Lambda architecture for cost-effective batch and speed big data processing". In: pp. 2785–2792. DOI: `10.1109/BigData.2015.7364082`.

Kreps, Jay (2014). "Questioning the lambda architecture." In: *Online article*.

Li, Junjie, Sanjay Ranka, and Sartaj Sahni (2011). "Strassen's matrix multiplication on GPUs". In: *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. IEEE, pp. 157–164.

Liu, Xiufeng, Nadeem Iftikhar, and Xike Xie (2014). “Survey of Real-Time Processing Systems for Big Data”. In: *Proceedings of the 18th International Database Engineering Applications Symposium*. IDEAS ’14. Porto, Portugal: Association for Computing Machinery, 356–361. ISBN: 9781450326278. DOI: 10.1145/2628194.2628251. URL: https://doi.org/10.1145/2628194.2628251.

Lu, Wei et al. (2013). “The bang for the buck: fair competitive viral marketing from the host perspective”. In: *KDD*, pp. 928–936.

Manogaran, Gunasekaran and Daphne Lopez (Jan. 2017). “A survey of big data architectures and machine learning algorithms in healthcare”. In: *International Journal of Biomedical Engineering and Technology* 25, p. 182. DOI: 10.1504/IJBET.2017.087722.

Manyika, J. et al. (2011). “Big data: The next frontier for innovation, competition, and productivity”. In: *McKinsey Global Institute*.

Marz, Nathan and James Warren (2015). “Big Data: Principles and Best Practices of Scalable Realtime Data Systems.” In: ISBN: 9781617290343.

Moens, M. F. (2000). *Automatic Indexing and Abstracting of Document Texts*. Kluwer Academic Publishers.

Nakamoto, S. (2008). “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: *Freely available on the web*.

Narayanan, A. and V. Shmatikov (2008). “Robust De-anonymization of Large Sparse Datasets”. In: *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. SP ’08. Washington, DC, USA: IEEE Computer Society, pp. 111–125. ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.33. URL: http://dx.doi.org/10.1109/SP.2008.33.

Nature (Sept. 2008). “Big Data”. In: *Nature*.

Newman, Mark (2010). *Networks: An Introduction*. New York, NY, USA: Oxford University Press, Inc.

Nickolls, John et al. (Mar. 2008). “Scalable Parallel Programming with CUDA”. In: *Queue* 6.2.

Osinski, Stanislaw and Dawid Weiss (2005). “A Concept-Driven Algorithm for Clustering Search Results”. In: *IEEE Intelligent Systems* 20.3, pp. 48–54. DOI: 10.1109/MIS.2005.38. URL: https://doi.org/10.1109/MIS.2005.38.

Payer, M., T. Hartmann, and T. R. Gross (2012). "Safe Loading - A Foundation for Secure Execution of Untrusted Programs". In: *2012 IEEE Symposium on Security and Privacy*, pp. 18–32. DOI: 10.1109/SP.2012.11.

Peng, Dai (2015). "Hybrid Crowdsourcing Platform". In: *United States Patent Application 20150178134*.

Persico, Valerio et al. (2018). "Benchmarking big data architectures for social networks data processing using public cloud platforms". In: *Future Generation Computer Systems* 89, pp. 98 –109. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.05.068. URL: http://www.sciencedirect.com/science/article/pii/S0167739X17328303.

Saccà, Domenico and Edoardo Serra (2012). "Data Exchange in Datalog Is Mainly a Matter of Choice". In: *Datalog*. Ed. by Pablo Barceló and Reinhard Pichler. Vol. 7494. Lecture Notes in Computer Science. Springer, pp. 153–164. ISBN: 978-3-642-32924-1.

Saccà, Domenico, Edoardo Serra, and Antonella Guzzo (2012). "Count Constraints and the Inverse OLAP Problem: Definition, Complexity and a Step toward Aggregate Data Exchange". In: *FoIKS*. Ed. by Thomas Lukasiewicz and Attila Sali. Vol. 7153. Lecture Notes in Computer Science. Springer, pp. 352–369. ISBN: 978-3-642-28471-7.

Singh, Shashank et al. (Jan. 2019). "A Survey on Information Diffusion Models in Social Networks". In: pp. 426–439. ISBN: 978-981-13-3142-8. DOI: 10.1007/978-981-13-3143-5_35.

Tang, Youze, Xiaokui Xiao, and Yanchen Shi (2014). "Influence maximization: near-optimal time complexity meets practical efficiency". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pp. 75–86.

Thota C., Manogaran G. Lopez D. and V. Vijayakumar (2016). "Big data security framework for distributed cloud data centers, Cybersecurity Breaches and Issues Surrounding Online Threat Protection". In: IGI Global, Hershey, PA, 288–310.

Witzel, RF, RW Burnham, and JW Onley (1973). "Threshold and suprathreshold perceptual color differences". In: *JOSA* 63.5, pp. 615–625.

Wood, Gavin et al. (2014). "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014, pp. 1–32.

Yang, Beverly and Hector Garcia-Molina (2001). "Comparing Hybrid Peer-to-Peer Systems". In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 561–570.

Yang, Min and Yuanyuan Yang (2010). "An Efficient Hybrid Peer-to-Peer System for Distributed Data Sharing". In: *IEEE Transaction on Computer* 59.9, pp. 1158–1171.

Zamir, Oren and Oren Etzioni (1998). "Web Document Clustering: A Feasibility Demonstration". In: pp. 46–54.