Query engine for massive distributed ontologies using MapReduce

---

Submitted by

Juan Esteban Maya Alvarez

maya.juan@googlemail.com

Information and Media Technologies

Matriculation Number

20729239

Supervised by

Prof. Dr. Ralf Moeller

Sebastian Wandelt

Hamburg, Germany

July 6, 2010

Acknowledgements

I would like to thank all the persons who helped me during the last past 6 months of work. To all the people who gave me their support and patience. Particularly I want to thank Sebastian Wandelt, who helped me through all the development of this thesis. Prof. Dr. Ralf Moeller for all the passionate lectures that put the seed of Description Logics on my education and for giving me the chance to work on this topic and Prof. Dr. Weberpals for its interest and accepting to be second examiner of this thesis. I would also like to thank The Public Group LLC and its IT manager, Luis Londoño, who allowed me to use their infrastructure to run the benchmarks required for this project. To my family for their valuable support even when they are thousands of kilometers away from me, to my friends for having the patience to deal with my days of bad mood and last, but not least, all my gratitude goes to my girlfriend Gabriela, who gave me her smile and moral support when I most needed it.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, July 4, 2010
Juan Esteban Maya A

Contents

List of Figures

List of Tables

Abstract

Due to the massification of the Semantic Web and the use of Description Logics algorithms to perform reasoning services in its data, there is a growing need to create architectures and algorithms that support practical TBox and ABox Reasoning. During the past years the DL community has focused its research on creating algorithms that support reasoning services on very expressive DL Languages, however, many times, when applied to real world applications, the algorithms don't scale so well to the needs of the Semantic Web.

The objective of this project is to propose an architecture that exploits the progress in the reasoning of very expressive languages but at the same time allow applications to scale to the proportions required by the Semantic Web. The proposed architecture takes the reasoning services provided by existent DL Reasoners and brings them to a distributed environment. The project focuses on describing the architecture and techniques used to support query answering in very large ABoxes without losing expressivity power.

Chapter 1

Introduction

The Semantic Web was created thinking about the next generation of the Web. Its application benefits from the combinations of expressive description logics (DL) and Databases; description logics are useful to structure and represent knowledge in terms of concepts and roles, but the reasoning procedures are currently not scalable for answering queries based on large scale data sets. Databases, on the other hand, are efficient to manage data, however, when thinking about massive amounts of data, even the most robust database systems fall short in terms of storage needs. This is not so difficult to imagine when we think about the space requirements and infrastructure of the search engine giants like Google or Yahoo or in general, companies whose business model is based on collecting and analyzing data face this problem, some examples are Facebook or Twitter.

Given the increasing need for a practical storage and infrastructure to process vast amounts of data, Google has come with a group of technologies that allow not only to store the data efficiently but also to process the data in a scalable way. In 2004, with [4] Google introduced for the first time the MapReduce programming model to process and generate large data sets. Because traditional databases were falling short to store all the data required by Google applications, BigTable was introduced in 2006 [5]. BigTable is a distributed storage system for managing structured data that is designed to scale to a very large size across thousands of commodity servers.

The features of description logics (DL) and databases have led researches to find techniques that could exploit the semantic representation and powerful reasoning services of DL as well as the efficient management and accessibility of databases. In 1993, the approach of loading data into description logic reasoners was investigated in [1]. [2]

extends the traditional DL ABox with a DBox so that users can make queries without being concerned about whether a database or Knowledge Base has to be accessed. [3] is focused on domains with massive data, where a large amount of related individuals exist.

It's safe to imagine that the next evolutionary step in the Semantic Web is to come with mechanisms that will allow to have expressive language reasoning in distributed environments containing vast amount of information. A few approaches have been proposed. In [6] the authors present an algorithm to reason over RDFS in top of DHTs[1] and in [8] MapReduce is used to execute RDFS reasoning with success. Yahoo has been also actively trying to incorporate the Semantic Web in its product portfolio[2]. Yahoo's interest produced a public paper [7] that explores MapReduce and related technologies to execute SPARQL queries in large distributed RDF triple stores.

In this thesis I will address the architectural problems needed to answer DL queries over large amount of data using a distributed environment that runs in commodity hardware, specifically using the MapReduce programming model [4] offered by the open source framework Hadoop. At the same time, techniques to improve the overall performance of the query engine will be used that make use of the progress done in the integration of DL and distributed persistence mechanisms.

1.1 Outline

The chapters in this thesis are organized as follows. Chapter 2 introduces Hadoop, MapReduce and the technologies around them. Chapter 3 introduces the Semantic Web, basic concepts of Description Logics, its algorithms and mechanisms to improve the performance of query answering. Chapter 4 proposes an architecture to perform DL Queries in a distributed environments. Chapter 5 presents the results obtained with the performance analysis reports. Finally, chapter 6, presents the conclusions and possible extensions to follow this work.

[1] Distributed Hash Tables

[2] SearchMonkey: `http://developer.yahoo.com/searchmonkey/`

Chapter 2

Hadoop and Map Reduce

This chapter introduces the frameworks and technologies around MapReduce required to build scalable applications that support vast amounts of data.

2.1 HADOOP

Hadoop is a project from the Apache Software Foundation written in Java and created by Doug Cutting[1] to support data intensive distributed applications. Hadoop enables applications to work with thousands of nodes and petabytes of data. The inspiration comes from Google's MapReduce[4] and Google File System[12] papers. Hadoop's biggest contributor has been the search giant Yahoo, where Hadoop is extensively used across the business platform.

Hadoop is an umbrella of sub-projects around distributed computing and although is best known for being a runtime environment for MapReduce programs and its distributed filesystem HDFS, the other sub-projects provide complementary services and higher level abstractions. Some of the current sub-projects are:

- **Core:** The Hadoop core consist of a set of components and interfaces which provides access to the distributed filesystems and general I/O (Serialization, Java RPC, Persistent data structures). The core components also provide "Rack Awareness", an optimization which takes into account the geographic clustering of servers, minimizing network traffic between servers in different geographic clusters. The distributed file system is designed to scale to petabytes of storage running on top of the filesystem of the underlying operating systems.

[1] Creator of Apache Lucene, the widely used text search library.

- **MapReduce:** Hadoop MapReduce is a programming model and software framework for writing applications that rapidly process vast amounts of data in parallel on large clusters of computer nodes. Section 2.2 introduces MapReduce with more detail.

- **HDFS:** Hadoop Distributed File System (HDFS) is the primary storage system used by Hadoop applications. HDFS is, as its name implies, a distributed file system that provides high throughput access to application data creating multiple replicas of data blocks and distributing them on compute nodes throughout a cluster to enable reliable and rapid computations.

- **HBase:** HBase is a distributed, column-oriented database. HBase uses HDFS for its underlying storage. It supports batch style computations using MapReduce and point queries (random reads). HBase is used in Hadoop when random, realtime read/write access is needed. Its goal is the hosting of very large tables running on top of clusters of commodity hardware.

- **Pig:** Pig is a platform for analyzing large data sets. It consists of a high level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The main characteristic of Pig programs is that their structure can be substantially parallelized enabling them to handle very large data sets. At the present time, Pig's infrastructure layer consists of a compiler that produces sequences of MapReduce programs and the textual language called Pig Latin.

- **ZooKeeper:** ZooKeeper is a high performance coordination service for distributed applications. ZooKeeper centralizes the services for maintaining the configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable.

- **Hive:** Hive is a data warehouse infrastructure built on top of Hadoop. Hive provides tools to enable easy data summarization, ad-hoc querying and analysis of large datasets stored in Hadoop files. It provides a mechanism to put structure on

this data and it also provides a simple query language called Hive QL, based on SQL, enabling users familiar with SQL to query this data. Hive QL also allows traditional MapReduce programmers to be able to plug in their custom mappers and reducers to do more sophisticated analysis which may not be supported by the built in capabilities of the language.

- **Chukwa:** Chukwa is a data collection system for monitoring large distributed systems. Chukwa includes a flexible and powerful toolkit for displaying, monitoring and analyzing results to make the best use of the collected data.

Hadoop can in theory be used for any sort of work that is batch oriented, very data intensive, and able to work on pieces of the data in parallel. Commercial applications of Hadoop include:

- Log and/or clickstream analysis of various kinds.
- Marketing analysis.
- Machine learning and/or sophisticated data mining[2].
- Image processing.
- Processing of XML messages.
- Web crawling and/or text processing.
- General archiving, including of relational/tabular data, e.g. for compliance.

2.1.1 Commercial use and Contributors

Some successful corporations are using Hadoop as backbone of their business. This is the case of Yahoo, one of the bigger supporters of the Project. On February 19, 2008, Yahoo launched what they claimed was the world's largest Hadoop production application. The Yahoo Search Webmap is a Hadoop application that runs on more than 10,000 core Linux cluster and produces data that is now used in every Yahoo Web search query. There are multiple Hadoop clusters at Yahoo, each occupying a single datacenter (or fraction of it). The work that the clusters perform is known to include the index calculations for the Yahoo search engine. On June 10, 2009, Yahoo! released its own distribution of Hadoop[3].

[2]The apache Mahout project, `http://mahout.apache.org/` implements several machine learning algorithms in top of Hadoop.

[3]`http://developer.yahoo.com/hadoop/`

In 2007, IBM and Google announced an initiative that uses Hadoop to support university courses in distributed computer programming[4]. In 2008 this collaboration, the Academic Cloud Computing Initiative (ACCI), partnered with the National Science Foundation to provide grant funding to academic researchers interested in exploring large data applications. This resulted in the creation of the Cluster Exploratory (CLuE) program.

Also Amazon, by allowing to run Hadoop on its Amazon Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (S3), has been a key player on increasing the popularity of Hadoop in the Commercial area. As an example The New York Times used 100 Amazon EC2 instances and a Hadoop application to process 4TB of raw image TIFF data (stored in S3) into 11 million finished PDFs in the space of 24 hours at a computation cost of about $240 (not including bandwidth)[5].

The official Apache Hadoop distribution supports, out of the box, the Amazon S3 filesystem. Additionally, the Hadoop team generates EC2 machine images after every release. From a pure performance perspective, Hadoop on S3/EC2 is inefficient, as the S3 filesystem is remote and delays returning from every write operation until the data is guaranteed to not be lost. This removes the locality advantages of Hadoop, which schedules work near data to save on network load. On April 2, 2009 Amazon announced the beta release of a new service called Amazon Elastic MapReduce which they describe as "a web service that enables businesses, researchers, data analysts, and developers to easily and cost-effectively process vast amounts of data. It utilizes a hosted Hadoop framework running on the web scale infrastructure of Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3)."

Other prominent users of Apache Hadoop include: AOL, Facebook, Freebase, Fox Interactive Media, ImageShack, Last.fm, Linkedln, Meebo, Ning and Twitter.

The following sections contain a brief description of HDFS and HBase to familiarize the reader with the Hadoop sub-projects that were directly used during this thesis.

[4] `http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html`
[5] `http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/`

2.1.2 HDFS

When a dataset grows beyond the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called distributed filesystems. Since they are network based there are many complications inherited from the network infrastructure. Problems like latency and node failure must be correctly handled without any data loss. HDFS is a filesystem designed for storing very large files with streaming data access patterns running on clusters of commodity hardware.

Filesystem	Description
Local	A filesystem for locally connected disks.
HDFS	Hadoop's distributed filesystem.
HFTP	Provides read only access to HDFS over HTTP.
HSFTP	Provides read only access to HDFS over HTTPS.
KFS (Cloud Store)	Distributed filesystem similary to HDFS (formerly known as Kosmos filesystem)
FTP	A filesystem backed by an FTP Server.
S3 (native)	A filesystem backed by Amazons S3. Has file size limit of 5GB.
S3 (block based)	A filesystem backed by Amazons S3, which store files in blocks. (Similar to HDFS)

Table 2.1: Hadoop Filesystems

Hadoop has an abstract notion of filesystem, of which HDFS is just an implementation. Table 2.1 presents the implementations available on the official Hadoop distribution.

2.1.3 HDFS Core Concepts

An HDFS cluster has two types of node operating in a master-worker pattern: a *namenode* (the master) and a number of *datanodes* (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk using two types of files: the namespace image and the edit log. It is also the namenode responsibility to know the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

A client accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. The client presents a POSIX[6]-like filesystem interface, so the user code does not need to know about the details of the namenode and datanodes to work correctly.

Datanodes perform the dirty work on the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode) and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode goes down, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure.

Figure 2.1 helps to get an idea of how the data flows between a client interacting with HDFS.

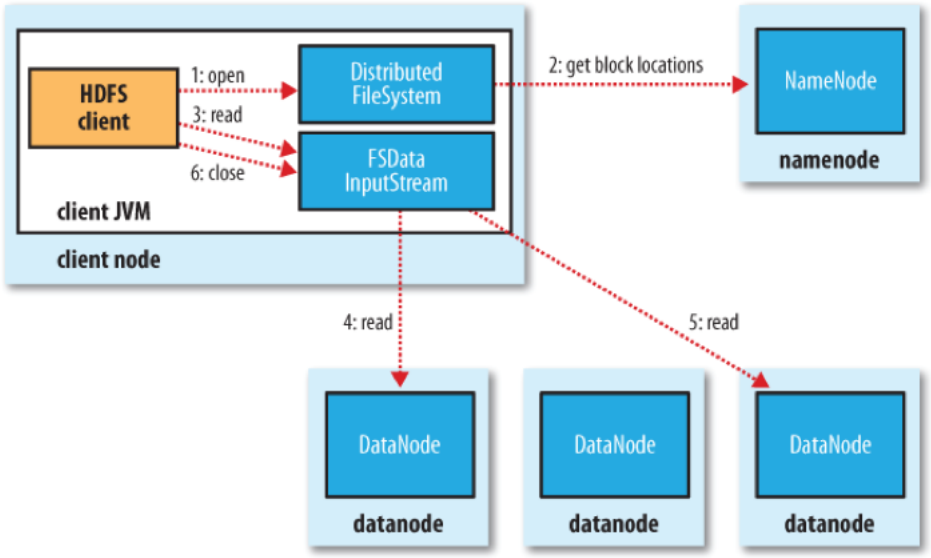


Figure 2.1: HDFS Data flow

[6]Portable Operating System Interface [for Unix]

Blocks

A disk has a block size which is the minimum amount of data that it can read or write. Filesystems for a single disk use this concept by dealing with data in blocks which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes. This is generally transparent for the filesystem user who is simply reading or writing a file. HDFS has also the concept of blocks, but it is a much larger size: 64 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage. Having larger blocks compared to normal disc blocks minimize the cost of seeks in HDFS. By making a block large enough, the time to transfer the data from the disk can be made to be significantly larger than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

Having a block abstraction for a distributed filesystem brings several benefits. First, a file can be larger than any single disk in the network. There is nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. Second, making the unit of abstraction a block rather than a file simplifies the storage subsystem. The storage subsystem deals with blocks, simplifying storage management and eliminating metadata concerns. Third, blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client. When a block that is no longer available, due to corruption or machine failure, can be replicated from their alternative locations to other live machines to bring the replication factor back to the normal level.

2.2 MapReduce

MapReduce is a programming model originally designed and implemented at Google [4] as a method of solving problems that require large volumes of data using large clusters of inexpensive machines. The programming model is based on two distinct phases. A *Map phase* executes a function of the form $Map : (k_1, v_1) \rightarrow list(k_2, v_2)$ that obtains the input and performs an initial transformation over the data. The *Reduce*

phase aggregates the output of the mappers allowing all associated records to be processed together in the same node. The function of the Reduce phase follows the form $Reduce : (k_2, list(v_2)) \rightarrow list(v_3)$. The transformation functions of the map and reduce phase can be executed in parallel, keeping in mind that the execution in each node is isolated from executions in other nodes. Figure 2.2 introduces the model of a MapReduce execution.

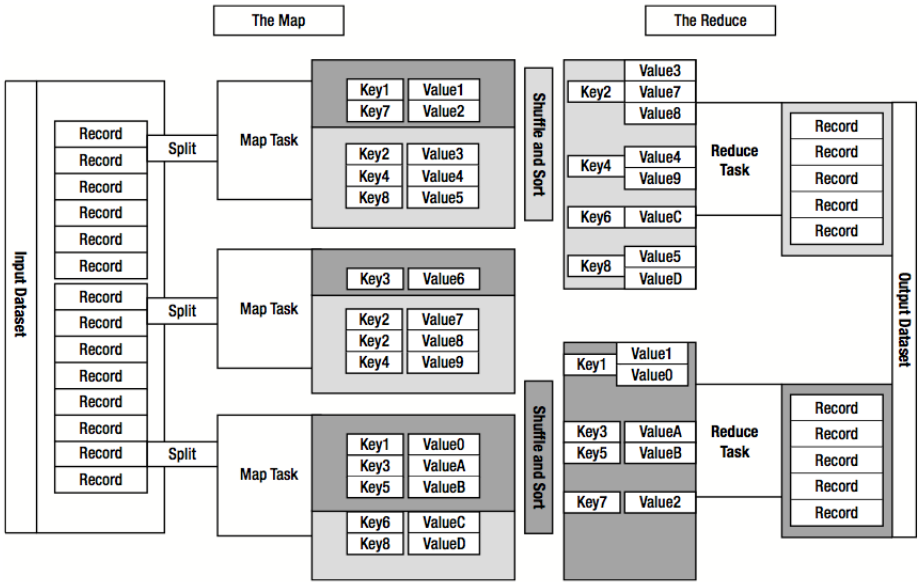


Figure 2.2: Map Reduce Flow

When a MapReduce job is submitted to Hadoop, the framework decomposes the job into a set of map tasks, shuffles, a sort and a set of reduce tasks. Hadoop is responsible for managing the distribution execution of the tasks, collect the output and report the status to the user. A typical MapReduce job in hadoop consist of the parts show in Figure 2.3.

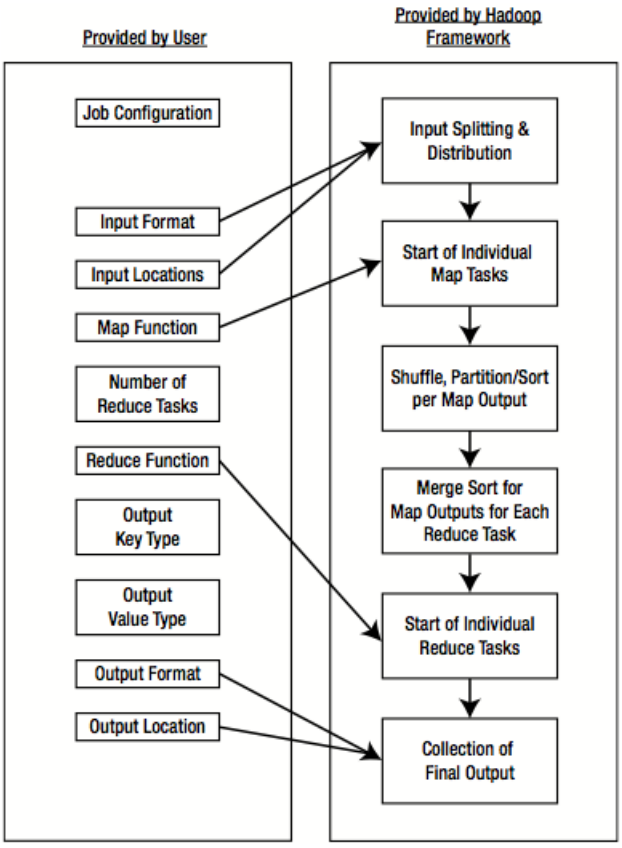


Figure 2.3: Parts of a MapReduce job

The advantage of the programming model provided by MapReduce is that it allows the distributed execution of the map and reduce operations. When each mapping operation is independent of the other, all maps can be performed in parallel, limited only by the data source and the number of CPUs available. Similarly, the reducers can perform the reduction phase in parallel. This kind of parallel execution also allows the execution to recover from a partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled.

To achieve reliability during the execution of a MapReduce job each node is expected to report back periodically with completed work and status updates. If a node doesn't report back for a period longer than a configurable interval, the master node marks the node as dead and distributes the node's work to other available nodes. Reduce operations operate much the same way. Because of their inferior properties with regard to parallel operations, the master node attempts to schedule reduce operations on the same node or in the same rack as the node holding the data being operated. This property is desirable

as it reduces the bandwidth consumption across the backbone network of the datacenter.

2.2.1 A MapReduce Example

In essence MapReduce is just a way to take a big task and split it into discrete task that can be done in parallel. A simple problem that is often used to explain how MapReduce works in practice consists in counting the occurrences of single words within a text. This kind of problem can be easily solved by launching a single MapReduce job as follows.

Initially, the input text is converted into a sequence of tuples that have as key and value all the words of the text. For example, the sentence "Hello world from MapReduce." is converted into 4 tuples, each of them containing one word of the sentence both as key and as value. The mapper algorithm is executed for every tuple in the input of the form <word,word>, the algorithm returns an intermediate tuple of the form <word, 1>. The key of this tuple is the word itself while 1 is an irrelevant value. After the map function has processed all the input, the intermediate tuples will be grouped together according to their key.

```
map(key, value):
//key: word
//value: word
        output.collect(key ,1)

reduce(key, iterator values):
        count = 0
        for (value in values)
                count = count + 1
        output.collect(key,count)
```

Figure 2.4: Word Counter

The reduce function counts the number of tuples in the group. Because the number of tuples reflects the number of times the mappers have encountered that word in the text, the reducers output the result in a tuple of the form <word,count>. The tuples encode the number of times that a word appears in the input text. An overall picture of this particular job execution is given in Figure 2.5.

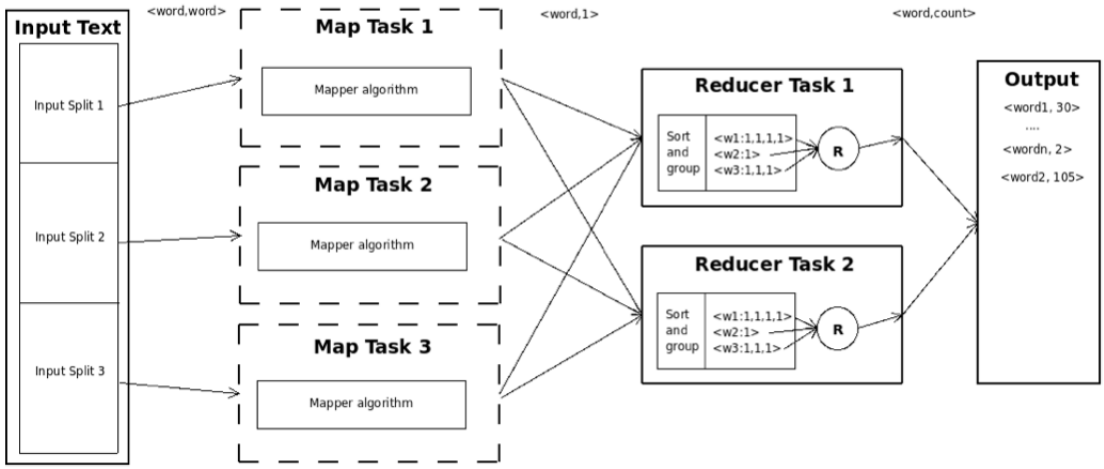


Figure 2.5: Map Reduce Example

2.3 HBase

HBase, modeled after Google's Bigtable[5], is a distributed column-oriented database written in Java and built on top of HDFS. HBase is used on Hadoop when real time read/write random access is required in very large datasets. Although there are many products for database storage and retrieval in the market, most solutions (specially relational databases) are not built to handle very large datasets and distributed environments. Many vendors offer replication and partitioning solutions to grow the database beyond the limits of a single node but these add-ons are generally an afterthought and are complicated to install and maintain. These add-ons also add several constraints to the feature set of RDBMs. Joins, complex queries, triggers, constraints become more expensive to run or do not work at all. To avoid those problems HBase was built from the ground to scale linearly by just adding nodes to an existing cluster.

Some of the HBase features include compression, in-memory operation and filters per column basis as outlined in the original BigTable paper[5]. Tables in HBase can be used as the input and output of MapReduce jobs that run in Hadoop or could be accessed through the Java API, REST or Thrift[7] gateway APIs.

[7]Thrift is a software framework from the Apache foundation for cross-language services development.

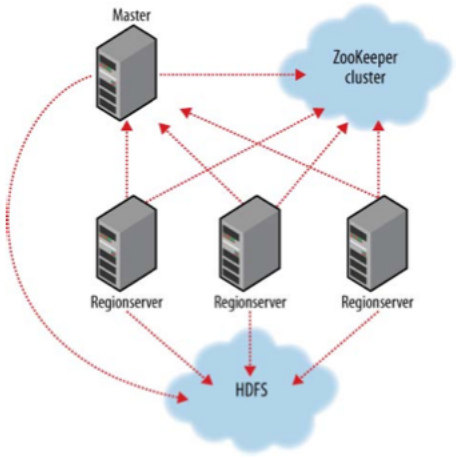


Figure 2.6: HBase Cluster

The scaling capabilities of HBase comes at a cost: HBase is not a relational database and doesn't support SQL, however, given the proper problem, HBase is able to do what a RDBMS cannot: host vast amounts of data on sparsely populated tables on clusters made from commodity hardware.

2.3.1 HBase Concepts

In HBase, the data is store into labeled tables made of rows and columns. Each table cell, an intersection of a row and a column, is versioned. By default, the version field is a timestamp auto assigned by HBase when the cell is inserted. The content of all the cells is an uninterpreted array of bytes. The row key of a Table, its primary key, is also a byte array, this means that any kind of object can serve as a row key, from strings to serialized data structures. Typically all the access to the table is done via the table primary key, although there are third party facilities in HBase to support secondary indexes.

The row columns in an HBase table are grouped together into *column families*. All columns family members have a common prefix, for example, the columns *person:name* and *person:lastname* are both members of the *person* column family. The column families are specified up front with the table schema definition, but new columns can be added to the column family on demand. Physically, the column family members are stored together on the filesystem.

Tables are automatically partitioned horizontally by HBase into *Regions*. Each region compromises a subset of table's rows. A region is defined by its first row, inclusive, and the last row, exclusive, plus a randomly generated region identifier. Initially a table

contains a single region but as the size of the region grows and after it crosses a configurable size threshold, it splits at a row boundary into two new regions of approximately equal size. Until the first split happens, all loading will be against the single server hosting the original region. As the table grows, the number of its regions grows. Regions are the units that get distributed over an HBase cluster. In this way, a table that is too big can be carried by a cluster of servers where each node hosts a subset of the table. Row Updates in HBase are atomic, no matter how many row columns constitute the row level transaction. This keeps the locking model simple for the user.

Chapter 3

Semantic Web and DL Concepts

This chapter focuses on presenting the main concepts related to the Semantic Web and Description Logics.

3.1 Semantic Web

The Semantic Web is an effort to bring back structure to the information available on the Web by describing and linking data to establish context or semantics that adhere to defined grammar and language constructs. The structures are semantic annotations that conform a specification of the intended meaning. Therefore, the Semantic Web contains implicit knowledge often incomplete since it assumes open world semantics.

The Semantic Web addresses semantics through standardized connections to related information. This includes labeling data, unique and addressable, allowing that data to be connected to a larger context, or the web. The web offers potential pathways to its definition, relationships to a conceptual hierarchy, relationships to associated information and relationships to specific instances. The flexibility of a web form enables connections to all the necessary information, including logic rules. The pathways and terms form a domain vocabulary or ontology.

The flexibility and many types of Semantic Web statements allow the definition and organization of information to form rich expressions, simplify integration and sharing, enable inference of information and allow meaningful information extractions while the information remains distributed, dynamic and diverse.

A set of statements that contribute to the Semantic Web exists primarily in two

forms; knowledge bases and files. Knowledge bases offer dynamic, extensible storage similar to relational databases. Files typically contain static statements. Table 3.1 compares relational databases and knowledge bases.

Feature	Relational Database	Knowledgebase
Structure	Schema	Ontology statements
Data	Rows	Instance statements
Administration Language	DDL	Ontology statements
Query Language	SQL	SPARQL
Relationships	Foreign Keys	Multidimensional
Logic	External to database/triggers	Formal logic statements
Uniqueness	Key for table	URI

Table 3.1: Comparison of Relational Databases and Knowledge Bases

Relational databases depend on a schema for structure. A knowledge base depends on ontology statements to establish structure. Relational databases are limited to one kind of relationship, the foreign key. Instead, the Semantic Web offers multidimensional relationships such as inheritance, part of, associated with, and many other types, including logical relationships and constraints. An important note is that the language used to form structure and the instances themselves is the same language in knowledge bases but quite different in relational databases. Relational databases offer a different language, Data Description Language (DDL), to establish the creation of the schema. In relational databases, adding a table or column is very different from adding a row. Knowledge bases really have no parallel because the regular statements define the structure or schema of the knowledge base as well as individuals or instances.

One last area to consider is the Semantic Web's relationship with other technologies and approaches. The Semantic Web complements rather than replaces other information applications. It extends the existing WWW rather than competes with it, offering powerful semantics that can enrich existing data sources, such as relational databases, web pages, and web services or create new semantic data sources. All types of applications can benefit from the Semantic Web, including standalone desktop applications, mission critical enterprise applications, and large scale web applications/services. The Semantic Web causes an evolution in the current Web to offer richer, more meaningful interactions with information.

The Semantic Web offers several languages. Rather than have one language to resolve all the information and programming needs, the Semantic Web offers a range from basic

to complex. This provides Semantic Web applications with choices to balance their needs for performance, integration and expressiveness. As shown on Figure 3.1 there are some standards on top of XML that allow to express knowledge in the Semantic Web. On the Semantic Web, information is modeled primary with a set of three complementary languages: The Resource Description Framework (RDF), RDF Schema and the Web Ontology Language, OWL. RDF Defines the underlying data model and provides a foundation for the more sophisticated features of the higher levels of the Semantic Web.

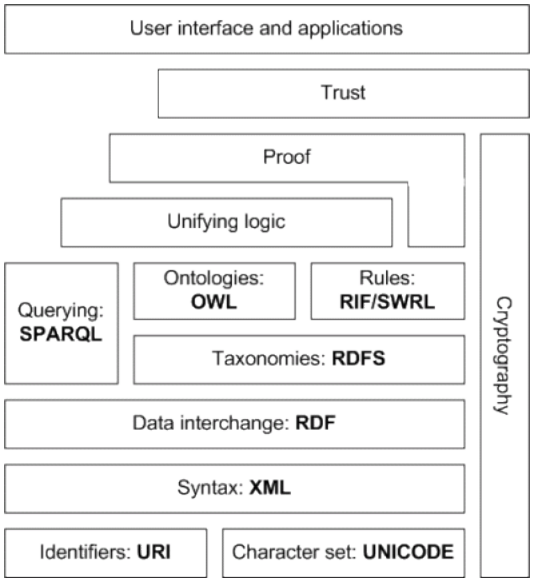


Figure 3.1: The Semantic Web Stack

3.1.1 The Resource Description Framework (RDF)

RDF consist of a family of specifications from the W3C[1] originally designed as a metadata model used now as a general method for conceptual description or modeling of information that is implemented in web resources.

In the Semantic Web information is represented as a set of assertions called *statements* that are made up of three parts: Subject, Predicate and Object. Because of these three elements, statements are typically referred as *Triples*. The *subject* of a statement is the thing that the statements describes. The *predicate* describes a relationship between the subject and the *object*. Every triple can be seen as a small sentence. An example could be the triple "John plays Guitar" where *John* is the subject, *eats* is the predicate and *Guitar* the object. This kind of RDF assertions form a directed graph, with subjects

[1]World Wide Web Consortium. `http://www.w3.org/`

and objects of each statements as nodes and predicates as edges.

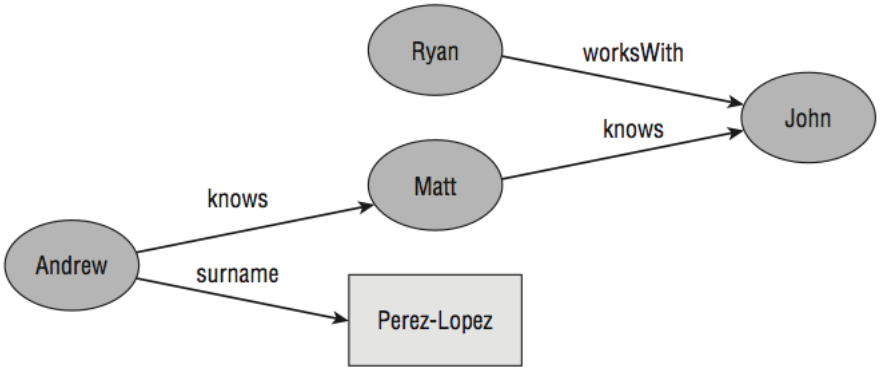


Figure 3.2: A graph representation of a RDF statement

The nodes in a RDF graph are the subjects and objects of the statements that make up the graph. There are two kinds of nodes: *Literals* and *Resources*. Literals represent the concrete data values like numbers or strings and can't be the subject of a statement, only the objects. Resources, in contrast, represent everything else; they can be either subjects or objects. Resources can represent anything that can be named. Resource names take the form of URIs (Universal Resource Identifiers). Predicates, also called *properties*, represent the connections between resources; predicates are themselves resources. Like subjects, predicates are represented by URIs.

An RDF graph can be Serialized using multiple formats making the information represented in them easy to exchange by providing a way to convert between the abstract model and a concrete format. There are several, equally expressive serialization formats. Some of the most popular are RDF/XML[2], N3 [3], N-Triples[4] and Turtle[5]. This means that depending of the application needs an RDF statement can be serialized in different ways.

[2]http://www.w3.org/TR/rdf-syntax-grammar/
[3]http://www.w3.org/DesignIssues/Notation3
[4]http://www.w3.org/TR/rdf-testcases/#ntriples
[5]http://www.w3c.org/TeamSubmission/turtle

3.1.2 RDFS

The Resource Description Framework (RDF) provides a way to model information but does not provide a way to specify what that information means. In other words RDF cannot express semantics. To add meaning to RDF a vocabulary of predefined terms is needed to describe the information semantics. RDF Schema (RDFS) extends RDF providing a language with which the users can develop shared vocabularies, that have a well understood meaning and it is used in a consistent manner to describe other resources. RDF Schema provides basic elements for the description of ontologies, otherwise called Resource Description Framework (RDF) vocabularies, intended to structure RDF resources.

RDFS vocabularies describe the classes of resources and properties being used in a RDF model allowing to arrange classes and properties in generalization/specialization hierarchies, define domain and range expectations for properties, assert class memberships and specify and interpret datatypes. RDFS is one of the fundamental building blocks of ontologies in the Semantic Web and is the first step to incorporate semantics into RDF.

The list of classes defined by RDFS is shown in Table 3.2. Classes are also resources, so they are identified by URIs and can be described using properties. The members of a class are instances of classes, which is stated using the *rdf:type* property.

rdfs:Resource	rdfs:subClassOf	rdf:type
rdfs:Resource	rdfs:Resource	rdfs:Class
rdfs:Class	rdfs:Resource	rdfs:Class
rdfs:Literal	rdfs:Resource	rdfs:Class
rdfs:Datatype	rdfs:Class	rdfs:Class
rdf:XMLLiteral	rdfs:Literal	rdfs:Datatype
rdf:Property	rdfs:Resource	rdfs:Class
rdf:Statement	rdfs:Resource	rdfs:Class
rdf:List	rdfs:Resource	rdfs:Class
rdfs:Container	rdfs:Resource	rdfs:Class
rdf:Bag	rdfs:Container	rdfs:Class
rdf:Seq	rdfs:Container	rdfs:Class
rdf:Alt	rdfs:Container	rdfs:Class
rdfs:ContainerMembershipProperty	rdf:Property	rdfs:Class

Table 3.2: RDFS Classes

In RDFS a class may be an instance of a class. All resources are instances of the class *rdfs:Resource*. All classes are instances of *rdfs:Class* and subclasses of *rdfs:Resource*. All properties are instances of *rdf:Property*. Properties in RDFS are relations between subjects and objects in RDF triples, i.e., predicates. All properties may have defined domain and range. The Domain of a property states that any resource that has given property is an instance of the class. Range of a property states that the values of a property are instances of the class. If multiple classes are defined as the domain and range then the intersection of these classes is used. An example stating that the domain of hasSon property is Person and that the domain of the same property is Man follows:

```
@prefix  : <http://www.example.org/sample.rdfs#>.
@prefix  rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
         :hasSon  rdfs:domain  :Person;
                  rdfs:range   :Man.
```

3.2 Web Ontology Language (OWL)

With RDF Schema it is possible to define only relations between the hierarchy of the classes and the properties or define the domain and range of these properties. The community needed a language that could be used for more complex ontologies and therefore work was done to create a richer language that would be later released as the OWL language. The Web Ontology Language (OWL) extends the RDFS vocabulary with additional resources that can be used to build more expressive ontologies for the web. OWL adds restrictions regarding the structure and contents of RDF documents in order to make processing and reasoning more computationally decidable. OWL uses the RDF and RDFS, XML Schema datatypes, and OWL namespaces. The OWL vocabulary itself is defined in the namespace *http://www.w3.org/2002/07/owl* and it is commonly referred to by the prefix owl. OWL 2 extends the original OWL vocabulary and reuses the same namespace. The full set of namespaces used in OWL and their associated prefixes are listed in Table 3.3.

Namespace	Prefix
http://www.w3.org/1999/02/22-rdf-syntax-ns	rdf
http://www.w3.org/2000/01/rdf-schema	rdfs
http://www.w3.org/2001/XMLSchema	xsd
http://www.w3.org/2002/07/owl	owl

Table 3.3: OWL Namespaces

Resources on the web are distributed, as a result, the resources descriptions contained in the Semantic Web are also distributed. OWL supports this kind of distributed knowledge base because is built on RDF, which allows you to declare and describe resources locally or refer to them remotely. OWL also provides a mechanism to import and reuse ontologies in a distributed environment.

3.2.1 OWL Languages and Profiles

Both the original OWL specification and OWL 2 provide profiles, or sublanguages of the language, that give up some expressiveness in exchange for computational efficiency. These profiles introduce a combination of modified or restricted syntax and nonstructural restrictions on the use of OWL. In the original OWL specification, there were three sublanguages:

- **OWL Lite:** The intention OWL Lite was to provide applications and tool developers with a development target or starting point for supporting primarily classification hierarchy and simple constraints. Unfortunately, OWL Lite was regarded mostly as a failure because it eliminated too many of the useful features without introducing enough of a computational benefit to make the reduced features attractive.

- **OWL Full:** OWL Full It is a pure extension of RDF. As a result, every RDF document is a valid OWL Full document, and every OWL Full document is a valid RDF document. The important point to make here is that OWL Full maintains the ability to say anything about anything. With the flexibility comes a tradeoff in computational efficiency making OWL Full not decidable[6].

- **OWL DL**: OWL DL provides many of the capabilities of description logics. It contains the entire vocabulary of OWL Full but introduces the restriction that the semantics of OWL DL cannot be applied to an RDF document that treats a URI as both an individual and a class or property. This and some additional restrictions make OWL DL decidable.

With OWL 2 the main purpose of an OWL profile, as with the original OWL Languages, is to produce subsets of OWL that trade some expressivity for better computational

[6]Decidability specifies that there exists an algorithm that provides complete reasoning. It does not say anything about the performance of such an algorithm or whether it will complete in an acceptable or realistic amount of time

characteristics for tools and reasoners. The profiles were developed with specific user communities and implementation technologies in mind. The three standardized profiles are:

- **OWL EL:** The OWL EL profile is designed to provide polynomial time computation for determining the consistency of an ontology and mapping individuals to classes. The relationship between ontology size and the time required to perform the operation can be represented by the formula $f(x) = x^a$. The purpose of this profile is to provide the expressive features of OWL that many existing large scale ontologies (from various industries) require while also eliminating unnecessary features. OWL EL is a syntactic restriction on OWL DL.

- **OWL QL:** The OWL QL profile is designed to enable the satisfiability of conjunctive queries in log-space with respect to the number of assertions in the knowledge base that is being queried. The relationship between knowledge base size and the time required to perform the operation can be represented by the function $f(a) = log(a)$**.** As with OWL EL, this profile provides polynomial time computation for determining the consistency of an ontology and mapping individuals to classes.

- **OWL RL:** The OWL RL profile is designed to be as expressive as possible while allowing implementation using rules and a rule processing system. Part of the design of OWL RL is that it only requires the rule processing system to support conjunctive rules. The restrictions of the profile eliminate the need for a reasoner to infer the existence of individuals that are not already known in the system, keeping reasoning deterministic.

3.3 Reasoning Basics

3.3.1 Open World Assumption

The open world assumption states that the truth of a statement is independent of whether it is known. In other words, not knowing whether a statement is explicitly true doesn't imply that the statement is false. The closed world assumption states that any statement that is not known to be true can be assumed to be false. Under the open world assumption, new information must always be additive. It can be contradictory, but it cannot remove previously asserted information. This assumption has a significant

impact on how information is modeled an interpreted.

Most systems operate with a closed world assumption. They assume that the information is complete and known. For many applications this is a safe and necessary assumption to make. However, a closed world assumption can limit the expressivity of a system in some cases because it is more difficult to differentiate between incomplete information and information that is known to be false.

The open world assumption impacts the kind of inference that can be performed over the information contained in the model. In DL it means that reasoning can be performed only over information that is known. The absence of a piece of information cannot be used to infer the presence of other information.

3.3.2 No Unique Name Assumption

The distributed nature of the Semantic Web makes unrealistic to assume that everyone is using the same URI to identify a specific resource. Often a resource is described by multiple users in multiple locations and each of those users is using his or her own URI to represent the resource.

The no unique name assumption states that unless explicitly stated, it can't be assumed that resources identified by different URIs are different. This assumption differs for the one used on traditional systems. In most database systems, all information is known and assigning and unique identifier, such a primary key, is possible. Like the open world assumption, the no unique names assumption impact inferences capabilities related to the uniqueness of resources. Redundant and ambiguous data is a common issue in information management systems, the unique names assumption makes these issues easier to handle because resources can be made the same without destroying any information or dropping and updating database records.

3.4 Description Logics

One of the reasons behind the fact that the Semantic Web community adopted Description Logics (DL) as a core technology is that DL has been purpose of multiple studies that try to understand how constructors interact and combine to affect reasoning.

The Language expressions using on a DL describe classes (concepts) of individuals

that share some properties. Properties can also be specified by means of relations (roles) between individuals. The language is compositional, i.e. the concept descriptions are built by combining different subexpressions using constructors. The semantics of the language is given in a set theoretical way over a domain which is a set of elements. A concept expression corresponds to a subset of the domain, while a role expression corresponds to a binary relation over the domain. As their shape suggests, some of these constructors have a strong relationship with boolean operators and logical quantifiers.

A DL knowledge base consists of a terminology, TBox, and an assertional part, ABox. The TBox contains the definitions of the terms (concept definitions), while the ABox contains a set of membership and role assertions. Membership assertions relate an individual to a concept, stating that the individual involved is an instance of the concept. Role assertions state that two individuals are linked by a given role. For example:

Terminology	Humans are Animals. Women are Humans and not Male	TBox	$Human \sqsubseteq Animal$ $Women \sqsubseteq Human \sqcap \neg Male$
Assertions	gabi is-a Human and not Male and has a Friend who is Male	ABox	$Human \sqcap \neg Male \sqcap$ $\exists hasFriend.\, Male\,(gabi)$

Figure 3.3: Knowledge base example

Reasoning with DL knowledge bases is a deduction process which extracts not only the facts explicitly asserted in a knowledge base but also their logical consequences. When only the terminology is involved in a deduction the reasoning is said to be *Terminological*, otherwise it is said to be *Hybrid*.

Description Logic languages can be categorized in many different logics, distinguished by the constructors they provides. The work done is this project is focused on ALC, considered a basis of many DL systems.

3.4.1 ALC Family of Description Logics

The language ALC consist of an alphabet of distinct concept names, role names and individual names, together with a set of constructors for building concept and role expressions. Formally, a description logic knowledge base is a pair $K =< T, A >$ where T

is a TBox and A is an ABox. The TBox contains a finite set of axiom assertions of the form $C \sqsubseteq D \mid C \doteq D$, where C and D are concept expressions. Concept expressions are of the form: $A \mid \top \mid \neg C \mid C \sqcap U \mid C \sqcup D \mid \exists R.C \mid \forall R.C$, where A is an atomic concept, R is a role name, $\top$ (top or full domain) is the most general concept and $\perp$ (bottom or empty set) is the least general concept. The ABox contains a finite set of assertions about individuals of the form $C(a)$ (Concept membership assertions) and $R(a, b)$ (Role Membership Assertions), where *a, b* are individual names.

The Semantics of description logic are defined in terms of an interpretation $I = (\triangle^I, .^I)$, consisting of a nonempty domain $\triangle^I$ and an interpretation function $.^I$. The interpretation function maps concept names into subsets of the domain ($A^I \subseteq \triangle^I$), role names into subsets of the Cartesian product of the domain ($R \subseteq \triangle^I * \triangle^I$), and individual names into elements of the domain. The only restriction on the interpretations is the unique name assumptions (UNA). Given a concept name A (or a role name R), the set denoted by A^I(or R^I) is called the interpretation or extension of A (or R) with respect to *I.*

The interpretation is extended to cover concepts built from negation ($\neg$) , conjunction($\sqcap$), disjunction($\sqcup$), existential quantification($\exists R.C$) and universal quantification($\forall R.C$) as follows:

$(\neg C)^I = \triangle^I \diagdown C^I$

$(C \sqcap D)^I = C^I \cap D^I$

$(C \sqcup D)^I = C^I \cup D^I$

$(\exists R.C)^I = \{x \ \in \ \triangle^I | \exists y. < x, y > \in \ R^I \bigwedge y \ \in \ C^I\}$

$(\forall R.C)^I = \{x \ \in \ \triangle^I | \forall y. < x, y > \in \ R^I \rightarrow y \ \in \ C^I\}$

An interpretation I satisfies (entails) an inclusion axiom $C \subseteq D$ (Written $I \models C \subseteq D$) if, and it satisfies an equality $C \doteq D$ if $C^I = D^I$. It satisfies a TBox T if it satisfies each assertion in T. The interpretation I satisfies a concept membership assertion *C(a)* if $a^I \ \in \ C^I$ and satisfies a role membership assertion *R(a,b)* if $(a^I, b^I) \ \in \ R^I$. I satisfies an ABox A ($I \vDash A$) if it satisfies each assertion in A. if I satisfies an axiom (or a set of axioms), then it is a model of the axiom (or set of axioms). Two axioms are equivalent

if they have the same models. Given a knowledge base $K =< T, A >$, the knowledge base entails an assertion α(written $(K \vDash \alpha)$) iff for every interpretation I, if $I \models A$ and $I \models T$ then $I \vDash \alpha$.

The DL ABox can be viewed as a semi-structured database, while the TBox contains a set of constraints for the data in the ABox. The TBox can be compared to data model in databases (Entity-Relationship Model in databases) but the semantic of description logics are defined in terms of interpretation differentiating them with databases. In addition, the domain of interpretation can be chosen arbitrary and it can be infinite. This, together with the open world assumption, are features that differentiate description logics from traditional databases. Another particular feature of description logic is the reasoning capabilities that are associated with it. Reasoning allow to exploit the description of the model to draw conclusions about a problem.

The research community has extensively study the decidability and complexity of ALC and its sublanguages, designing sound and complete subsumption testing algorithms, some extensions of ALC include:

- $ALC, ALCR, ALCNR$: Add number restriction concept expressions (N) and/or role conjunction (R) to ALC.
- $ALCF$: Add attributes (also called features), attribute composition and attribute value map concept expressions to ALC.
- $ALCFN, ALCFNR$: Add number restriction expressions and role conjunction to $ALCF$.
- $ALCN(o)$: Adds role composition in number restriction expressions to ALC.
- ALC_+: Adds union, composition and transitive closure role expressions to ALC.
- ALC_{R+}: Adds transitively closed primitive roles to ALC (axioms of the form $RN \in R_+$).
- $ALC_{\bigoplus}$: Adds a restricted form of predictive role introductions axioms to ALC_{R+}.
- TSL: Adds union, composition, identity, transitive reflexive closure and inverse role expressions to ALC.
- CIQ: Adds qualified number restriction expressions (inverse roles are the only form of role expression allowed in qualified number restrictions) to TSL.

It is important to notice that extending the syntax of the DL language does not necessary increase its expressiveness. Figure 3.4 show some of the members of the ALC family.

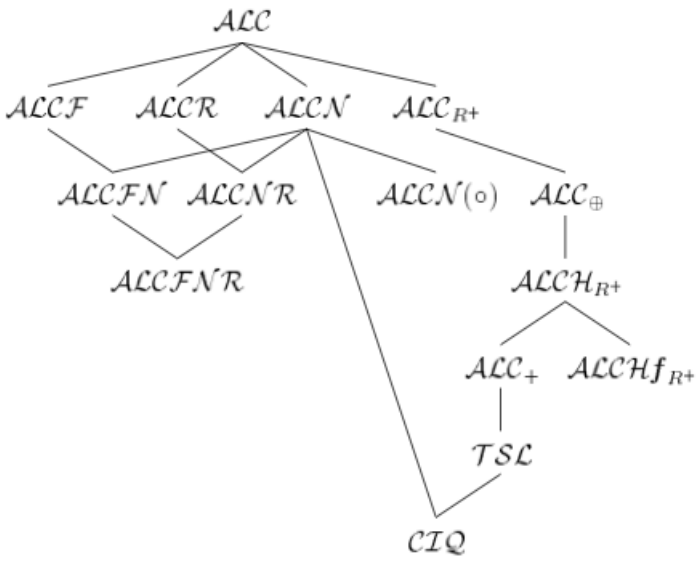


Figure 3.4: ALC family hierarchy

3.5 Reasoning in Description Logics

A DL knowledge base support two kinds of reasoning tasks: TBox reasoning and ABox Reasoning. In TBox reasoning the basic reasoning services consist of:

- **Knowledge Base Consistency:** A TBox T is consistent iff it is satisfiable, i.e. there is at least a non empty model for T. An interpretation I is a model for T if it satisfies every assertion in T.
- **Satisfiability:** *A* concept C is satisfiable with respect to T if there exist a model I of T such that C^I is nonempty. I is also a model of C.
- **Subsumption:** *A* concept D subsumes a concept C with respect to T *(*$C \subseteq_T D$ *or* $T \models C \doteq D$*)* if $C^I = D^I$ for every model I of T.
- **Equivalence:** Two concepts C and D are equivalent with respect to T ($C \doteq_T D$ *or* $T \models C \doteq D$) if $C^I = D^I$ for every model I of T.
- **Disjointness:** Two concept C and D are disjoint with a respect to T if for every model I of T, $C^I \cap D^I = \theta$.

Concept satisfiability is the key inference for TBox reasoning. Subsumption, equivalence and disjointness can be reduced to concept (un)satisfiability test, which can be achieved though applying the Tableau Algorithm explained on Section 3.5.1

- **Subsumption:** C is subsumed by D ($C \subseteq D$) iff $C \sqcap \neg D$ is unsatisfiable with respect to T.
- **Equivalence:** C and D are equivalent ($C \doteq D$) iff both $C \sqcap \neg D$ and $D \sqcap \neg C$ are unsatisfiable with respect to T.
- **Disjointness:** C and D are disjoint iff $C \sqcap D$ is unsatisfiable with respect to T.

When reasoning in the ABox it has to be taken into consideration that there are only two kinds of assertions: concept membership assertion of the form *C(a)* and role membership assertion of the form *R(a,b).* Therefore, the ABox alone can't be seen as a knowledge base; the ABox must be attached with its TBox. Consequently ABox reasoning will always be done with respect to its TBox. In Description Logics the basic reasoning services for ABox are:

- **Instantiation Check:** The problem consist to determine whether an assertion is entailed by ABox A or not ($A \models C(a)$). An assertion is entailed if every interpretation that satisfies A also satisfies *C(a).*
- **Realization:** Given an individual a and a set of concepts, find the most specific concept C from the set such that $A \models C(a)$
- **Retrieval:** Given an ABox A and concept C, find all individuals a such that $A \models C(a)$
- **ABox Consistency:** An ABox A is consistent iff it is consistent with respect to the TBox T. TBox reasoning must be used for ALC expanding the ABox with unfolded TBox concepts. An unfolded concept *C'* is obtained by replacing the descriptions of the original concept with their descriptions in T. Due to the fact that C is satisfiable with respect to T iff *C'* is satisfiable; the original concept and the unfolded concept are equivalent, $C \doteq_T C'$. Thus, the expansion of A with respect to T can be obtained by replacing each concept assertion *C(a)* in A with the assertion *C'(a).* In every model of T, a concept C and its expansion *C'* are interpreted in the same way. Therefore, *A'* is consistent with respect to T iff *A'* is consistent. *A'* is consistent iff it is satisfiable (There is at least a nonempty model for *A'*)

Realization and retrieval can be reduced to an instantiation test. They can be done through a series of instantiation tests. Additionally, the instantiation test can be reduced to the consistency problem for ABox since $A \models C(a)$ iff $A \cup \{\neg C(a)\}$ is inconsistent. Concept satisfiability can also be reduced to an ABox consistency test since C is satisfiable iff *{C(a)}* is consistent, where a is an arbitrary individual.

3.5.1 Tableaux Algorithm

The main idea around the Tableaux algorithm is to prove the satisfiability of Concept expression D by finding a model $I = (\triangle^I, .^I)$ in which $D^I \neq \phi$; a tableau is a graph which represents such model with nodes corresponding to individual and edges corresponding to relationships between the individuals. Typically, the algorithm starts with a single individual that satisfies D and tries to construct a complete model by inferring the existence of additional individuals or of additional constraints on the individuals. The inference mechanism consists of applying a set of expansion rules that correspond to the logical constructs of the language; the algorithm terminates either when the model is complete (no further inferences are possible) or when an obvious contradiction appears.

To simplify the algorithm, the concept expression *D,* is assumed to be an unfolded concept expression in *negation normal form (NNF).* A concept is in NNF when negation is applied only to concept names and not to compound terms. Arbitrary concept expression can be transformed into NNF using a the DeMorgan's laws from Figure 3.5.

Initial Concept	Equivalent Concept
$\neg(A \sqcap B)$	$\neg A \sqcup \neg B$
$\neg(A \sqcup B)$	$\neg A \sqcap \neg B$
$\neg\neg A$	A
$\neg\forall R.\, A$	$\exists R.\neg A$
$\neg\exists R.\, A$	$\forall R.\, \neg A$

Figure 3.5: De Morgan's rules

3.5.2 Tableaux Algorithm for ALC

The tableaux algorithm uses a tree to represent the model being constructed. Each node x in the tree represents an individual and is labeled with a set of expression which it must satisfy: $C \in \mathcal{L}(x) \Rightarrow x \in C^I$. Each edge $< x, y >$ in the tree represents a pair of individuals in the interpretation of a role and is labeled with the role name:

$R = \mathcal{L}(< x, y > \Rightarrow < x, y > \in R^I)$.

To determine the satisfiability of a concept expression D, a tree $\mathbf{T}$ is initialized to contain a single node x_o, with $\mathcal{L}(x_0) = \{D\}$, the tree is expanded by continuously applying the rules from table 3.4. $\mathbf{T}$ is fully expanded when none of the rules can be applied. $\mathbf{T}$ contains an obvious contradiction when, for some node x and some concept C, either $\bot \in \mathcal{L}(x)$ or $\{C, \neg C\} \subseteq \mathcal{L}(x)$.

	Condtion	Actions
$\sqcap$ Rule	1. $C_1 \sqcap C_2 \in \mathcal{L}(x)$ 2. $\{C_1, C_2\} \nsubseteq \mathcal{L}(x)$	$\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$ Rule	1. $(C_1 \sqcup C_2) \in \mathcal{L}(x)$ 2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$	a. save T b. Try $(x) \rightarrow \mathcal{L}(x) \cup \{C_1\}$ if, clash restore T and try: $(x) \rightarrow \mathcal{L}(x) \cup \{C_2\}$
$\exists$ Rule	1. $\exists R.C \in \mathcal{L}(x)$ *2.* There is no y s.t $\mathcal{L}(< x, y >) = R$ and $C \in \mathcal{L}(y)$	create node y and *edge* <x,y> with: $\mathcal{L}(y) \rightarrow \mathcal{L}(y) \cup \{C\}$ and $\mathcal{L}(< x, y >) = R$
$\forall$ Rule	1. $\forall R.C \in \mathcal{L}(x)$ There is some y s.t. $\mathcal{L}(< x, y >) = R$ and $C \notin \mathcal{L}(y)$	$\mathcal{L}(y) \rightarrow \mathcal{L}(y) \cup \{C\}$

Table 3.4: ALC Tableau completion rules

A fully expanded class free tree $\mathbf{T}$ can be converted into a model to probe the satisfiability of D:

$\triangle^I = \{x \mid x \textit{ is a node in } T\}$

$C = \{x \in \triangle^I \mid C \in L(x)\}$ for all concept names C in D

$R^I = \{< x, y > \mid < x, y > \textit{ is an edge in } T \textit{ and } \mathcal{L}(< x, y >) = R\}$

The algorithm is guaranteed to terminate because:

- The $\sqcap, \sqcup, \exists$ rules can only be applied once to any given concept expression C in $\mathcal{L}(x)$.
- The $\forall$-rule can be applied many times to a given $\forall R.C$ expression in $\mathcal{L}(x)$ but only once to any given edge $< x, y >$.
- Applying a rule to concept expression D extends the labeling with a concept expression which is always smaller than D.

The $\sqcup$-rule is non-deterministic and operates by performing a depth first backtracking search of the possible expansions resulting from the disjunction in D, stoping when a fully expanded tree is found or every when every possible expansion is shown to lead to a clash.

Example: Demonstrating Subsumption using Tableaux Algorithm

Given the TBox:
$B \doteq P \sqcap \forall RA$
$D \doteq P \sqcap \forall R.(A \sqcup C)$

The tableaux algorithm can be used to show that $B \sqsubseteq D$ by demonstrating that $B \sqcap \neg D$ is not satisfiable:

1. Unfold and normalize $B \sqcap \neg D$:

   $P \sqcap \forall RA \sqcap (\neg P \sqcup \exists R.(\neg A \sqcap \neg C))$

2. Initialize **T** to contain a single node x.

   $\mathcal{L}(x) = \{P \sqcap \forall RA \sqcap (\neg P \sqcup \exists R.(\neg A \sqcap \neg C))\}$

3. Apply the $\sqcap$-rule:

   $\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{P, \forall RA, (\neg P \sqcup \exists R.(\neg A \sqcap \neg C))\}$

4. Apply the U-rule to $\neg P \sqcup \exists R.(\neg A \sqcap \neg C)$:

   (a) Save **T** and try: $\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{\neg P\}$
   $\mathcal{L}(x)$ contains a clash: $\{P, \neg P\} \subseteq \mathcal{L}(x)$

   (b) Restore T and try: $\mathcal{L}(x) \rightarrow \mathcal{L}(x) \cup \{\exists R.(\neg A \sqcap \neg C)\}$

5. Apply the $\exists$-rule to $\exists R.(\neg A \sqcap \neg C) \in \mathcal{L}(x)$:

   create a new node y and a new edge $< x, y >$

   $\mathcal{L}(y) = \{\neg A \sqcap \neg C\}$

   $\mathcal{L}(< x, y >) = R$

6. Apply the $\forall$-rule to $\forall R.A \in \mathcal{L}(x)$ and $\mathcal{L}(< x, y >) = R$:

   $\mathcal{L}(y) \rightarrow \mathcal{L}(y) \cup \{A\}$

7. Apply the $\sqcap$-rule to $\neg A \sqcap \neg C \in \mathcal{L}(y)$:

   $\mathcal{L}(y) \rightarrow \mathcal{L}(y) \cup \{\neg A, \neg C\}$

   $\mathcal{L}(y)$ contains a clash: $\{A, \neg A\} \subseteq \mathcal{L}(y)$.

Because all the possible applications of the $\sqcup$-rule (step 4) have been shown to lead to a contradiction, it can be concluded that $B \sqcap \neg D$ is unsatisfiable, with respect to *T,* thus we can say that $B \sqsubseteq D$.

3.6 Query Efficiency

There are two standard type of queries allowed in Description Logic: boolean queries and non boolean queries, they can respectively be seen as the instance checking and retrieval ABox reasoning services. A boolean query Q_b refers to a formula of the form $Q_b \leftarrow QExp$, where $QExp$ is an assertion about an individual. For example the query: $Q_b \leftarrow John : (Student \sqcap \exists takesCourse.Course)$ will return a boolean *True* or *False.* Q_b will return *True* if and only if every interpretation that satisfies the knowledge base K also satisfies *QExp* and return *False* otherwise.

A non boolean query Q_{nb} refers to a formula of the form $Q_{nb} \leftarrow QExp$ where QExp is a concept expression. For example $Q_{nb} \leftarrow Student \sqcap \exists takesCourse.Course$. In this case, the query will return one of the members of the set $\{\perp, M\}$, where $\perp$ refers to the empty set and M represent a set of models that satisfies *QExp* with respect to the knowledge base K. A non boolean query is trivially transformed into a set of boolean queries. Answering a boolean query consist in resolving an entailment problem. For example, answering $Q_b \leftarrow John : (Student \sqcap \exists takesCourse.Course)$ can be resolved by checking $K \models John : (Student \sqcap \exists takesCourse.Course)$. Thus, boolean query or instance checking can be reduce to knowledge base satisfiability: $K \models C(a)$ iff $K \cup \{\neg C(a)\}$ is unsatisfiable.

As explained before, many DL reasoners can deal with very expressive languages, however, when the reasoning has to be performed on large ABoxes, scaling problem appear. The main reasons for this problem is that the reasoners must compute the tableau algorithm, mainly using main memory and the fact that some of the Tableau algorithms require a lot of CPU power to run.

3.6.1 Retrieval Optimization using the Pseudo model technique

In order to avoid the need to run the completion of a Tableau model for every candidate individual in the ABox, the first optimization that comes to mind is to detect the obvious instances of a concept C. [10] introduces such a method. The advantage of this technique is that reduces the number satisfiability tests executed during the query answering process. The pseudo model technique introduces a very effective mergeable test which reuses information computed from previous satisfiability tests.

Each subsumption test ($C \subseteq D$) can be transformed into a satisfiability test $unsat(C \sqcap \neg D)$. If $C \sqcap \neg D$ is satisfied then $C \nsubseteq D$. In order to check if the conjunction $C \sqcap \neg D$ is satisfiable or not, a mergeable test between C's pseudo model and $\neg D$'s pseudo model can be applied. If there is not interaction between the pseudo models of C and $\neg D$, then $C \sqcap \neg D$ is satisfied. Therefore, $C \nsubseteq D$.

The pseudo model technique is sound but it is not complete. This means that if the pseudo models of C and $\neg D$ cannot be merged, the completion of the Tableau model must be complete to test the satisfiability of $C \sqcap \neg D$. However, using the mergeable test can effectively reduce the number of individual tests, reducing the number oft times the Tableau algorithm needs to run.

3.6.1.1 Flat pseudo models for ABox Reasoning

To realize an individual a, given the concepts $D_1 ... D_n$, it is required to perform a set of ABox consistency tests for $A_{D_i=} A \cup \{\neg D_i(a)\}$. The main purpose of the flat pseudo model technique is to provide an efficient sound mergeable test for an individual a and sets of concept terms $\neg D_i$.

Pseudo model for an individual *a*

Assuming that the ABox A is consistent and there exists a non-empty set of completions C. let $A' \in C$. The pseudo model M for the individual a in A is defined as the tuple $< M^A, M^{\neg A}, M^{\exists}, M^{\forall} >$ w.r.t A' and A using the following definitions:

$$M^A = \{A \mid a : A \, \epsilon \, A^{'}\}$$
$$M^{\neg A} = \{A \mid a : \neg A \, \epsilon \, A^{'}\}$$
$$M^{\exists} = \{R \mid a : \exists R.C \, \epsilon \, A^{'}\}$$
$$M^{\forall} = \{R \mid a : \forall R.C \, \epsilon \, A^{'}\}$$

Pseudo model for a concept D

Similarly the pseudo model for a concept D can be defined as follow. Given the set $L_A(a)$ defined as the set of concept terms from all concept assertions for a in a completed ABox A'. Let D be a concept and A the ABox $A = D(a)$, the pseudo model M for D consists of the sets:

$$M_D^A = \{D \mid D \,\epsilon\, L_A(a)\}$$
$$M_D^{\neg A} = \{D \mid \neg D \,\epsilon\, L_A(a)\}$$
$$M_D^A = \{R \mid \exists R.C \,\epsilon\, L_A(a)\}$$
$$M_D^A = \{R \mid \forall R.C \,\epsilon\, L_A(a)\}$$

The mergeable test for the flat pseudo models M_1 and M_2 consist in checking whether there are interactions between the models by checking for atomic concepts: $((M_D^{A_1} \cap M_{\neg D}^{A_2} \neq \phi) \vee (M_D^{A_2} \cap M_{\neg D}^{A1} \neq \phi))$ and for roles successors: $((M_{\exists R}^{A_1} \cap M_{\forall R}^{A_2} \neq \phi) \vee (M_{\exists R}^{A_2} \cap M_{\forall R}^{A1} \neq \phi))$.

The pseudo model can be used before the TBox subsumption or the ABox satisfiability test for an individual and a concept. For example, to test whether D is the type of individual a, it is sufficient to test whether a's pseudo model is mergeable with $\neg D$'s pseudo model, if they are mergeable then D is not the type of individual a.

3.6.2 Soundness and Completeness

A procedure is *sound* when there are no wrong inferences drawn from the knowledge base using the procedure. A sound procedure may fail to find solutions in some cases, even though they exist. A procedure is *complete* if its execution can obtain all the correct inferences from the knowledge base. A complete procedure may find solutions when there are actually no solutions. In other words, if the procedure is sound, and the answer found is affirmative, the answer can be trusted. On the other hand, if the procedure is complete, and the answer is negative, then the answer can be trusted.

There are many sound and incomplete algorithms which are considered as good approximation to resolve a problem because they can simplify the procedure to find a solution reducing the computational complexity. The pseudo model technique is one of these algorithms. Therefore, when the mergeable test is applied to resolve the subsumption of $C_1 \sqsubseteq C_2$, being C_1 and C_2 complex concept expressions and the following

conditions:

$$M_D^{C_1} \cap M_{\neg D}^{\neg C_2} = \emptyset$$
$$M_D^{\neg C_2} \cap M_{\neg D}^{C_1} = \emptyset$$
$$M_{\exists R}^{C_1} \cap M_{\forall R}^{\neg C_2} = \emptyset$$
$$M_{\exists R}^{\neg C_2} \cap M_{\forall R}^{C_1} = \emptyset$$

If all the conditions are kept true, meaning there is no interaction between the pseudo models, then the models can be merged which means the conjunction of the concept $C_1 \sqcap \neg C_2$ is satisfiable what implies that $C_1 \nsubseteq C_2$. If one of the conditions is false, which means there is an interaction between the pseudo model of C_1 and C_2. In this case, the Tableau algorithm must be run to test the satisfiability of the conjunction. In other words, if there is no interaction between the pseudo models, there is no need to execute the completion of the Tableau Model reducing the computational complexity of the query answering process.

3.6.3 Individual Realization using Pseudo Models

Let a be an individual in a consistent ABox A w.r.t. a TBox T, $\neg C$ be a satisfiable concept, M_a and $M_{\neg C}$ the pseudo models for the individual a and the concept $\neg C$ respectively. If the mergeable test returns true, meaning there is not interaction between the models, the ABox $A \cup \{\neg C(a)\}$ is consistent, so a is not an instance of C.

Individuals in an ABox belong only to a few concepts, therefore, the proof of contradiction will not derive a clash, which means in most of the cases, a is not an instance of C. As for the mergeable test, if a is not an instance C, there is not interaction between the pseudo models. Since the mergeable test is a sound but incomplete, the affirmative result can be trusted. There is no need to execute the completion of the Tableau model for the majority of the individuals in the ABox.

As an example, suppose the following knowledge base and query:

TBox: $C \equiv \exists R.Y$
ABox: $R(a, b),\ Y(b)$

Query: $C(a) = ?$

The pseudo model for a and $\neg C$ corresponds to:

$$M_a = \begin{array}{l} M_a^A = \phi \\ M_a^{\neg A} = \phi \\ M_a^{\exists} = \{R\} \\ M_a^{\forall} = \phi \end{array} \qquad M_{\neg C} = \begin{array}{l} M_{\neg C}^A = \phi \\ M_{\neg C}^{\neg A} = \{C\} \\ M_{\neg C}^{\exists} = \phi \\ M_{\neg C}^{\forall} = \{R\} \end{array}$$

There is an interaction between M_A and $M_{\neg C}$, the two pseudo models are unmergeable. However, if the algorithm is applied recursively, the mergeable test for M_b and M_y can be performed.

$$M_a = \begin{array}{l} M_b^A = \{X\} \\ M_b^{\neg A} = \phi \\ M_b^{\exists} = \phi \\ M_b^{\forall} = \phi \end{array} \qquad M_{\neg C} = \begin{array}{l} M_{\neg Y}^A = \phi \\ M_{\neg Y}^{\neg A} = \{Y\} \\ M_{\neg Y}^{\exists} = \phi \\ M_{\neg Y}^{\forall} = \phi \end{array}$$

There is a no interaction between the pseudo models M_b and M_y, therefore they are mergeable which can lead to the conclusion that a is not an instance of C.

Chapter 4

System Architecture

This chapter presents the architecture and design used during the prototype implementation of a MapReduce based DL query engine.

Design an architecture that scales to the requirements of the Semantic Web is a challenging thing to do. Just thinking about a "Supercomputer" that could store and process all the information on the web is far beyond reality. There is not one single machine that could handle even a small fraction of then information that exists in the web. However, during the last years, Cloud computing has brought hope to applications that need to scale beyond single machine architectures reducing the hardware requirements of complex algorithms executions.

The proposed architecture exploits the facilities that MapReduce and Hadoop provide to distribute the execution of repetitive tasks over large datasets. Specifically, MapReduce is used to distributedly execute the knowledge base import process, create a pseudo model cache for all the individuals in the ABox, extract the ABox candidate individuals for a given concept expression using the pseudo model technique and, finally, MapReduce is used to process the completion of the Tableau algorithm that runs over all the candidate individuals from the ABox. Figure 4.1 introduces the overall architecture of the system. Notice that all the components run inside a Hadoop cluster, not only providing access to HBase and HDFS but also allowing the execution of MapReduce jobs in the cluster. In the following sections each module is explained with more detail.

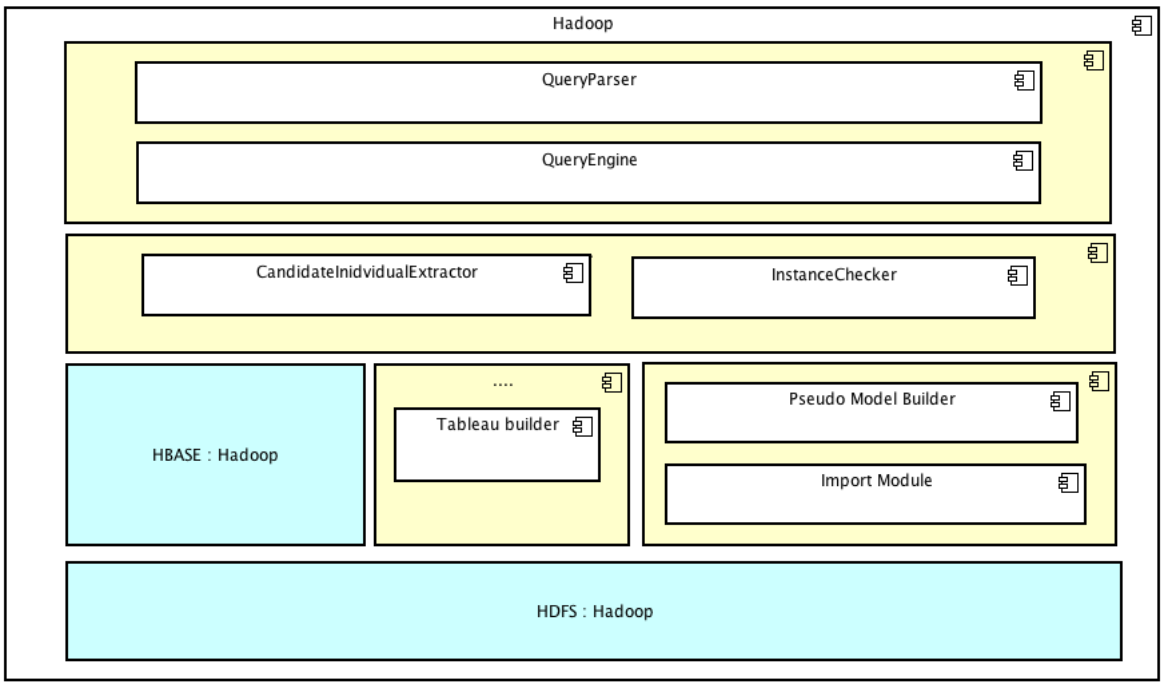


Figure 4.1: Overal System Architecture

4.1 Overview

The purpose of the section is to introduce the overall architecture of the system before explaining the details of how each module works, presenting an overview of the query answering process.

Before being able to answer queries the system needs to import the knowledge base that will be used. The initial import process loads, both TBox and ABox data, from a list of files in a specified path in the HDFS filesystem, indexing and encoding the information in a distributed database. Once the whole knowledge base has been imported an optimization process is executed that creates the Pseudo Models for all the individuals in the ABox. After the execution of the import process is complete the system is ready to answer DL queries from the user.

The prototype implementation supports the execution of Instance Retrieval queries. Instance Retrieval queries answer the question of which individuals, mentioned in the knowledge base, are instances of a concept expression C. As explained in section 3.5, Instance Retrieval queries are resolved by applying the Instance Checking algorithm to

all[1] the individuals in the ABox. Because of the inherited repetitive nature of the Instance Retrieval algorithm a MapReduce job is used to distribute its execution.

When a query is submitted to the system a Tableau model is created for the concept expression representing the query. A MapReduce job uses the query expression pseudo model and the cached individual pseudo models to identify which individuals must be checked using the instance checking algorithm. Once the candidate individuals are identified, a MapReduce Job performs the completion of the Tableau model for the concept expression representing the user query for all the candidate individuals in the ABox. Hadoop facilitates the efficient distribution of the individuals through the cluster. Figure 4.2 presents the Sequence Diagram of the query execution process.

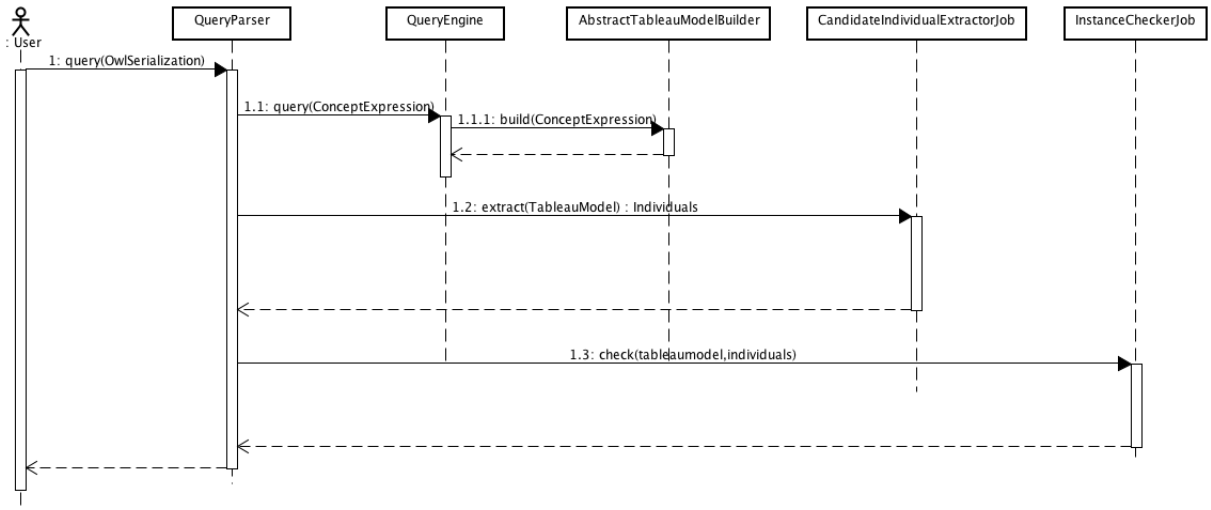


Figure 4.2: Query Execution Process

4.1.1 HBase Schema

HBase is used as persistence mechanism of the knowledge base inside the Hadoop cluster. The schema, presented on Figure 4.3, was created having on mind that HBase is not a traditional RDBMS system, because of this reason, Join operations add a lot of overhead to the system, therefore the schema tables don't follow the normalization rules[2] common in traditional relational database schemas.

[1] As explained later some techniques are used to discard individuals to check.
[2] http://publib.boulder.ibm.com/infocenter/idshelp/v10/topic/com.ibm.ddi.doc/ddi56.htm

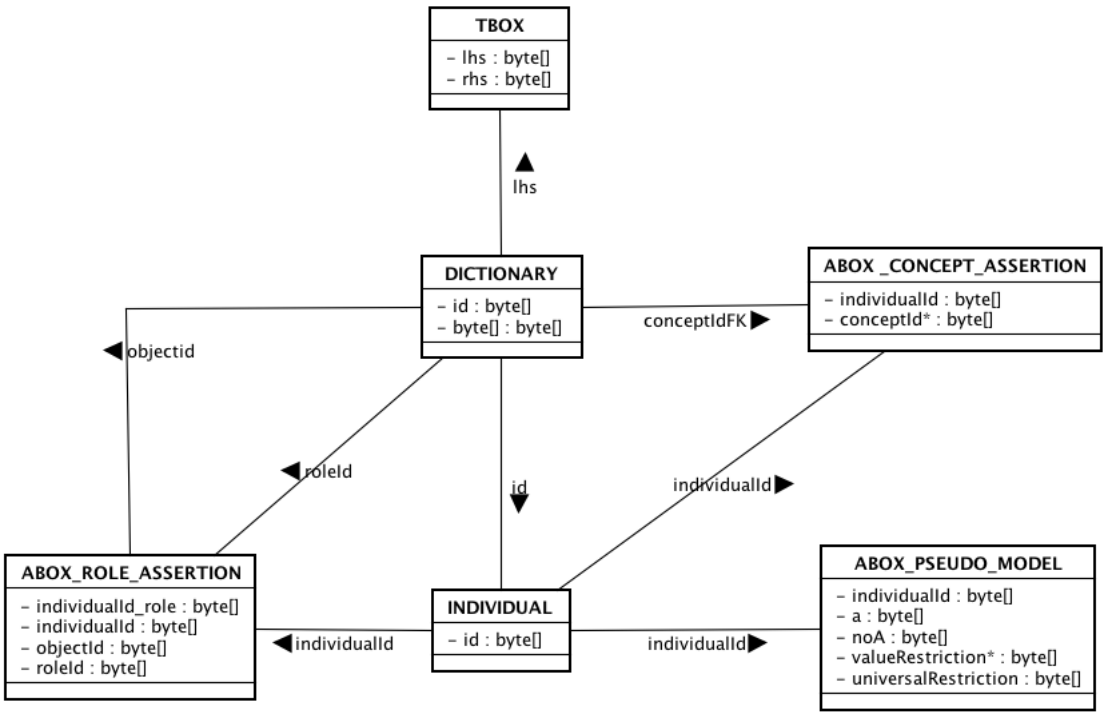


Figure 4.3: HBase Schema

- **DICTIONARY:** The *DICTIONARY* table contains the encoded values of all the resources mentioned in the knowledge base. Encoding the data is necessary due to the large size of data sets in the Semantic Web, applications that want to process data efficiently must implement an encoding process that reduces the total size of the data manipulated and the memory footprint of the reasoner algorithms. For this implementation, the id that represents a resource is the key of the dictionary table.

- **TBOX :** The *TBOX* table contains the list of all the equivalence or subclass axioms mentioned in the knowledge base. The data is saved in the table in a denormalized way because on the contrary, due to the HBase nature, the number of joins to perform would make the system unusable, an effect of this design decision is that every time an expression is loaded the system has to perform a parse operation that converts a serialized String axiom into a *mapo.common.concept.Concep*t expression using *mapo.common.parser.impl.ALCParser*. The table contains expressions of the type $C \equiv D$ and $C \sqsubseteq D$ using as key of the table left hand side of the axiom and the axiom type. For example, the axiom $C \equiv D \sqcap \exists R.D$ is persisted in HBase as { C + 'e', D .AND (.EXISTS R. (D)) } and the axiom

$C \sqsubseteq D \sqcap \exists R.D$ as { C + 's', D .AND (.EXISTS R. (D)) }.

- **ABOX_CONCEPT_ASSERTION and ABOX_ROLE_ASSERTION**. The *ABOX_CONCEPT_ASSERTION* table is used to persist ABox assertions of the form *C(a)*, using the individual *a* as key. The Multimap capabilities of HBase are used to save more than a concept assertion for a given individual. The reason the individual is used as key is because it facilitates the random access to obtain the asserted concepts to a given individual. The *ABOX_ROLE_ASSERTION* table contains role assertions of the type *R(a, b)*, using as key $\boldsymbol{R} + \boldsymbol{a}$. This key allows to easily load the objects *b*, that are related to an individual *a* given the role *R*.

- **INDIVIDUAL:** The *INDIVIDUAL* table contains the list of all the individual mentioned in the knowledge base ABox. The table allows to easily iterate over all the individual, mentioned in the ABox, during the execution of the reasoning services. The table is completely traversed during the extraction of the individual pseudo models and the execution of the instance checking algorithm.

- **ABOX_PSEUDO_MODEL:** The *ABOX_PSEUDO_MODEL* table contains a cache with the pseudo model associated of all the individuals mentioned in the ABox. The table is populated by the import module, the first time the system starts and it is used during the execution of DL queries to reduce the number of individuals to verify using the Tableau Algorithm. The natural key of the table is the individual hash code.

The access to all the tables is implemented in the Persistence layer using the DAO pattern[3] that allows the persistence mechanism to be changed easily without rewriting other layers of the system. A custom OWLObjectRenderer, *ALCRenderer*, is used to serialize the OWLOntology axioms to a string that could be easily parsed during posterior executions facilitating the persistence and debug process of the executions. The *ALCRenderer* uses the rules presented in the table 4.1 to perform the serialization process.

[3]Data Access Object: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

Axiom	Serialization Syntax
OWLSubClassOfAxiom	C .SUB D
OWLNegativeObjectPropertyAssertionAxiom	.NO (C)
OWLDisjointClassesAxiom	C .OR D
OWLClassAssertionAxiom	C(a)
OWLEquivalentClassesAxiom	C .EQ D
OWLObjectIntersectionOf	C .AND D
OWLObjectUnionOf	C .OR D
OWLObjectSomeValuesFrom	.EXIST R. (D)
OWLObjectAllValuesFrom	.FORALL R. (D)

Table 4.1: ALCRender syntax

4.2 Knowledge Base Importer Module

The objective of the Knowledge Base Importer Module is to facilitate the execution of the reasoning services, executed on top layers of the architecture. Specifically, the module encodes and indexes all the knowledge base, TBox and ABox data, and creates the Pseudo models for all the individual in the ABox as explained in section 3.6.1. The import module populates the tables of the HBase schema presented on the figure 4.3. The encoded key of a resource is found by executing the hash function[4] $h = S_0 * 31^{n-1} + S_1 * 31^{n-2} + ... + S_{n-1}$. The function h generates an unique hash code that represents a resource from the knowledge base. During the import process the hash codes for all the mention resources are calculated and persisted; when the system is answering a query, the dictionary table is used to translate the hash codes into their original values. The import process may take several minutes or hours to complete depending of the size of the knowledge base and the network topology of the cluster used.

The import process receives as input a directory containing a set of OWL files with the complete knowledge base to process. To read all the files, a new hadoop FileInputFormat and RecordReader, *OWLFIleInputFormat* and *OWLFileRecordReader* were implemented. These classes are responsible for parsing the OWL files and creating a valid input for the MapReduce Mappers. The OWL files are read using the open source framework OWL-API[5]. For each of the configured mappers a new *OWLOntolgyWritable* object is generated that contains a fragment of the the knowledge base to process. Each

[4]http://www.javamex.com/tutorials/collections/hash_function_technical.shtml
[5]http://owlapi.sourceforge.net/

of the mappers configured receives an *OWLOntolgyWritable* that contains the abstraction of one of the owl files in the input directory.

4.2.1 Import Mapper

The *OWLImportMapper* class is responsible for processing the ontology's files represented in the *OWLOntolgyWritable* object. The input of the mapper consist of a tuple of the type *<NullWritable, OWLOntologyWritable>*, its input key is not relevant, therefore a NullWritable value is used as wildcard; the tuple's value contains the abstraction of the OWL file to process.

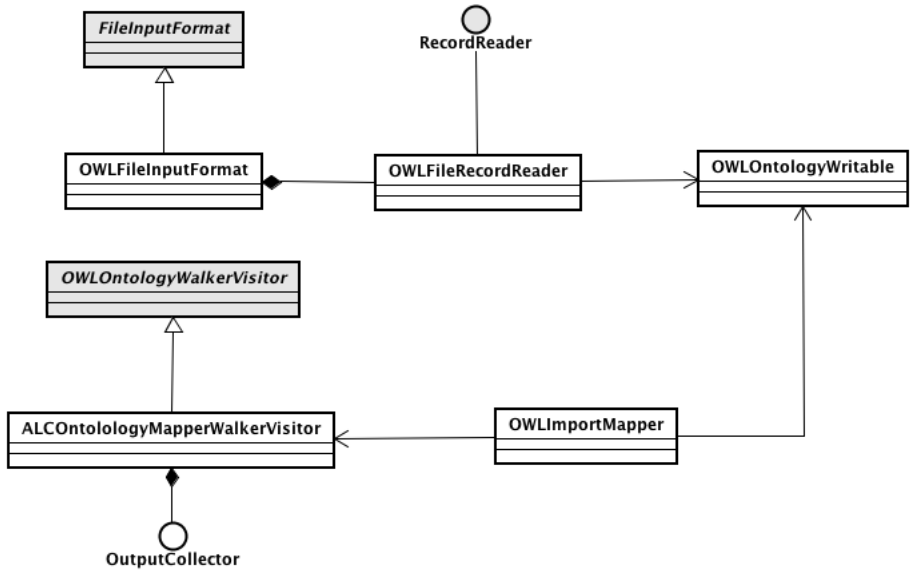


Figure 4.4: Import Mapper

- The mapper uses a Visitor[6], *ALCOntologyMapperWalkerVisitor*, to traverse the graph that represents an OWLOntology. The responsibility of the *ALCOntologyMapperWalkerVisitor* is to extract those RDF and OWL axioms present in the OWLOntology that correspond to the ALC family language and other elements required by the reasoner services. The *ALCOntologyMapperWalkerVisitor* acts as a Translator that receives an OWLOntology and collects the output to be processed by the configured Reducers. Table 4.2 presents an overview of the axioms and the output tuples generated by the *ALCOntologyMapperWalkerVisitor*. The tuples correspond to the output generated by each of the mappers configured in the execution, because each mapper processes the equivalent to an OWL file in

[6]Visitor Pattern: http://sourcemaking.com/design_patterns/visitor

the knowledge base, the *ALCOntologyMapperWalkerVisitor* generates multiple outputs in the same mapper allowing Hadoop to sort and group the output tuples by its key. The hash value of the resources mentioned corresponds to the hash function one introduced on 4.1.1.

OWL Constructor	**Output Tuple**
OWLEquivalentClassesAxiom	<hash(C), ".EQ hash(D)">
OWLSubClassAxiom	<hash(C), ".SUB hash(D)">
OWLClassAxiom	<C,C>
OWLObjectProperty	<r, C>
OWLClassAssertionAxiom	<hash(a), hash(C)>
OWLAnonymousIndividual	<a, ->
OWLNamedIndividual	<a, ->
OWLDataPropertyAssertionAxiom	<hash(a), R(hash(b))>
OWLObjectPropertyAssertionAxiom	<hash(a), R(hash(b))>

Table 4.2: *ALCOntologyMapperWalkerVisitor* Axioms

4.2.2 Import Reducer

The *OWLImportReducer* class is responsible for processing the tuples collected by the *OWLImportMapper* as shown in the Table 4.2. The reducers receive as input tuples of the type <Text, [Text]*> and, after identifying the type of element to process, the reducer persist the information to HBase.

The reducers identify the tuples type by checking each of the tuple's values. A tuple containing a TBox concept is identified by selecting the tuples whose value contains the string constants *'.EQ'* or *'.SUB';* once identified, the value from a TBox tuple is parsed to validate that represents a valid ALC concept and persisted in the system using the *TBoxDao* interface.

If the tuple is not identified as a TBox tuple then the reducers check if the tuple belongs to the ABox. ABox role assertions tuples are identified checking if the tuple's value contains a character '(' and ABox concept assertions are identified checking if both, the input key and the tuple value, are valid integer objects. Once an ABox tuple is identified, the *ABoxDao* interface is used to persist the ABox data.

Tuples that have as key a string value correspond to ontology Classes or Role properties definitions. Both of them are added to the dictionary table. Tuples without a value correspond to mentioned ABox individuals. Individual tuples are added to both, the individual and dictionary table, using its correspondent dao classes.

When all the import reducers configured in system finish their execution, the schema contains the encoded and indexed knowledge in the Hadoop cluster. Reasoning services in above layers are now able to perform operations on the given data.

4.2.3 Pseudo Model Builder

After the knowledge base has been encoded and indexed the system performs an optimization process that allows the query engine to answer instance retrieval queries in a more efficient manner. The implemented algorithm follows the Pseduo Model technique introduced in Section 3.6.1.

A MapReduce job is used to find the set of completions C, for each individual in the ABox. The *PseudoModelBuilderMapper* is responsible for creating a cartesian product between all the individuals in the ABox with all the expressions in the TBox.

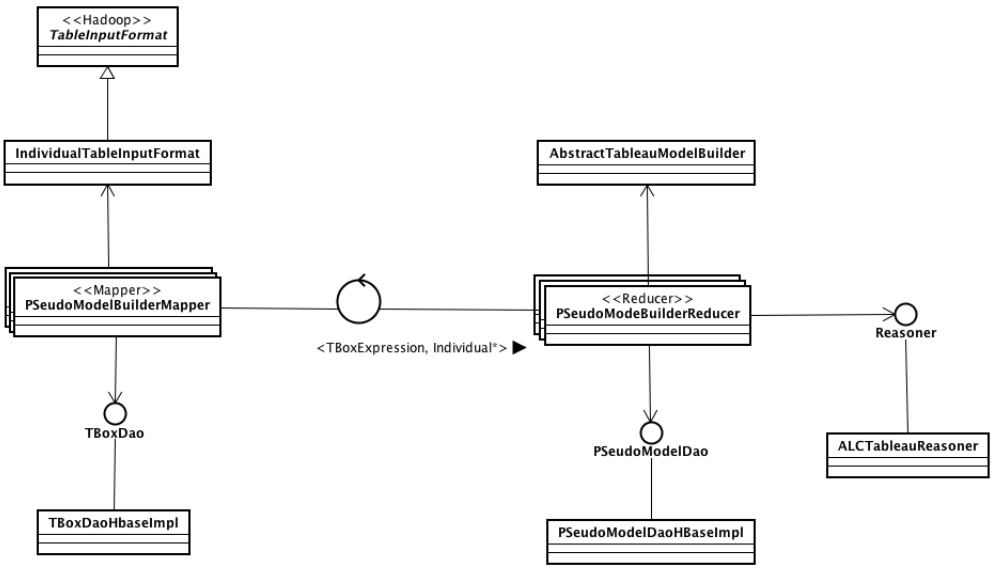


Figure 4.5: Pseudo model builder class diagram

The Execution of the algorithm starts by obtaining all the Individuals mentioned in the ABox. This is done by implementing a new Hadoop *TableInputFormat* that allows to convert HBase tabular data into a format that is consumable by MapReduce

jobs. When the execution starts, the *PseudoModelBuilderMappers* receive a tuple: *<ImmutableBytesWritable, RowResult>*, whose value contains a row of the Individual Table. The mappers use the configured TBoxDao to obtain all the expressions of the the form: $C \subseteq D$ or $C \equiv D$. Each triple of the type $C \subseteq D$ found is converted to a concept expressions of the type: $\neg C \sqcup D$. Equivalently, triples of the type $C \equiv D$ are converted to two concept expression of the type $\neg C \sqcup D$ and $\neg D \sqcup C$.

The *PseudoModelBuilderMapper* mapper uses the generated concept expressions from the TBox and the input individuals to create output tuples of the form *<ConceptExpression, Individual>*. The concept expression must be used as key of the output tuple, allowing Hadoop to group the common keys. The *PseudoModelBuilderReducer* receives as input tuples of the form *<ConceptExpression, [Individual]*>* and finds the pseudo models for the input expression and individual. The reducer finally persist the pseudo models to the HBase table *individual_pseudo_model*. Figure 4.6 presents the sequence diagram that follows the execution of the algorithm.

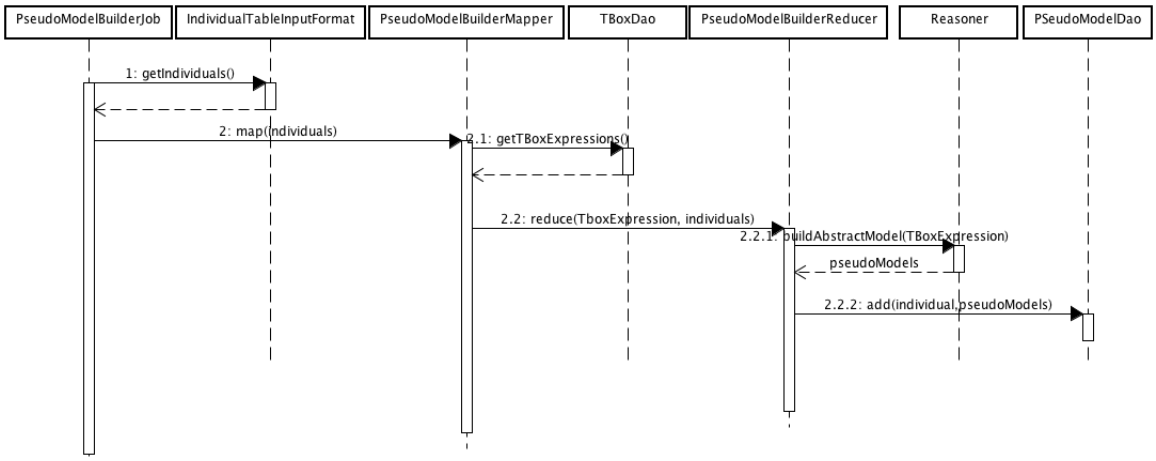


Figure 4.6: Pseudo Model Creation Sequence Diagram

4.3 Reasoning Module

The reasoning module receives a concept expression in Normalized Negation Form (NNF) to create a Tableaux model and execute an instance checking algorithm to selected individuals in the knowledge base. In the worst case, all the individuals in the knowledge base must be validated trough the instance checking algorithm but, as explained in section 3.6.1, optimization technics are applied to detect obvious clashes discarding individuals to check.

A MapReduce job is launched, to distributively execute the instance checking algorithm, where each mapper receives as input a candidate individual to check and the tableau model of the specified concept expression[7]. The instance checking algorithm uses the knowledge base ABox to find clashes with the concept expression. Additionally, the process saves a cache of the individual model extracted from the ABox. The cached model is used in posterior executions to identifying the candidate ABox individuals to process. Figure 4.7 shows, graphically, the execution of the process. In the following sections each process will be explained with more details.

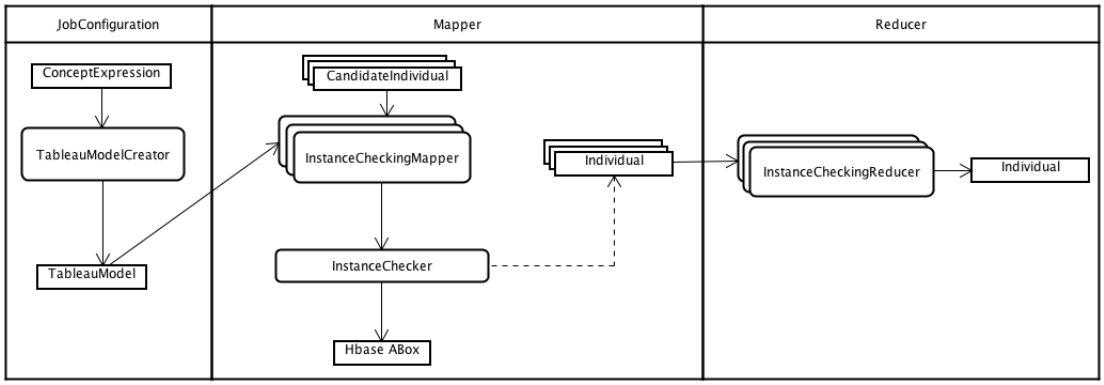


Figure 4.7: Query Answering Process

4.3.1 Tableau Model Creation

The first thing the reasoning module does is to run a Tableau algorithm to obtain the associated model from the specified concept expression. The process receives a concept expression in NNF and uses an and-or tree to create the Tableau model using the ALC completion rules from Figure 3.4. Each node in the tree contains the properties:

- **Content*:*** Set of concepts in the node.
- **Individual:** Represents the key of the Node.
- **Status:** Specifies the currents status of the node. The possible values are: unexpanded, expanded, satisfiable, unsatisfiable .
- **Type:** Specifies the type of node. The possible values are: and-node, or-node.
- **Parent:** Parent node of the current node.
- **Children:** Set of children of the node. Each node is associated with an edge.

[7]The concept expression is specified as configuration parameter of the job.

The type of the node is determined by the rules applied in the node content. Specifically, if a $\sqcup - rule$ is applied the node becomes an or-node. The type of node determines the way the satisfiability of the node is calculated. An and-node requires all its children to be satisfiable, on the other side, an or-node is satisfiable if at least one of its children is satisfiable. Figure 4.8 presents the pseudo code for the creation of the model:

```
Initialization: set i=0, E = Concept expression;

create n = <i, E, unexp, and-node>;
while (n.status not in (sat, unsat)){
  if (n.content not rules to apply){
     n = n.parent;
     if (n.content has clash){
        n.status = unsat;
     }else{
        for each (c on n.children){
           if (n.type = and-node){
              n.status = c.status and n.status;
           }else{
              n.status = c.status or n.status;
           }
         }
      }
    }
  }else{
     if (n-rule){
        n.content=new_content;
     }else if (u-rule){
        n.type=or-node;
        create w = <i,E1, unexp, and-node>;
        create z = <i,E2, unexp, and-node>;
        n.children = n.children + {w, z};
        n = w;
     }else{ //ER.C or VR.C
        create y = <i++,E2, unexp, and-node>;
        n.children = n.children + edge-R(y);
        n = y;
     }
  }
}
```

Figure 4.8: Tableau Model Creation Pseudo-Code

4.3.1.1 Lazy Unfolding

The Tableau algorithm implemented uses lazy unfolding to expand the concept expressions that appear in a node's content after applying the Tableau completion rules from figure 3.4. Unfolding allows to eliminate from a concept expression C, whose satisfiability is going to be tested with respect to the TBox T, all the concept names occurring in T using a recursive substitution procedure.

The unfolding process works a follows: for a non primitive concept A, defined in T by an axiom $A \equiv D$, the procedure substitutes A with D wherever it occurs in C and then the procedure is repeated to unfold D. For a primitive concept name A, defined in T by an axiom $A \sqsubseteq D$, the procedure is more complex. Wherever A occurs in C is substituted with a concept $A' \sqcap D$, where A' is a new concept name that does not occur in T or C, and D is then recursively unfolded. The concept A' represents the unspecified characteristics that differentiate it from D.

Although the Tableaux algorithms generally assume that the concept expression to be tested is fully unfolded, in practice it is usual to unfold a expression only when required by the execution of the algorithm. For example, if T contains the definition $A \equiv C$ and the $\sqcap - rule$ is applied to a concept $(A \sqcap D) \in L(x)$ so that A and D are added to *L(x)*, then A can be unfolded by replacing it with C. In this way, lazy unfolding avoids the unnecessary expansion of irrelevant subconcepts because contradictions could be discovered without fully expanding the tree. A greater increase in efficiency is achieved by retaining the names when their definitions are added, instead of substituting them. The efficiency is achieved because the discovery of a clash between concept names can avoid the expansion of their definitions. The algorithm implemented uses the expansion rules from Table 4.3.

Name	If	Then
$U_1 - Rule$	$A \in L(x)$ and $(A \equiv C) \in T$ $C \notin L(x)$	$L(x) \rightarrow L(x) \cup \{C\}$
$U_2 - Rule$	$\neg A \in L(x)$ and $A \equiv C \in T$ $\neg C \notin L(x)$	$L(x) \rightarrow L(x) \cup \{\neg C\}$
$U_3 - Rule$	$A \in L(x)$ and $(A \sqsubseteq C) \in T$ $C \notin L(x)$	$L(x) \rightarrow L(x) \cup \{C\}$

Table 4.3: Lazy unfolding expansion rules

The unfolding algorithm implementation access the *TBOX* table to expand a given concept. The expansion process uses the hash, of the concept to unfold, as index key.

The TBoxDao implementation is responsible for obtaining the list of subclass and equivalence axioms present in the knowledge base. The TBoxDao uses the index *h(C) + 'E'* obtaining as result the equivalent concept expressions *and h(D)+'s'* to look for any subclass axioms.

4.3.2 Candidate Individuals Selection

When an instance retrieval query is posted to the system, the naive approach to find which individuals are valid answers for the query consist on finding the Tableau model completion for all the individuals in the system. This would be inefficient because normally individuals in the ABox are associated with only a few concepts in the TBox. As explained in Section 3.6.1, pseudo models are used to reduce the number of ABox individuals to check every time a query is answered.

The candidate selection algorithm works as follow. For a concept expression, Q, that represents a query to be answer, the algorithm finds the pseudo model, M_Q, for the concept expression $\neg Q$. Once M_Q is found, a MapReduce job is used to find the interaction between M_Q and the cached pseudo models, M_x, for every individual x mentioned in the ABox found during the import process (Section 4.2.3).

The MapReduce mappers receive as input all the cached the pseudo models from the individual in the ABox using a Hadoop *TableInputFormat*. The mappers receive the pseudo model for the query concept expression as a parameter, this parameter remains constant for all the mappers in the MapReduce job. The input tuples for the mappers have the format *<individual, [*M_a^A,$M_a^{\neg A}$, $M_a^{\exists}$,$M_a^{\forall}$*]>*. Each mapper tries to find if there is an interaction between the pseudo models of the configured concept expression and the mapper input for each individual in the ABox. If an interaction is found, the pseudo models are unmergeable, therefore the Tableau Model completion has to be performed for the current individual being tested by the pseudo model. The Mappers generate an output tuple for every individual that has to be checked using the Tableau algorithm. The output tuples are of the form *<individual, null>*.

The reducer receive as input tuples, equivalent to the output of the reducer, with the format *<individual, null>*. The reducers generate don't perform any specific job, they let Hadoop generate a list of files in the HDFS file system that contain the list of candidate individuals to be checked. Figure 4.9 presents the flow of the execution.

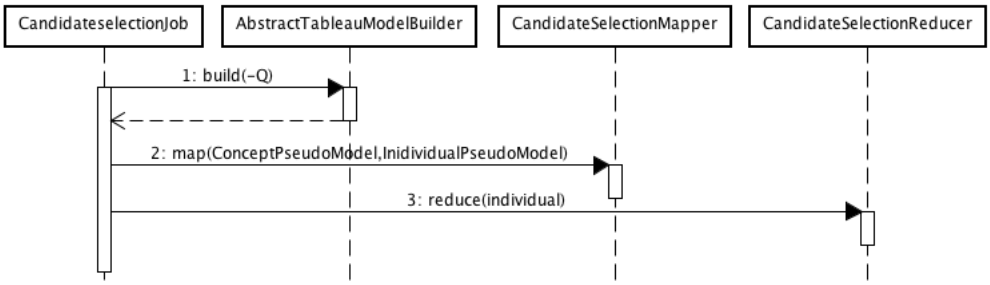


Figure 4.9: Candidate Individual Selection Sequence Diagram

4.3.3 Instance Checking Algorithm

The objective of the instance checking algorithm is to verify the satisfiability of a concept expression with respect to an individual in the ABox. The algorithm receives as input a Tableau model, that corresponds to a complex concept expression and a list of candidates individuals from the ABox to which the algorithm will be applied.

The Instance checking algorithm runs using a MapReduce job that allows the algorithm to be distributed through all the nodes in the Hadoop cluster. Hadoop and MapReduce dramatically could improve the performance and scaling capabilities of the algorithm compared with traditional reasoners. In traditional reasoners, the instance checking algorithm is typically executed in single machine, limiting the algorithm to the characteristics of a specific hardware. In the proposed architecture, allows the checking of the individuals to be executed in parallel scaling the execution of the algorithm and removing any single point of failure in the architecture.

Specifically, the implementation of the Instance checking algorithm uses an Tableau tree to check for clashes of a specific individual in the ABox. The algorithm uses the indexes created in the knowledge base importer module, 4.2, to fast access the information in the ABox, executing the completion model for the Tableau Model of a concept expression for each candidate individual. The instance checking process is best explained with an example.

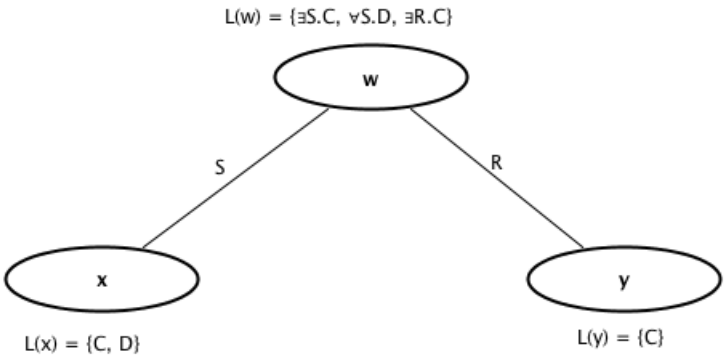


Figure 4.10: Tableau model for concept expression $\exists S.C \sqcap \forall S.\neg D \sqcap \exists R.C$

The algorithm to execute the instance checking for the individual a Given the Tableau model from figure 4.10, that corresponds to the concept expression $\exists S.C \sqcap \forall S.\neg D \sqcap \exists R.C$.

ABox = { S(a,b), D(b), R(a,c), C(c) }

Figure 4.11: Sample ABox

Given the ABox from Figure 4.11, the algorithm checks every individual mentioned in the ABox[8], therefore the Tableau Completion has to be run for the individuals *a,b,c*. After the algorithm runs, only the model for individual a is open w.r.t to the ABox, as shown in Figure 4.12. Therefore, individual a is a valid answer for the query $\exists S.C \sqcap \forall S.\neg D \sqcap \exists R.C$.

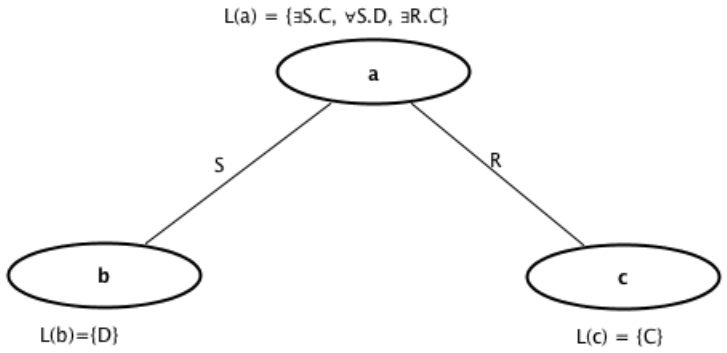


Figure 4.12: Tableau model completion for Indiviudal a

[8]For this example the pseudo model technique is not being applied

4.4 Query Engine Module

The query engine module is responsible for parsing a concept expression that represents a query and orchestrates the execution of the query using the reasoning services explained in Section 4.3.

4.4.1 Query Parser

The query parser is responsible for converting a serialized OWL concept expression in a format that could be processed by the system. The instance retrieval queried posted to the system are expressed, as any other concept expression, in one of the popular OWL Serialization formats: XML/OWL or Turtle. The OWL-API framework[9] is used to parsed the input file and obtain an in-memory representation of the concept expression. With the memory representation of the concept expression to answer, the query parser delegates the control to the query executor.

4.4.2 Query Executor

The query executor orchestrates the execution of the query answer process for a concept expression. The query executor receives an abstraction of a concept expression and, using services from the reasoning module, obtains the pseudo model for the query concept expression. With the pseudo model of the concept expression, the query executor calls the method to obtain the candidate individuals, based on the cached pseudo models, from the knowledge base ABox. Once the candidate individuals are identified, the query executor executes the process responsible for executing the completion of Tableau models for the concept expression that represents the query for every selected candidate individual. The query executor reports the answers to the user once the process has been completed.

[9]The OWL API is a Java API and reference implmentation for creating, manipulating and serialising OWL Ontologies. (http://owlapi.sourceforge.net/)

Chapter 5

Evaluation

In this chapter the performance results for the proposed architecture are presented and analyzed. Initially the chapter introduces the test data used followed by the environment used to conduct the test and the results obtained.

5.1 Lehigh University Benchmark for OWL

The Lehigh University Benchmark (LUBM)[13] is a benchmark developed at the Lehigh University for testing the performance of ontology management and reasoning systems. The ontology describes organizational structure of universities and it is relatively simple: it does not use disjunctions or number restrictions, but it does use existential quantifiers. Due to the absence of disjunctions and equality, query answering on LUBM can be performed deterministically.

The benchmark for OWL consists of the following:

- A plausible OWL ontology named univ-bench[1] for the university domain.
- Repeatable synthetic OWL data sets that can be scaled to an arbitrary size. Both the uni-bench ontology and the data are in the OWL Lite sublanguage.
- Fourteen test queries that cover a range of types in terms of properties including input size, selectivity, complexity, assumed hierarchy information and assumed inference.
- A set of performance metrics including data loading time, repository size, query response time, and query answer completeness and soundness. With the exception of completeness, they are all standard database metrics.

[1] `http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl`

- A test module that allows to generate test knowledge bases for the ontology.

Figure 5.1 presents graphically the asserted model generated using Protégé.[2] The figure helps to visualize the model represented by the LUMB ontology.

Figure 5.1: LUMB Ontology - Asserted Model

[2] http://protege.stanford.edu/

5.2 Performance Evaluation

The goal of the the performance evaluation was to test the scalability of the proposed architecture and not the algorithms implemented, that is, to see how the architecture performs depending on the amount of data and the number of machines running in the Hadoop Cluster. The reason for focusing in benchmarking the architecture and not the algorithms implemented is because there exist already more efficient DL algorithms than the ones implemented that make use of some of the optimizations mention in Section 6.2. Also, due to the MapReduce batch processing nature and its execution environment, it is safe to suppose that if the algorithms are optimized then the overall performance of the system will improve with them. Because of this reason the results presented in this section should be taken qualitatively, rather than quantitatively.

5.2.1 System Setup

The test were performed using three different Hadoop clusters configurations in order to determine how the performance of the architecture is affected by adding nodes to the cluster. The Hadoop clusters used for the benchmarks were configured using 2, 5 and 7 machines all of them with a 2 GHz Intel processor, 2 GB of RAM, running Linux Ubuntu 10.4, Hadoop 0.19.2 and HBase 0.19.3. The Java virtual machine used was Sun's Java 1.6 with the VM memory limited to 1500 MB.

Two different test datasets were created using the test module of the LUBM benchmark. Table 5.1 summarize the details of the data sets used during the benchmarks.

Name	Universities	$C \sqsubseteq D$	$C \equiv D$	Domain	Range	Instance Number
Dataset 1	10	36	6	25	18	1'298.988
Dataset 2	20	36	6	25	18	2'790.579

Table 5.1: LUMB test data

5.2.2 System load time

As explained in Section 4.2, during the import process the system encodes the knowledge base, creates the pseudo models for all the individual and saves the information in the distributed persistence mechanism, HBase.

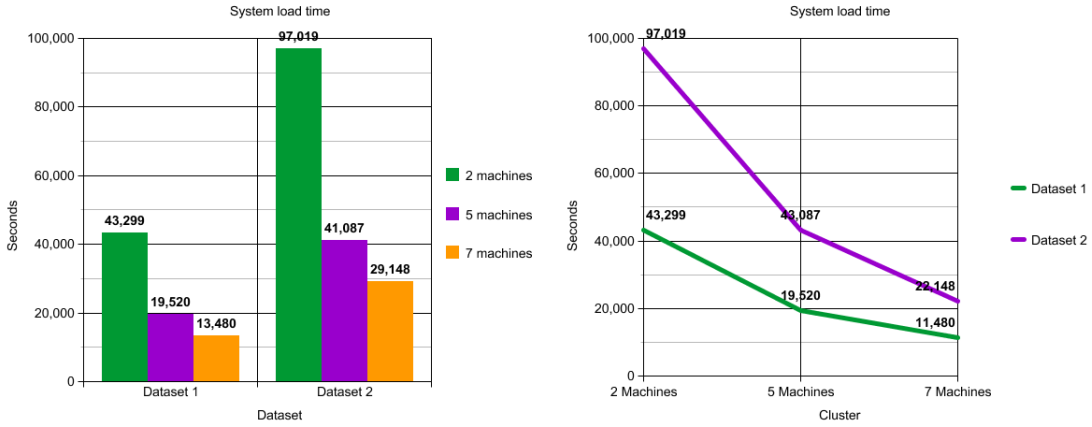


Figure 5.2: System load time

Figure 5.2 presents the overall time that it takes to each cluster to import the complete knowledge base. From the figure we can notice how the performance of the import process improves, reducing the completion time, by adding machines to the Hadoop cluster. The increased performance of the process can be explained due to the fact that the execution of the algorithms used during the import process is done distributedly and independently across the machines in the cluster. It's also interesting to notice that when a new machine is added to the cluster there is a slight penalty introduced. The Tables 5.2 present the penalty index introduced for the import process, loading both test datasets. The penalty index follows the formula: $P_i = 1 - \frac{T_i}{T_e}$, where T_e is the expected execution time, found using the formula: $T_e = \frac{T_r * n_r}{n_i}$, T_r is the reference time that corresponds to the total time used by the reference cluster to complete the process, and T_i corresponds to the total time that takes to the cluster i to complete the process.

	2 Machines	5 Machines	7 Machines
Penalty Index	-	0.11	0.08
Penalty per machine	-	0.02	0.01

(a) Penlaty index Dataset 1

	2 Machines	5 Machines	7 Machines
Penalty Index	-	0.05	0.05
Penalty per machine	-	0.01	0.01

(b) Penalty index Dataset 2

Table 5.2: System load Penalty Index

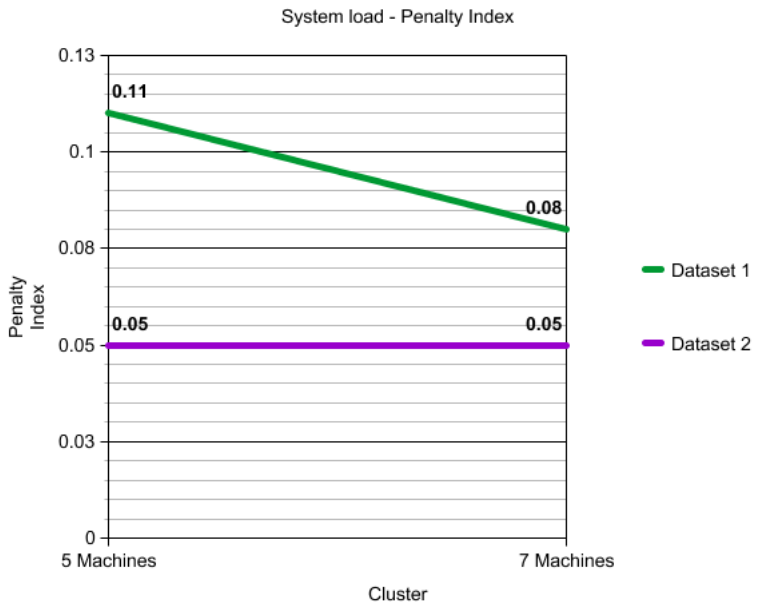


Figure 5.3: System load penalty index

Due to the small size of the of the test clusters the penalty index is not significant but it is expected to grow with the size of the Hadoop cluster. When the penalty index per machine is too big the executions times of the processes running inside the cluster will increase. However, when the knowledge base is too big and the memory available per machine is reduced, it could be desirable to increase the size of the cluster beyond the optimum penalty index to allow the processes to complete, even if it takes a longer theoretical completion time.

5.2.3 Query Answering Time

The implemented prototype architecture supports answering instance retrieval queries, because of this reason, not all of the 14 test queries provided by the LUBM benchmark can be used to execute the benchmarks. Two instance retrieval queries from the benchmark can be used to test the architecture. Those queries are:

- $Q_1(x) \equiv UndergraduateStudent(x)$: The query assumes both the explicit subClassOf relationship between *UndergraduateStudent* and *Student* and the implicit one between *GraduateStudent* and *Student*. It has large input and low selectivity.

Figure 5.4: Query 1 - Asserted Model

- $Q_2(x) \equiv Student(x)$: This query has a large input and low selectivity and does not assume any hierarchy information or inference.

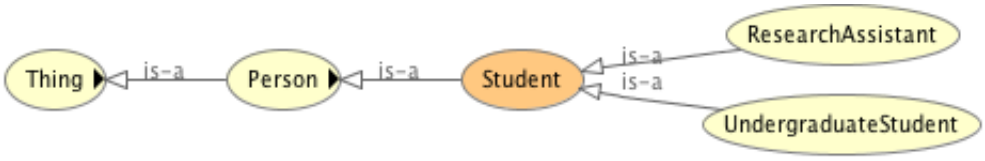
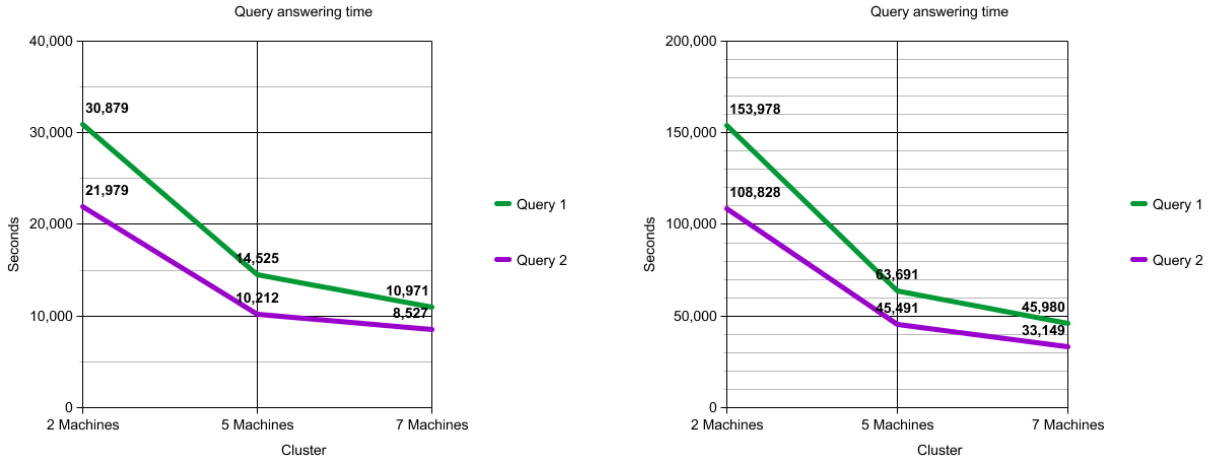


Figure 5.5: Query 2 - Asserted Model

The benchmarks for the query answering time were obtained executing every test query against the two test datasets on the three Hadoop clusters introduced in Section 5.2.2. To take into account the cache optimizations included in the architecture, each of the three queries were executed ten times consecutively and the average time was used as final query time.



(a) Dataset 1

(b) Dataset 2

Figure 5.6: Query answering time

In terms of query answering we can see from Figure 5.6 that the architecture has a similar behavior as observed during the import process. Adding machines to a cluster reduces the query answering time for each of the test queries. In fact, the scalability

increases in a rate similar for both queries. However, as observed during the import process, a penalty index is introduced for every machine added to the Hadoop cluster.

What can be concluded from the benchmarks is that the use of Hadoop to distribute the execution of both, the import process and the query processing affects the performance of the architecture positively. The batch processing characteristics of Hadoop allow the DL algorithms to be executed concurrently not only reducing the overall execution time but allowing the architecture to process large knowledge bases.

The factors that influence the penalty index are of different kinds most of them being do to network traffic, overhead added by Hadoop and the HBase persistence mechanism. This factors can, in some measure, be tweaked to reduce the penalty index per machine allowing the overall system to perform better. For example, the network infrastructure should be as optimized as possible having the Hadoop cluster aware of the network configuration, the Hadoop HDFS system should also be tuned according to the network topology used and with an optimum block size. Increasing the RAM used by the java vm of the machines in the Hadoop clusters can also reduce the penalty index given the fact that the Hadoop engine requires memory to coordinate the execution of processes in the cluster. The Hadoop community has collected some resources with tips that could be used to improve the performance of Hadoop clusters.[3]

[3] `http://wiki.apache.org/hadoop/PerformanceTuning`

Chapter 6

Conclusions and future work

This chapter summarizes the contributions of the project and proposes possible directions for future work.

6.1 Conclusions

This thesis was focused on creating an architecture that could make use of the advantages offered by the MapReduce programming model to create a query engine for massive and distributed ontologies knowledge bases. The proposed architecture was designed taking into consideration the resent proliferation of Cloud based services like Amazon's Elastic Compute Cloud (Amazon EC2) that have democratized the possibility to have robust environments using a cluster of machines to execute complex algorithms. The architecture was specifically designed thinking on how to use those clusters of machines with an already successful programming model, MapReduce, to execute the already proven algorithms required to answer DL queries. The architecture allows the knowledge base data to be distributed across different machines using a distributed persistence system, HBase, and the distribution of the execution of the reasoning algorithms, required to answer DL queries, allowing the execution to scale by dynamically distributing the processes through the configured machines in a cluster.

The power of the MapReduce programming model shines when one algorithm has to be applied many times and this is exactly the type of execution characteristic of the Instance Checking algorithm where all the individuals mention in the ABox must be checked. As shown in Chapter 5, the query answering time is reduced by adding machines to the Hadoop cluster. However, one problem that appears is that the access to the knowledge base persistence mechanisms is costly. This is not a simple problem to

solve using MapReduce because the programmer does not have control of the machines where the data is persisted. One optimization, implemented in this project, to reduce the persistence access times, consists on encoding the knowledge base and creating a cache of the TBox expression on every machine in the cluster. The disadvantage of this solution is that the machines running in the Hadoop cluster will require more memory, depending of the size of the TBox.

However, although MapReduce allows the system to scale beyond what traditional architectures could do, its programming model introduces some restrictions that had to be taken into consideration during the design of the query engine architecture. The most critical restriction is that in order to scale and exploit the scaling power of MapReduce, the tasks executed by the mappers and the reducers must be completely independent between each other. This gives MapReduce jobs a batch execution behavior that allows the execution of a MapReduce Job to be distributed across a cluster of machines. An implication of this is that MapReduce is not a reasonable alternative to create distributed algorithms. Specifically for the work on this thesis, MapReduce can not be efficiently use to create an execution of the Tableau Models that expands the Model distributedly across different machines. Such execution would require different control mechanisms not provided by any MapReduce execution environment and that go beyond the MapReduce programming model. For the prototype implementation in this project the Tableau Model construction was created using Threads allowing the model to be created more efficiently but with the hardware limitations of the machine where they run.

Another disadvantage is the need for a persistent mechanism. The initial HBase releases suffered terrible performance for random reads and writes, primarily because HDFS is not optimized for low latency random access but with the addition of a memcached-based[1] intermediate layer its performance has been improved. HBase servers, particularly the version using memcached, are memory intensive and generally require at least a gigabyte of memory per server. Also a smoothly performing HDFS filesystem is critical for the correct operation of HBase, any datanode instability will show up as HBase errors, this can become frustrating and cumbersome to solve.

6.2 Future Work

The prototypical implementation of the architecture supports DL instance retrieval queries but, in order for the system to be more usable, it needs to support more robust

[1] `http://memcached.org/`

queries. The mechanism to answer conjunctive queries could be implemented as presented on [15] where a *rolling up* technique is used to reduce the complexity of conjunctive queries with and without variables. The technique uses a simple transformation to convert every role term into a concept term. For example the role $< John, Bill >: Bother$ can be transformed into the equivalent concept $John : \exists Brohter.\ Bill$. Other additionally concept terms can be rolled up into the rolled up concept term. For example, the conjunction: $< John, Sally >: Parent\ \wedge Sally : Female\ \wedge\ Sally : PhD$ can be transformed into $John : \exists Parent(Sally \sqcap Female \sqcap PhD)$. On queries without variables the transformation reduces the number of conjuncts what means that the number of satisfiability tests needed to answer the query will also be reduced. The technique requires a more sophisticated *rolling up* technique when the queries contain variables as expressed in the referenced paper.

The algorithm to obtain the pseudo model for concept expressions and ABox could also be improved to use a deep pseudo model technique. On [16] a Deep Model Merging technique is presented and its advantages against the Flat pseudo model technique are empirically demonstrated.

Other optimization to be implemented could be classified in two categories: techniques to improve the subsumption testing algorithm and techniques to reduce the number of subsumption tests. Some of those techniques are:

- **Normalization and Encoding:** the technique detects structurally obvious satisfiability and unsatisfiability and enhance both the efficiency and effectiveness of other optimizations.
- **GCI Absorption:** the technique works by absorbing GCI axioms into primitive concept introduction axioms. This is a novel technique which can eliminate most of GCIs from a terminology which are a major cause of intractability.
- **Boolean Constraint Propagation:** the technique is used to maximize deterministic expansion of the Tableau model, pruning the search tree via clash detection before performing a non-deterministic expansion.
- **Heuristic Guided Search:** heuristic techniques can be used to guide the search in a way which ties to minimize the size of the search tree.
- **Dependency Directed Backtracking:** the technique adapts a form of dependency directed backtracking called back-jumping, which has been used in solving

constraint satisfiability problems. Back-jumping works by labeling concept expressions with a dependency set indicating the $\sqcup - Nodes$ on which they depend. The algorithm can jump back those nodes without exploring alternative successors.

Another step further would be to create a mixed architecture that uses the scaling power of MapReduce and the reasoner power of existing database based reasoners such as DLDB-OWL[2] or database-based Sesame[3]. Such reasoners already have implemented several optimization techniques that could improve drastically the query answering time. A special effort would have to be invested in creating a bridge between the reasoners and MapReduce and to be able to change its underlying persistence mechanism to HBase or more robust distributed persistent mechanism base on HBase such as Apache Cassandra[4].

[2] `http://swat.cse.lehigh.edu/downloads/dldb-owl.html`
[3] `http://www.openrdf.org/`
[4] `http://cassandra.apache.org/`

Bibliography

[1] R.Brachman, A.Borgida: Loading data into description reasoners. Volume 22. SIGMOD, 1993

[2] P. Bresciani: Querying database from description logics. In KRDB'95, 1995.

[3] T. R.Gruber : Toward principles for the design of ontologies used for knowledge sharing. International Journal of Human-Computer Studies, Volume 43.

[4] Jeffrey Dean and Sanjay Ghemawat : MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, 2006.

[6] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. RDFS Reasoning and Query Answering on Top of DHTs. The Semantic Web - ISWC 2008: 7th International Semantic Web.

[7] Peter Mika, Giovanni Tummarello. Web Semantics in the Cloud. The Semantic Web, IEEE 2008

[8] Jacopo Urbani, RDFS/OWL reasoning using the MapReduce framework. vrije Universiteit Amsterdam, 2009.

[9] Franz Baader, Diego Calvanese, Deborah L. McGuinness, and Daniele Nardi, R. Moeller. The Description Logic Handbook: Theory, Implementation and Applications. Sept. 24, 2007

[10] V. Haarslev and R. Moeller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In Ninth International Conference

on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5, pages 163–173, 2004.

[11] Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-memory hash tables for accumulating text vocabularies. Information Processing Letters.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. San Francisco, 2003

[13] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems.

[14] Bitton, D., DeWitt, D., and Turbyfill, C. Benchmarking Database Systems, a Systematic Approach. In Proc. of the 9th International Conference on Very Large Data Bases. 1983

[15] Ian Horrocks and Sergio Tessaris. A Conjunctive Query Language for Description Logic ABoxes.

[16] Volker Haarslev and Ralf Moeller. Optimizing TBox and ABox Reasoning with Pseudo Models, University of Hamburg, Computer Science Department.