Technische Universitaet Hamburg-Harburg
Institute of Software Systems

Master of Science Thesis

Distributed Storage System for Description Logic Knowledge Bases

by

Anh Ngoc Nguyen

Supervisor
Prof.Dr. Ralf Moeller
Prof.Dr. Karl-Heinz Zimmermann
Sebastian Wandelt

Hamburg, 2009

.....

Contents

Acknowledgements

I would like to express my appreciation to Prof. Ralf Moeller and Dipl. Sebastian Wandelt for giving me the oppotunity to work on this thesis in the Institute of Software System. And Mr. Wandel, I would like to send my special thanks to you for your time, your guidance, patience and understanding during the project. I would not be able to finish the project without the full support from you. I also would like to thank Prof. Zimmermann to be the second supervisor for my master thesis.
My gratitude also goes to my family and my friends, who always support and give me more courages during the work.

Chapter 1

Introduction

1.1 Description Logics and the idea of distributed storage

Description Logics are a family of knowledge representation languages, using the definition of concept description and individual to express an application domain. Having the logic based semantics, with the supporting for logical reasoning, Description Logics is widely used in Semantics Web, and the OWL-DL and OWL-Lite of the Web Ontology Language (OWL) is also based on Description logics.
For the last few years, the interest in Semantics Web has been strongly increased. The size of the assertional part becomes much and much bigger. Soon, the traditional in memory approach for reasoning using Tableau algorithm will be inefficient, since the completion tree for ABox is no more fit into the main memory. There have been several researches on this problem, and in this thesis we introduce an approach that divides the ABox into many smaller *islands* based on the information extracted from the terminology. After the ABox is divided, some islands will be load into the memory for solving a given reasoning problem instead of the whole ABox.
Further more, in this thesis we developed the proposed thoery into a distributed storage system for a Description Logics knowledge base, which supports updating and reasoning w.r.t both ABox and TBox.
The base algorithm in this thesis is extended from the island algorithm in the article [15] and the further development of update algorithm for partitioning in [16].

1.2 Structure of the report

The report is structured as following: Chapter 2 introduces some basic knowledge about description logics and the reasoning problem with respect to a knowledge base. Chapter 3 proposes an approach to the partitioning of assertionology and of the description logic ontology. The further development of partitioning, the decentralization of storage for ABox assertions is discussed in Chapter 4. Chapter 4 also describes the architecture of a system which was built to be a distributed storage system for a knowledge base. Some experimental results and evaluation are exposed in chapter 5. Chapter 6 summarizes the work which has been done in this thesis as well as propose future development of the approach.

Chapter 2

Description Logics

2.1 Introduction

Researches in the field of knowledge representation and reasoning has been focusing on formalizing high level descriptions of the world that can be effectively used in "intelligent" system. "Intelligent" refers to the ability of systematically finding the implicit meaning from the explicit represented knowledge. Since the 1970s, several approaches for knowledge representation have been evolved, including logic based formalisms and network structure representation.
Realizing that the network structures, such as *semantic network* and *frames*, being more appealing and effective in practical, can be given the logical semantics by relying on first order logic, reseach in the area of Description Logics began under the label of *terminological system*. In more recent years, with more development in the field, the term *Description Logics* became more popular. It is now getting much more interests and became the basis for many knowledge base representation systems.

2.2 Description Logics basic

Description Logics expresses the world concepts using elementary descriptions and complex descriptions. Elementary description includes *atomic concepts* and *atomic roles*. Complex descriptions can be derived by combining elementaries using *concept constructors*. In notation, A and B are used as atomic concepts, R is used as atomic role, and C and D are used as complex descriptions.

Different description languages are distinguised by the constructors they provide. In this thesis we use one in the family of $\mathcal{AL}$ -languages ($\mathcal{AL}$- Attributive language). The following table is the syntax rules for *the basic description language* $\mathcal{AL}$.

C, D	$\rightarrow$	A	(atomic concept)
		$\top$	(universal concept)
		$\bot$	(bottom concept)
		$\neg A$	(atomic negation)
		$C \sqcap D$	(intersection)
		$\forall R.C$	(value restriction)
		$\exists R.\top$	(limited existential quantification)

Note that the negation is only applicable for atomic concept. $\mathcal{AL}$ does not allow negation for complex description. And only the top concept is allowed in an existential quantification over a role.

The other languages in the family is extended from $\mathcal{AL}$ by adding more constructors. The more constructors added to the language, the more expressive and complex it is.

We can make an example of how to use $\mathcal{AL}$ to represent the real world. We suppose having atomic concepts: $Person, Male, Female$; and atomic roles *hasChild*. We can build more complex description from these atomic ones as following

$$Person \sqcap Male$$
$$Person \sqcap Female$$
$$Person \sqcap \forall hasChild.Male$$

2.3 Semantics of Description Logics

A formal semantic of the $\mathcal{AL}$ language is defined using an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where

- $\Delta^{\mathcal{I}}$ is a non-empty set, also called the domain of the interpretation.
- $\cdot^{\mathcal{I}}$ is a function that maps

– every concept to a subset of $\Delta^{\mathcal{I}}$.

– every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

For more complex concept descriptions, the semantic is interpretated as in following table

$$\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \emptyset \\
(\neg A)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash A^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} | \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \\
(\exists R.\top)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} | \exists b.(a,b) \in R^{\mathcal{I}}\}
\end{aligned}$$

2.4 The family of Description Logics

In here we consider the family of $\mathcal{AL}$ languages. The $\mathcal{AL}$ language presented in the previous section is the most basic language for the family. More expressive language is extended from $\mathcal{AL}$ language by adding more constructors.

The *union* constructor (denoted by the letter $\mathcal{U}$) is the union of concepts, written as $C \sqcup D$, and is interpreted as $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$.

Full existential quantification (indicated by the letter $\mathcal{E}$) is written as $\exists R.C$, and is interpreted as

$$(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} | \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$$

Note that different from $\exists R.\top$, in full existential quantification, arbitrary concepts are allowed to occur in the scope of the existential quantifier.

Number restrictions, which is denoted by the letter $\mathcal{N}$, have two different forms: $\geq nR$ (at least restriction) and $\leq nR$ (at most restriction) with n is a nonnegative integer. Number restrictions are interpretated as

$$(\geq nR)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} || \{b | (a,b) \in R^{\mathcal{I}}\}| \geq n\}$$

and

$$(\leq nR)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} || \{b | (a,b) \in R^{\mathcal{I}}\}| \leq n\}$$

where $|\cdot|$ is the cardinality of a set.

The *complex concept negation* introduces contructor allowing negation of concepts that are not atomic concept. Indicated by letter $\mathcal{C}$, written as $\neg C$, the negation is interpreted as

$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$$

Role hierarchy (indicated by letter $\mathcal{H}$) introduces role inclusions to the language. A role inclusion $R \subseteq S$ means for all interpretation $\mathcal{I}$, $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$.

Another role constructor is the *Inverse roles* (letter $\mathcal{I}$). Written as $R^{-}(a,b)$, inverse role is interpreted as

$$(R^{-})^{\mathcal{I}} = \{(a,b) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} | (b,a) \in R^{\mathcal{I}}\}$$

.

The expressive description language extending from $\mathcal{AL}$ is written by the string of the form

$$\mathcal{AL}[\mathcal{U}][\mathcal{E}][\mathcal{N}][\mathcal{C}][\mathcal{H}][\mathcal{I}]$$

where the presence of the letter in the name indicates the corresponding added constructor to the basic language $\mathcal{AL}$. For example, $\mathcal{ALUEN}$ is the extension of $\mathcal{AL}$ by union, full existential quantification and number restrictions.
However, from the semantic point of view, not all of the extended languages are distince. Negation can easily be used to replace the union and full existential quantification, and vice verse, as we have following transformation

$$\begin{gathered} C \sqcup D \equiv \neg(\neg C \sqcap \neg D) \\ \exists R.C \equiv \neg \forall R.\neg C \end{gathered}$$

Thus we can safely assume that a languages containing negation constructor also have union and full existential quantification constructors, and vice verse. For instance, $\mathcal{ALC}$ is also equivalent to $\mathcal{ALUEC}$.([5],[2])
The work in this thesis is based on $\mathcal{ALCHI}$ language, the extension from $\mathcal{AL}$ with union, full existential quantification, complex nagation, role hierarchy and inverse role.

2.5 Knowledge base

Terminology

In this section we want to introduce *terminology axioms*, which are used to represent the relations between concepts or roles. There are two kinds of terminological axioms: *inclusion* and *equality* , written as

$$C \sqsubseteq D \ \textit{inclusion}$$
$$C \equiv D \ (\text{equality})$$

An interpretation $\mathcal{I}$ satisfies an inclusion $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Similarly, $\mathcal{I}$ satisfies an equality $C \equiv D$ if $C^{\mathcal{I}} = D^{\mathcal{I}}$.

A *definition* is a special case of equality when the left side is an atomic concept. This can be illustrated by an example

$$Mother \equiv Woman \sqcap \exists hasChild.Man$$

which defines a $Mother$ to be a $Woman$ that has a child who is a Man.

An atomic concept C is called *directly uses* an atomic concept D if D is presented in the definition of C. The *uses* relation is defined as the transitive closure of the *directly uses*; which means C_1 *uses* C_i if there exists $\{C_1, C_2, .., C_i\}$ such that C_k *directly uses* C_{k+1}.

A *terminology* $\mathcal{T}$ (or a TBox) is composed of a finite set of terminological axioms. An interpretation $\mathcal{I}$ satisfies $\mathcal{T}$ if and only if it satisfies all the axioms in $\mathcal{T}$, and is called a model of $\mathcal{T}$.

$\mathcal{T}$ is *cyclic* iff there exists an atomic concept in $\mathcal{T}$ that *uses* itself. Otherwise, $\mathcal{T}$ is *acyclic*.

Assertions

The terminology axioms are used to describe world concepts, or classes of entity. On the other hand, an *assertion* is used to describe one entity or a relation between entities. For example, one concept assertion

$$Mother(Marry)$$

defines a single person *Marry* to be a *Mother*. And an assertion on role

$$hasChild(Tom, Peter)$$

says that *Tom* has a child who is *Peter*.
In generalization, assertions can be denoted by $C(a)$ or $R(a, b)$ with C, R are concept and role, respectively, and a, b are specific individuals.

An *world descriptions*, or *ABox* - denoted by $\mathcal{A}$, is composed of a finite set of assertions. Semantic for ABox is given by extending the interpretation of atomic concepts and roles in TBox for also individual names. That is, with an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, the interpretation function $\cdot^{\mathcal{I}}$ maps

- every concept to a subset of $\Delta^{\mathcal{I}}$.
- every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.
- every individual a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.

An interpretation $\mathcal{I}$ satisfies a concept assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, and satisfies a role assertion $R(a, b)$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$. $\mathcal{I}$ satisfies an ABox $\mathcal{A}$ if it satisfies every assertions in $\mathcal{A}$, and is called a model of $\mathcal{A}$.

Knowledge base

A knowledge base for description logics includes one TBox and one ABox.

$$\Sigma = \langle TBox, ABox \rangle$$

An interpretation $\mathcal{I}$ satisfies a knowledge base Σ if it satisfies the TBox and the ABox in Σ. In that case, $\mathcal{I}$ is called a model of Σ. A knowledge base Σ is *satisfiable* if it has a model.

2.6 Reasoning with Knowledge base

Reasoning problem

The inference problems on a knowledge base $\Sigma = \langle \mathcal{T}, \mathcal{A} \rangle$ consists of problems which can be defined as following

Definition 1. *Given a knownledge base* $\Sigma = \langle \mathcal{T}, \mathcal{A} \rangle$

- Σ *is called* consistent *if it has a model* $\mathcal{I}$.
- *A concept* C *is called* satisfiable w.r.t Σ *if there is a model* $\mathcal{I}$ *of* Σ *such that* $C^{\mathcal{I}} \neq \emptyset$. $\mathcal{I}$ *is called a model of* C *w.r.t* Σ.

- *The concept D* subsumes *the concept C w.r.t Σ (written as $\Sigma \vDash C \sqsubseteq D$) if for all model $\mathcal{I}$ of Σ, $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ (subsumption problem).*
- *The concept C is* equivalent *with the concept D w.r.t Σ if they* subsumes *each other w.r.t Σ.*
- *An individual a is* an instance of concept C w.r.t Σ *($\Sigma \vDash a : C$) if for all model $\mathcal{I}$ of Σ, $a^{\mathcal{I}} \in C^{\mathcal{I}}$ (instance checking problem).*
- *A pair of individuals (a, b) is* an instance of a role R w.r.t Σ *($\Sigma \vDash (a, b) : R$) if for all model $\mathcal{I}$ of Σ, $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$.*

Following we will consider the solutions for ABox consistency problem. All other problems, as we will see later, can be transformed into ABox consistency. The transformation for some problems will also be considered in this section.

Tableau calculus for ABox consistency

Firstly we introduce the tableau algorithm for ABox $\mathcal{A}$ consistency problem only, as for a knowledge base $\Sigma = \langle \mathcal{T}, \mathcal{A} \rangle$ with empty TBox $\mathcal{T}$. Before applying taleau algorithm, it is convinient to assume that all the concept expression are in *negation normal form* (NNF), i.e., that negation occurs only directly before concept name. Other arbitrary concepts can be transformed into NNF by pushing negation inward by using the de Morgan's rules and the duality of existential and universal restrictions, which is

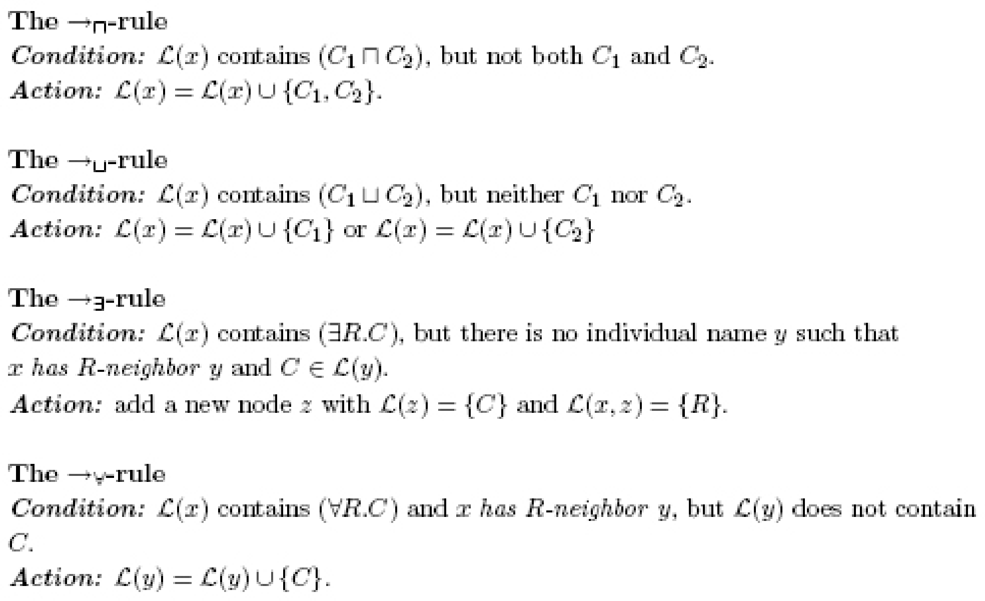
$$\neg\forall R.C \equiv \exists R.\neg C \text{ and } \neg\exists R.C \equiv \forall R.\neg C$$

The tableau algorithm tries to construct a finite representation of an interpretation $\mathcal{I}$ by working on *completion trees*. A completion tree is a tree with nodes are the individuals' name x, each of which are labeled with a set of concept expressions $\mathcal{L}(x)$; and edge between x and y, $\mathcal{L}(x, y)$ are roles connecting that two individuals. The starting completion tree is created directly from the ABox using its concept assertions and role assertions. The tableau calculus expands the trees using following semantics

- if $\mathcal{A} \vDash x : C_1 \sqcap C_2$ then $\mathcal{A} \vDash x : C_1$ and $\mathcal{A} \vDash x : C_2$.
- if $\mathcal{A} \vDash x : C_1 \sqcup C_2$ then $\mathcal{A} \vDash x : C_1$ or $\mathcal{A} \vDash x : C_2$.
- if $\mathcal{A} \vDash \forall R.C(a)$, then for every b such that $\mathcal{A} \vDash R(a, b)$, we have $\mathcal{A} \vDash C(b)$.

- if $\mathcal{A} \vDash \exists R.C(a)$, then there must be at least one individual b such that $\mathcal{A} \vDash R(a,b)$ and $\mathcal{A} \vDash C(b)$

From these semantics, we have the following expanding rules for $\mathcal{ALCHI}$ ABox consistency tableau agorithm, listed in Figure 2.1



The $\rightarrow_{\sqcap}$-rule
Condition: $\mathcal{L}(x)$ contains $(C_1 \sqcap C_2)$, but not both C_1 and C_2.
Action: $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$.

The $\rightarrow_{\sqcup}$-rule
Condition: $\mathcal{L}(x)$ contains $(C_1 \sqcup C_2)$, but neither C_1 nor C_2.
Action: $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1\}$ or $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_2\}$

The $\rightarrow_{\exists}$-rule
Condition: $\mathcal{L}(x)$ contains $(\exists R.C)$, but there is no individual name y such that x *has R-neighbor* y and $C \in \mathcal{L}(y)$.
Action: add a new node z with $\mathcal{L}(z) = \{C\}$ and $\mathcal{L}(x,z) = \{R\}$.

The $\rightarrow_{\forall}$-rule
Condition: $\mathcal{L}(x)$ contains $(\forall R.C)$ and x *has R-neighbor* y, but $\mathcal{L}(y)$ does not contain C.
Action: $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$.

Figure 2.1: Tableau transformation rules of the satisfiability algorithm for $\mathcal{ALCHI}$

Here a node x is called *has R-neighbor y* if $R \in \mathcal{L}(x,y)$ or $R^- \in \mathcal{L}(y,x)$. In case of satisfiability problem for the $\mathcal{ALCHI}$ knowledge base with non empty TBox, which is affected by role hierarchy, the definition of *has R-neighbor* needs to be extended as following: x is called *has R-neighbor y* if there exists a role S such that $S \in \mathcal{L}(x,y)$ or $S^- \in \mathcal{L}(y,x)$, and $S \sqsubseteq_\Sigma R$. The rule handling the disjunction is *nondetermistic* in the sense that the completion tree will be transformed into either one of the two possible options. The original completion tree is *open* if any of the two derived trees is *open*. A completion tree is *open* if it is *complete* and has no *clash*. The tree is *complete* where there is no more application for Tableau rules or it contains a *clash*. The tree contains a *clash* if there exists a node x and concept C such that $\{C, \neg C\} \in \mathcal{L}(x)$. If the tree is *open*, then the ABox is satisfiable.

Soundness. Tableau algorithm is sound because all the rules are deduced straight forward from the semantics of the constructors.

Termination. One important property of the tableau is that the concepts added to a label by applying any rules always has smaller size compared to the original concept. And since the original tree has finite size, the transformation procedure is terminated. More details on termination proof can be found in [7], [4].

Completeness. From a complete and clash-free tree we can build a model $\mathcal{I}$ for original ABox with

- The domain $\Delta^{\mathcal{I}}$ consists of all the nodes in the tree.
- For any concept name C, we have $\cdot^{\mathcal{I}} : C \rightarrow \{x|C \in \mathcal{L}(x)\}$.
- For any role name R, we have $\cdot^{\mathcal{I}} : R \rightarrow \{x, y|R \in \mathcal{L}(x, y)\}$.

Knowledge base consistency problem

Unlike the ABox consistency problem, in the knowledge base consistency problem we need to consider the presence of the terminology $\mathcal{T}$. For the case of regular terminology which contains only equality axioms, Tableau algorithm can be applied on the ABox after eliminating the TBox, which is done by extending TBox $\mathcal{T}$ into new TBox $\mathcal{T}'$, such that all the equivalent axioms in $\mathcal{T}'$ contains only base concept name (for more details please refer to [5]).
However, it is not as simple for the case of generalized terminology. The presense of inclusion axioms of the form $C \sqsubseteq D$ (C, D can be complex concept descriptions) makes the expanding no longer possible. In order to eliminate the TBox, another technique is introduced. First, the set of inclusion axioms $C_1 \sqsubseteq D_1, ..., C_n \sqsubseteq D_n$ in the terminology is transformed into a single equivalent axiom $\top \sqsubseteq \widehat{C}$ where

$$\widehat{C} = (\neg C_1 \sqcup D_1) \sqcap ... \sqcap (\neg C_n \sqcup D_n)$$

The single axiom $\top \sqsubseteq \widehat{C}$, with the $\top$ on the left side, simply means that every individual in the ABox is an instance of the concept $\widehat{C}$.
The knowledge base consistency problem now can be transform into ABox consistency problem by modifying the Tableau algorithm for ABox introduced above such that it takes this new axiom into account: the labels of every nodes in the ABox contain $\widehat{C}$ (which means every individuals is an instance of $\widehat{C}$), and the label for every new nodes created by applying $\exists$ rule also contains $\widehat{C}$.
However, the new Tableau algorithm needs not to be terminated. For example, considering consistency problem for a knowledge base with ABox

$\mathcal{A} = \{C(x_0)\}$ and TBox $\mathcal{T} = \{\top \sqsubseteq (\exists R.C)\}$, the tableau algorithm generates an infinite sequence of nodes for the completion tree $\{x_0, x_1, ...\}$ such that any node x_i has the label $\mathcal{L}(x_i) = \{C(x_i), \exists R.C(x_i)\}$. Here the algorithm runs into a cycle.
In order to make the algorithm terminated, we use the *dynamic blocking* technique to terminate those cycles. The application of $\rightarrow_\exists$ on node y is *blocked* by its ancestor x if $\mathcal{L}(x) = \mathcal{L}(y)$, whereas other Tableau rules are still applied on a blocked node. A blocking, however, can be broken if the condition is no more satisfied (i.e. changes to $\mathcal{L}(x)$, making $\mathcal{L}(x) \neq \mathcal{L}(y)$, which unblocks y). More details of blocking can be refered in [5], [13].
The key idea of blocking is that, whenever a cycle is detected, by checking whether the label of generated node is the same as the label of generating node, then the expansion from generated node (by $\rightarrow_\exists$ rule) is no longer allowed. However, with the presense of inverse roles, a newly generated node can affects the label of its ancester's label, thus breaks the blocking condition. In this case, the blocking need to be able to established broken and reestablished when needed.

Concept satisfiability problem

Given a knowledge base $\Sigma = \langle \mathcal{T}, \mathcal{A} \rangle$, we have from the definition that a concept C_0 is satisfiable w.r.t Σ if there is a model $\mathcal{I}$ of Σ such that $C_0^\mathcal{I} \neq \emptyset$. From a semantic view, we can see that ABox has no influence on the concept satisfiability problem in $\mathcal{ALCHI}$, and we can safely assume that the considered knowledge base has an empty ABox.
To transform this problem into ABox consistency, we consider new knowledge base $\Sigma' = \langle \mathcal{T}, \mathcal{A}_0 \rangle$ with $\mathcal{A}_0 = \{C(x_0)\}$, with x_0 is some newly created individual name. We have that C is satisfiable if Σ' is satisfiable, i.e. has a model. And since the satisfiability problem for a knowledge base, as solved above, is equivalent to the satisfiability problem of an ABox, then the problem of concept satisfiable for C_0 is also equivalent to the ABox consistency problem as well.

Concept subsumption problem

Given knowledge base Σ, a concept C is subsumed by D if $C^\mathcal{I} \subseteq D^\mathcal{I}$ for all model $\mathcal{I}$ of the knowledge base. This is equivalent to there is no model $\mathcal{I}$ of Σ that satisfies $\neg(C \sqsubseteq D)$; and since we have

$$\neg(C \sqsubseteq D) \equiv C \sqcap \neg D$$

we have the concept subsumption problem turns into the concept satisfiability problem for concept expression $C \sqcap \neg D$ w.r.t knowledge base Σ, which is, as just been discussed, can be transfomed into ABox consistency problem.

Instance checking problem

The instance checking problem for checking concept assertion $C(a)$ w.r.t the knowledge base Σ checks that if for all models $\mathcal{I}$ of Σ we have $a^{\mathcal{I}} \in C^{\mathcal{I}}$. This is also equivalent to there is no model $\mathcal{I}$ of Σ that satisfies $\neg C(a)$, or $\Sigma' = \langle \mathcal{T}, \mathcal{A}' \rangle$ is inconsistent, where $\mathcal{A}' = \mathcal{A} \cup \{\neg C(a)\}$. The instance checking problem is then turned into a knowledge base consistency problem.

Chapter 3

Partitioning of Abox in DL Knowledge Base

3.1 ABox partitioning

3.1.1 Introduction

As we already discussed in previous chapter, one of the most basic reasoning problem w.r.t $\mathcal{ALCHI}$ ABox is the instance checking. Lately, the Semantic Web are getting more and more of interest and the size of assertionology is greatly increased; and as the standard in-memory reasoning algorithm based on loading all the ABox into the memory, as well as the completion trees used in reasoning, the size of physical memory will soon be inefficient. There are several works on partitioning/modularizing (e.g. [11]) or on sumarization techniques ([9]). In this work we will continues and extend the approach of ABox partitioning based on $\forall$-*info structure* of the terminology that are proposed in [15].
The idea of the approach is that, considering the problem of checking if individual a is an instance of concept C w.r.t a knowledge base $\Sigma = \langle \mathcal{T}, \mathcal{A} \rangle$, usually only a small subset of ABox $\mathcal{A}$ is needed to do the reasoning (i.e. Figure 3.1). We only need to load this small subset into the memory instead of the whole ABox for more efficient usage of physical memory. Thus, the approach try to extract that small subset from the assertionology, and even more, divide ABox into smaller inter-disconnected partitions which can be use for executing the instance checking problem.

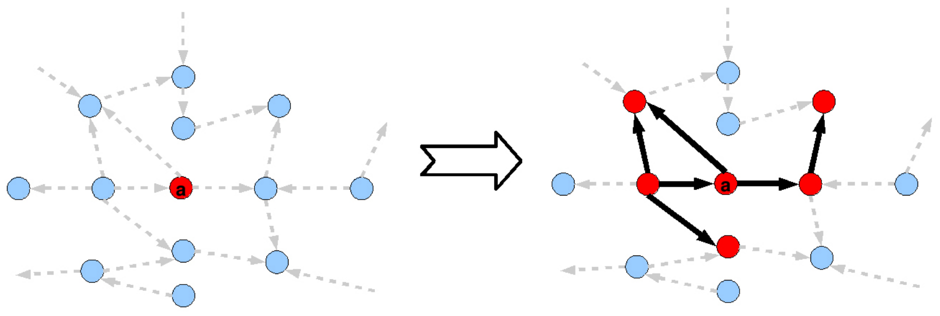


Figure 3.1: Irrelevent subset need for inferences on individual a

3.1.2 Forall-info structure of terminology

Considering reasoning problem for instance checking on individual a, and from the Tableau algorithm discussed in previous chapter we can see that the expanding process only propagates to new node when there is existential ($\exists$) or universal ($\forall$) restrictions; and to expands to an existed node in the ABox, the universal restriction is needed.
Assuming the terminology does not trigger any application for $\rightarrow_\forall$ rule when applying Tableau algorithm, and since our ABox can only be composed of the assertion of the type $C(a)$ with C is an atomic concept or negation of atomic concept, there would be no application for $\forall$-rule in Tableau application. That means the label for one node is only deduced from itself, but not from any other node; thus the instance checking problem for atomic concept name on individual a will need only node a to get the answer.
Now if we consider TBox containing a universal restriction, i.e. $\top \sqsubseteq \forall R.\neg C$, and ABox contains $R(b,a)$. If we carry out the instance checking for a on concept C, then applying Tableau rules for $\forall$ on individual b gives us:

$$(\forall R.\neg C(b), R(b,a)) \rightarrow \neg C(a)$$

This causes a clash $(C, \neg C)$ on node a. Thus, the presense of the universal restriction makes that a and b are both needed for the instance checking problem, or the connection between a and b is *inseparable*, and by intuition we see that a and b should be on the same *partition*.
From above example, we realize that the info related to universal restriction extracted from the terminology is needed to carry out the partitioning. To make the extracting convinient, we assume that all the axioms in the terminology are in *Shalow Normal Form (SNF)*.

Definition 2. *A concept description C is in* Shallow Normal Form (SNF), *if it has the shape $C = C_1 \sqcup C_2 \sqcup ... \sqcup C_n$, s.t. each C_i is either*

- *an atomic concept,*
- *a negated atomic concept,*
- *an $\exists$-constraint $\exists R.D$, s.t. D is an arbitrary concept description in negation normal form,*
- *a $\forall$-constraint $\forall R.D$, s.t. D is an arbitrary concept description in negation normal form.*

Lemma 3.1.2.1. *Each generalized concept inclusion $C \sqsubseteq D$ can be converted into a set S of equivalent concept description in SNF. Equivalent means that $C \sqsubseteq D$ is unsatisfiable iff the conjunction of the formulas in S is unsatisfiable.([15])*

With all the axioms in TBox being in SNF form, we have the following definition for $\forall$-info structure.

Definition 3. *A* $\forall$-info structure *for TBox $\mathcal{T}$ is a function $f_{\mathcal{T}}^{\forall} : N_R \rightarrow \mathcal{P}(N_C \cup \{\neg A | A \in N_C\} \cup \{\bot\} \cup \{*\})$, s.t., $N_C(N_R)$ is the set of concept (role) used in $\mathcal{T}$.The function $f_{\mathcal{T}}^{\forall}$ is used to manage the $\forall$-constraints, i.e. the function assigns to each role name in N_R one of the following entries:*

- *$\emptyset$ if we know that there is no $\forall$ constraint for R in $\mathcal{T}$.*
- *a subset S of $N_C \cup \{\neg A | A \in N_C\} \cup \{\bot\}$, s.t. there is no other concept but those in S, which occurs $\forall$-bound (i.e. they are subconcepts of a $\forall$) constraint) on R in $\mathcal{T}$.*
- *$*$, if there are arbitrary conplex $\forall$ constraints on role R in $\mathcal{T}$.*

([15])

The $f_{\mathcal{T}}^{\forall}$ function simply extracts the direct sub concept under every $\forall$ constraints in the $\mathcal{T}$, for example if we have following axiom in $\mathcal{T}$

$$FatherWithOnlySon \sqsubseteq \forall hasChild.Man$$

which has SNF form

$$\neg FatherWithOnlySon \sqcup \forall hasChild.Man$$

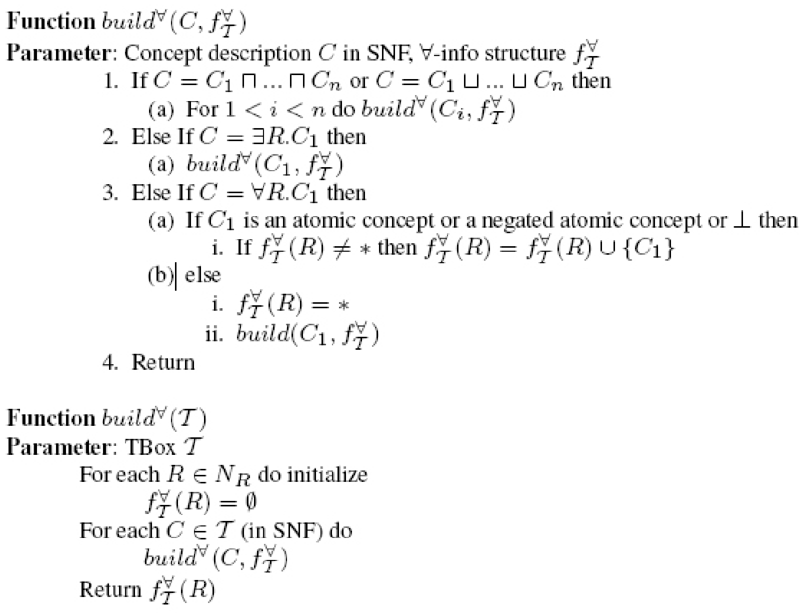
Then we have

$$f_{\mathcal{T}}^{\forall}(hasChild) = \{Man\}$$

while in case of following axiom

$\neg FatherWithGraduatedSon \sqcup \forall hasChild.(\exists graduatedFrom.University)$

then $f_{\mathcal{T}}^{\forall}(hasChild) = \{*\}$ because $\exists graduatedFrom.University$ is a complex concept description.
If $\mathcal{T}$ consists of both above axioms, then $f_{\mathcal{T}}^{\forall}(hasChild) = \{*\}$. The algorithm for calculating the $\forall$-info structure is illustrated in Figure 3.2.



```
Function $build^{\forall}(C, f_{\mathcal{T}}^{\forall})$
Parameter: Concept description $C$ in SNF, $\forall$-info structure $f_{\mathcal{T}}^{\forall}$
        1. If $C = C_1 \sqcap ... \sqcap C_n$ or $C = C_1 \sqcup ... \sqcup C_n$ then
            (a) For $1 < i < n$ do $build^{\forall}(C_i, f_{\mathcal{T}}^{\forall})$
        2. Else If $C = \exists R.C_1$ then
            (a) $build^{\forall}(C_1, f_{\mathcal{T}}^{\forall})$
        3. Else If $C = \forall R.C_1$ then
            (a) If $C_1$ is an atomic concept or a negated atomic concept or $\bot$ then
                i. If $f_{\mathcal{T}}^{\forall}(R) \neq *$ then $f_{\mathcal{T}}^{\forall}(R) = f_{\mathcal{T}}^{\forall}(R) \cup \{C_1\}$
            (b)| else
                i. $f_{\mathcal{T}}^{\forall}(R) = *$
                ii. $build(C_1, f_{\mathcal{T}}^{\forall})$
        4. Return

Function $build^{\forall}(\mathcal{T})$
Parameter: TBox $\mathcal{T}$
      For each $R \in N_R$ do initialize
          $f_{\mathcal{T}}^{\forall}(R) = \emptyset$
      For each $C \in \mathcal{T}$ (in SNF) do
          $build^{\forall}(C, f_{\mathcal{T}}^{\forall})$
      Return $f_{\mathcal{T}}^{\forall}(R)$
```

Figure 3.2: Calculate $\forall$-info structure

The definition of $\forall$-info structure is extended w.r.t ontology $\mathcal{O}$ as following

Definition 4. *A* $\forall$-info structure *for ontology* $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ *is a function* $f_{\mathcal{O}}^{\forall} : N_R \rightarrow \mathcal{P}(N_C \cup \{\neg A | A \in N_C\} \cup \{\bot\} \cup \{*\})$, *s.t.*

$$f_{\mathcal{O}}^{\forall}(R) = \begin{cases} * & \exists S \in N_{R}.\ \sqsubseteq_{\mathcal{R}} (R, S) \wedge (f_{\mathcal{O}}^{\forall}(S) = *) \\ \bigcup_{R \sqsubseteq_{\mathcal{R}} S} f_{\mathcal{T}}^{\forall}(S) & else \end{cases}$$

3.1.3 Partitioning of Abox

In order to partition the ABox, we need to evaluate the *importance* of a role assertion. We also need to develop a method to split ABox at a role assertion. Lets say that the *importance* of a role assertion is its $\mathcal{O}$-*separability*, s.t.

Definition 5. *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, a role assertion $R(a,b)$ is called $\mathcal{O}$-separable, if we have $INC(\mathcal{O}) \Leftrightarrow INC(\langle \mathcal{T}, \mathcal{R}, \mathcal{A}_2 \rangle)$, where*

$$\mathcal{A}_2 = \mathcal{A} \backslash \{R(a,b)\} \cup \{R(a,i_1), R(i_2,b)\} \cup \{i_1 : C | b : C \in \mathcal{A}\} \cup \{i_2 : C | a : C \in \mathcal{A}\},$$

where i_1 and i_2 are fresh individual names.

This definition also proposes a method to split the Abox, such that if the role assertion is $\mathcal{O}$-separable, then the consistency of the ontology is preserved. To determine the $\mathcal{O}$-separability, we have following formal criterion on a role assertion $R(a,b)$, w.r.t. ontology $\mathcal{O}$

Lemma 3.1.3.1. *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and a role assertion $R(a,b) \in \mathcal{A}$, it holds that $R(a,b)$ is $\mathcal{O}$-separable if*

1. *For each $C \in f^{\forall}_{\mathcal{O}}(R)$*
   - *$C = \bot$ or*
   - *we can find a concept description $D \in \{E | b : E \in \mathcal{A}\}$, such that we have $D \sqsubseteq_{\mathcal{T}} C$,*
2. *For each $C \in f^{\forall}_{\mathcal{O}}(R^-)$*
   - *$C = \bot$ or*
   - *we can find a concept description $D \in \{E | a : E \in \mathcal{A}\}$, such that we have $D \sqsubseteq_{\mathcal{T}} C$.*

Proof for this lemma can be found in [15]. By applying the replacement for $\mathcal{O}$ separable roles as in above definition, we can create the partitioning of the ABox.

Definition 6. *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, let $RED(\mathcal{A})$ be the ABox computed from $\mathcal{A}$ by replacing each $\mathcal{O}$-separable role assertion $R(a,b)$ by $\{R(a,i_1), R(i_2,b)\} \cup \{i_1 : C | b : C \in \mathcal{A}\} \cup \{i_2 : C | a : C \in \mathcal{A}\}$, with i_1, i_2 are fresh individuals. An* interconnection based partitioning *for $\mathcal{A}$, denoted*

$P(\mathcal{A}) = \{\mathcal{A}_1, .., \mathcal{A}_n\}$, is built by role-connectedless, i.e. two individuals are in the same partition iff there exists an explicit role assertion between these two individuals in $RED(\mathcal{A})$.

Following is an example about partitioning an ontology. Considering ontology $\mathcal{O}_{EX}\langle\mathcal{T}_{EX}, \mathcal{R}_{EX}, \mathcal{A}_{EX}\rangle$, where

$$\begin{array}{rl}
\mathcal{T}_{EX} = & \{ \\
& Chair \equiv \exists headOf.Department \sqcap Person \\
& Professor \sqsubseteq Faculty \\
& Book \sqsubseteq Publication \\
& GraduatedStudent \sqsubseteq Student \\
& Student \equiv Person \sqcap \exists takesCourse.Course \\
& \top \sqsubseteq \forall teacherOf.Course \\
& \exists teacherOf.\top \sqsubseteq Faculty \\
& Faculty \sqsubseteq Person \\
& \top \sqsubseteq \forall publicationAuthor^{-}.(Book \sqcup ConferencePaper) \\
& \} \\
\mathcal{R}_{EX} = & \{headOf \sqsubseteq worksFor, worksFor \sqsubseteq memberOf, memberOf \doteq member^{-}\} \\
\mathcal{A}_{EX} = & \text{see Figure 3.3}
\end{array}$$

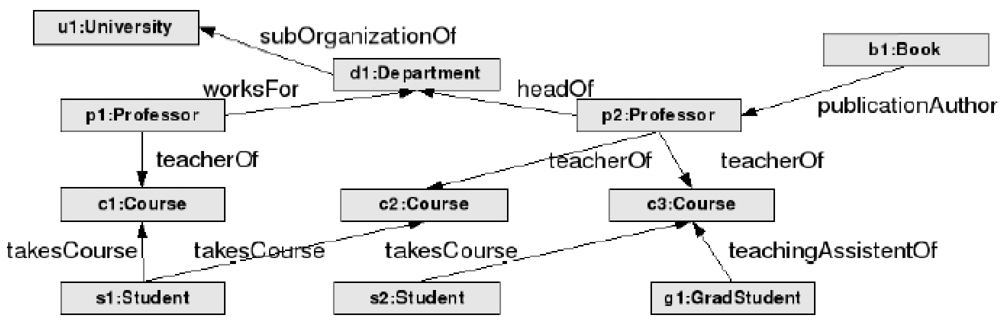


Figure 2. Guiding Example: ABox $\mathcal{A}_{EX}$ for ontology $\mathcal{O}_{EX}$

Figure 3.3: ABox $\mathcal{A}_{EX}$ of the example

And we have the partitioning of the Abox $\mathcal{A}_{EX}$ after applying the rules from the definitions is shown in Figure 3.4

Finally, we have the following lemma:

Lemma 3.1.3.2. *Denoting $P_a(\mathcal{A})$ the partition containing individual a in the set of partitioned ABox $P(\mathcal{A})$ of ABox $\mathcal{A}$, and given $CON(\langle\mathcal{T}, \mathcal{R}, \mathcal{A}\rangle)$, we have that $INC(\langle\mathcal{T}, \mathcal{R}, \mathcal{A} \cup \{a : C\}\rangle)$ iff $INC(\langle\mathcal{T}, \mathcal{R}, P_a(\mathcal{A}) \cup \{a : C\}\rangle)$ ([16])*

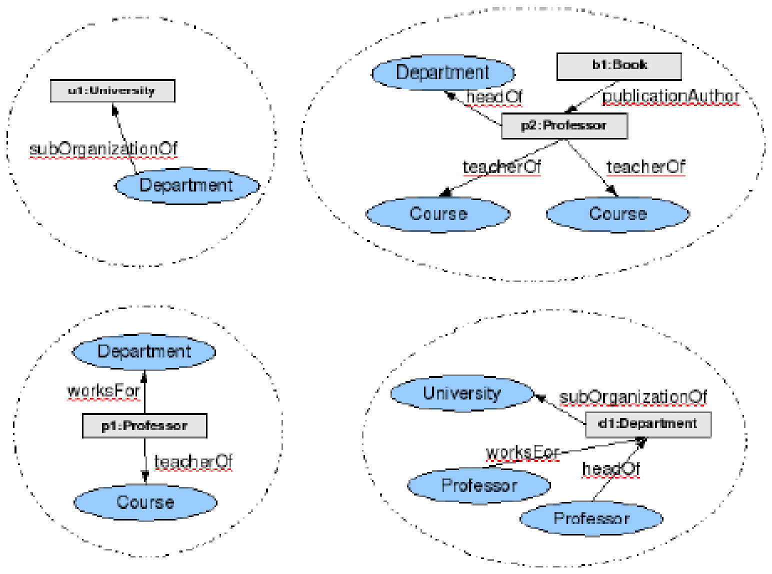


Figure 3.4: ABox $\mathcal{A}_{EX}$ of the example after partitioning

The instance checking problem w.r.t. ontology $\mathcal{O}$ now can be executed by loading $P_a(\mathcal{A})$ only, and not all ABox $\mathcal{A}$ as before. And from the experiment results, the size of $P_a(\mathcal{A})$ is significantly smaller than $\mathcal{A}$, especially when $\mathcal{A}$ is big.

3.2 Updating partitioned Knowledge base

The method for partitioning ABox proposed above has been proved to be effective using practical experimental data. However, one big obstacle needs to be overcome is the updating problem. As we already discussed, the ABox is partitioned based on the $\mathcal{O}$-separability of role assertions, and $\mathcal{O}$-separability is considered based on the $\forall$-info structure of the terminology as well as the concept assertions in the assertionology. Thus, updating an ontology $\mathcal{O}$ might cause one role assertion losing or gaining the $\mathcal{O}$-separability property, which leads to the changes in the partitioning. To solve this problem, in the following sections we will propose methods for updating our partitioned $\mathcal{ALCHI}$ ontology. In the first section we will discuss the method for updating ABox's concept and role assertion. In the latter section, the method for updating TBox axioms and RBox hierarchy will be mentioned.

3.2.1 Updating ABox

In the updating ABox, for the convinience we will adopt the *Syntactic ABox Updates* from [12].

Definition 7. *Let S be the set of assertions in an initial ABox $\mathcal{A}$. Then under* Syntactic Updates*, updating S with an ABox addition (respectively deletion) α, resulting in an updated set of ABox axioms S' such that $S' = S \cup \alpha$ (respectively $S' = S \backslash \alpha$).*

In the syntactic updates, there is no consistency cheking when adding a new assertion, as well as the enforcement of non-entailment when removing. However, systactic updates is computationally easier to handle. Further more, the consistency checking can be done before the updating being executed, if needed.

Before going into the details of updating ABox, we need to make some more refinement to the partitioning to make the updating comfortable because the current definition for partitioning is not really feasible for adding or removing of an assertion. The *updatable partitioning* will be built step by step throught the following definitions.

Definition 8. *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, an* ABox Partition *for $\mathcal{A}$ is a tuple $AP = \langle IN, S \rangle$ such that $IN \subseteq Inds(\mathcal{A})$ and*
$S = \{a : C | a \in M \wedge a : C \in \mathcal{A}\} \cup \{R(a,b) | (a \in IN \vee b \in IN) \wedge R(a,b) \in \mathcal{A}\}$,
where $M = \{a | b \in IN \wedge (R(a,b) \in \mathcal{A} \vee R(b,a) \in \mathcal{A})\} \cup IN$
And we define $\pi_{IN}(AP) = IN$, and $\pi_S(AP) = S$.

Informally speaking, an *ABox Partition* is composed of two components. The individuals set IN which contains the core individuals of the partition, and the assertion set S containing all the assertions needed in the partition. As depicted in the formula, if a is an individual in IN, then S contains all the assertions involving a and all the concept assertions involving all the direct neighbours of a.

Definition 9. *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, an* ABox Individual Partitioning *for $\mathcal{A}$ is a set $P = \{ap_1, .., ap_n\}$, such that each ap_i is an* ABox Partition *for $\mathcal{A}$ and*

1. *For each ap_i, ap_j, $(i \neq j)$ we have $\pi_{IN}(ap_i) \cap \pi_{IN}(ap_j) = \emptyset$*
2. *$Ind(\mathcal{A}) = \bigcup_{i=1..n} \pi_{IN}(ap_i)$*

3. $\mathcal{A} = \bigcup_{i=1..n} \pi_S(ap_i)$

The definition simply says that all the partitions has non-intersect core individual sets, the union of all the core individual sets of all the partitions is exactly the individual set of $\mathcal{A}$, and the union of all the assertion sets of all the partitions is the assertion set of $\mathcal{A}$.
Since each individual is assigned to only one ABox partition as core individual, we define a function $\phi_P : Ind(\mathcal{A}) \rightarrow P$ that return the partition for an given individual a. If $a \notin Ind(\mathcal{A})$ then $\phi_P(a) = \emptyset$.

We already had the partitioning for ABox. Now we will define the partitioning for the ontology. We also take into account the requirement for an instance checking problem to be able to execute on only one partition, which is our main motivation.

Definition 10. *Given a consistent ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, an* Ontology Partitioning *for $\mathcal{O}$ is a structure $OP_{\mathcal{O}} = \langle \mathcal{T}, \mathcal{R}, P \rangle$, where P is an ABox Partitioning for $\mathcal{A}$ such that for each individual $a \in Ind(\mathcal{A})$ and each atomic concept C we have $\mathcal{O} \vDash a : C$ iff $\langle \mathcal{T}, \mathcal{R}, \pi_S(\phi_P(a)) \rangle \vDash a : C$.*

To satisfy the requirement defined for Ontology partitioning (the instance checking being executed on only one partition), we use the $\mathcal{O}$-separability of role assertion to determine the partitioning of $\mathcal{A}$. From the previous section, it holds that with the partitioning an ABox based on the $\mathcal{O}$-separability of role assertions, the instance checking problem can be done with only one partition; thus applying it here also preserve that property.

Lemma 3.2.1.1. *Given ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, an ontology partitioning $OP_{\mathcal{O}} = \langle \mathcal{T}, \mathcal{R}, P \rangle$ and a role assertion $R(a, b) \in \mathcal{A}$, we have that $R(a, b)$ is $\mathcal{O}$-separable w.r.t $\mathcal{A}$ iff $R(a, b)$ is $\mathcal{O}$-separable w.r.t $\pi_S(\phi_P(a))$ (respectively $\pi_S(\phi_P(b))$)*

This lemma proposes another convinience for dynamically update an ABox partitioned by $\mathcal{O}$-separability of role assertions. When adding (removing) an assertion, we can decide if the change makes any role assertion losing or gaining $\mathcal{O}$-separability, causing the changes in partitioning, by testing the separability on only one partition, not with the whole ABox.

In the remaining part of this section, we will introduce means to preserve the partitioning of the Ontology under Syntatic ABox Update. We start from the begining with an empty Ontology (has no assertion in ABox)

and its corresponding partitioning, and then step by step build up the partitioned Ontology by using the two update functions for the Syntatic ABox Update, the *merge* function and the *reduce* function.
First we say that for the empty Ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \{\} \rangle$, the corresponding partitioning is $OP_{\mathcal{O}} = \langle \mathcal{T}, \mathcal{R}, P \rangle$ where $P = \{\langle \{\}, \{\} \rangle\}$ ([16])
We have the following formal definition for the two update functions:

Definition 11. *The result of the* merge operation *on a set of ABox Partitions for $\mathcal{A}$, $Merge(\{ap_1, .., ap_n\})$, is defined as the ABox Partition ap for $\mathcal{A}$, s.t.*

$$ap = \langle \bigcup_{i \leq n} \pi_{IN}(ap_i), \bigcup_{i \leq n} \pi_S(ap_i) \rangle$$

Definition 12. *The result of the* reduce operation *on an ABox Partition for $\mathcal{A}$, Reduce(pa), is defined as a set of ABox Partition $\{ap_1, .., ap_n\}$ built as follows:*

1. *For each $R(a, b) \in \pi_S(ap)$ do: if $R(a, b)$ is $\mathcal{O}$-separable, then replace $R(a, b)$ with $\{R(a, b*), R(a*, b)\} \cup \{a* : C | a : C \in \pi_S(ap)\} \cup \{b* : C | b : C \in \pi_S(ap)\}$, where $a*$ and $b*$ are fresh individual names for a and b.*
2. *Let $\{ap_1, .., ap_n\}$ be the disconnected partition in ap.*
3. *Replace each $a*$ in each ap_i by a.*
4. *Replace each $b*$ in each ap_i by b.*

The *merge* operation simply merge all the core individual sets and the assertion sets of all the partitions. The *reduce* operation, in the other hand, divides an ABox Partition into smaller partitions based on the $\mathcal{O}$-separability of role assertions.

We have the algorithm for updating ABox being illustrated in Figure 3.5, which can be informally expressed as following:
Adding a role assertion $R(a, b)$: first we ensure that partitions are existed for both a and b (if not, create new partition). If a and b are in the same partition, then the role assertion is just simply added to the partition. If a and b are in two different partitions, and $R(a, b)$ is not $\mathcal{O}$-separable, then the two partitions are merged.
Removing a role assertion $R(a, b)$: if a and b are in different partitions, then the role assertion is just simply removed from both partitions. If a and b in the same partition, then after removing the role assertion the partition

need to be rechecked to see if the removal of the role assertion causes the partition to be reduce-able.
Adding a concept assertion $C(a)$: first we ensure that partition is existed for individual a. Then we add concept assertion $C(a)$ to the partition of a ($\phi_P(a)$), and all the partitions that containing any role assertion for a, to maintain the data consistency between partitions.
Removing a concept assertion $C(a)$: remove the concept assertion from all the partitions containing it. After that all the role assertion involving a need to be $\mathcal{O}$-separability checked. If any of the role becomes inseparable due to the removal, then the corresponding partitions need to be merged.

```
Adding a role assertion $R(a,b)$:

1. If $\phi_P(a) = \emptyset$ then add $\langle \{a\}, \{R(a,b)\} \rangle$ to $P$
2. If $\phi_P(b) = \emptyset$ then add $\langle \{b\}, \{R(a,b)\} \rangle$ to $P$
3. If $\phi_P(a) = \phi_P(b)$ then $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \cup \{R(a,b)\}$
4. Else If $R(a,b)$ is $\mathcal{O}$-separable w.r.t. $\pi_S(\phi_P(a))$ then
   (a) Add $R(a,b)$ to $\pi_S(\phi_P(a))$ and to $\pi_S(\phi_P(b))$
   (b) Add $\{b : C \mid b : C \in \pi_S(\phi_P(b))\}$ to $\pi_S(\phi_P(a))$
   (c) Add $\{a : C \mid a : C \in \pi_S(\phi_P(a))\}$ to $\pi_S(\phi_P(b))$
5. Else
   (a) Add $R(a,b)$ to $\pi_S(\phi_P(a))$
   (b) $P = P \setminus \{\phi_P(a), \phi_P(b)\} \cup Merge(\phi_P(a), \phi_P(b))$
```

```
Removing a role assertion $R(a,b)$:

1. If $\phi_P(a) \neq \phi_P(b)$ then
   (a) $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \setminus \{R(a,b)\}$
   (b) $\pi_S(\phi_P(b)) = \pi_S(\phi_P(b)) \setminus \{R(a,b)\}$
2. Else
   (a) If $R(a,b)$ was $\mathcal{O}$-separable w.r.t. $\pi_S(\phi_P(a))$ then
         $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \setminus \{R(a,b)\}$
         $\pi_S(\phi_P(b)) = \pi_S(\phi_P(b)) \setminus \{R(a,b)\}$
   (b) Else $P = P \setminus \{\phi_P(a), \phi_P(b)\} \cup Reduce(Merge(\phi_P(a), \phi_P(b)))$
```

```
Adding a concept assertion $a : C$:

1. If $\phi_P(a) = \emptyset$ then add $\langle \{a\}, \{\} \rangle$ to $P$
2. $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \cup \{a : C\}$
3. For each $ap_i \in P$ do
     If $a \in Ind(\pi_S(ap_i))$ then $\pi_S(ap_i) = \pi_S(ap_i) \cup \{a : C\}$
4. $P = P \setminus \{\phi_P(a)\} \cup Reduce(\phi_P(a))$
```

```
Removing a concept assertion $a : C$:

1. $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \setminus \{a : C\}$
2. For each $ap_i \in P$ do
     – If $a \in Ind(\pi_S(ap_i))$ then $\pi_S(ap_i) = \pi_S(ap_i) \setminus \{a : C\}$
3. For each $R(a,b) \in \phi_P(a)$ do
     – If R(a,b) is not $\mathcal{O}$-separable, then $P = P \setminus \{\phi_P(a), \phi_P(b)\} \cup \{Merge(\phi_P(a), \phi_P(b))\}$
4. For each $R(b,a) \in \phi_P(a)$ do
     – If R(b,a) is not $\mathcal{O}$-separable, then $P = P \setminus \{\phi_P(b), \phi_P(a)\} \cup \{Merge(\phi_P(b), \phi_P(a))\}$
```

Figure 3.5: Updating ABox

3.2.2 Updating TBox and RBox

Unlike updating ABox, which might cause those partitions relating to one or two individuals to be merged/reduced, updating TBox or RBox is much more costly in term of computational resources: the adding/removing of a concept inclusion might triggers the changes on the whole partitioning. For example, an addition of a concept inclusion containing $\forall$ constraint for role R, might causes the merging/reducing of all the partition involving R, which, in practical, can involves a big part of the ABox.
Another reason which makes the updating TBox and RBox expensive is that it involves the instance retrieval problem (i.e. find all the role assertions for a given role expression R). Our algorithm for Ontology partitioning is based on the goal of optimizing the instance checking problem while minimize the data needs to be loaded into main memory. An instance retrival problem, in another hand, will require to be solved using all the partitions, consuming alot of computational power as well as the capacity of the main memory.

Before going into details of updating TBox, let us consider the concept taxonomy of the terminology. The taxonomy is critical for our algorithm, since it is used to determine the $\mathcal{O}$-separability of role assertions. Actually, computing the exact concept taxonomy from a terminology is as complex as a reasoning problem itself. Thus in our approach, instead of computing the exact concept taxonomy, we use an approximate concept taxonomy which is extracted directly from the axioms in TBox.

Definition 13. *A* Simple Concept Hierarchy H_S *of a TBox* $\mathcal{T}$ *is the subsume hierarchy of the concepts in* $\mathcal{T}$ *which can be explicitly deduced from* $SNF(\mathcal{T})$*. In other words, a subsume* $C \sqsubseteq D$ *exists in the hierarchy iff there exists* $\neg C \sqcup D$ *in* $SNF(\mathcal{T})$*.*

Simple Concept Hierarchy is a sound subtree of the complete concept hierarchy, and it is much easier to compute than the complete concept hierarchy of the TBox. Thus from now on we assume that the concept hierarchy used in this project is the Simple Concept Hierarchy; and all the inclusion relations of atomic concepts ($\sqsubseteq$ and $\sqsubset$) are based on the simple concept hierarchy.

We also extend the definition of the $\forall$-info structure, by introducing the *reduced* $\forall$-info structure and *extended* $\forall$-info structure.

Definition 14. *A* reduced $\forall$-*info structure for ontology $\mathcal{O}$ is a function $e^{\forall}_{\mathcal{O}}$ which is extend from $\forall$-info structure $f^{\forall}_{\mathcal{O}}$ such that for every role R:*

$$e^{\forall}_{\mathcal{O}}(R) = f^{\forall}_{\mathcal{O}}(R) \backslash \{C_k | \exists C \in f^{\forall}_{\mathcal{O}} : C \sqsubset C_k\}$$

Definition 15. *An* extended $\forall$-*info structure for ontology $\mathcal{O}$ is a function $g^{\forall}_{\mathcal{O}}$ which is extended from reduced $\forall$-info structure $e^{\forall}_{\mathcal{O}}$ as following:*

- *If $e^{\forall}_{\mathcal{O}}(R) = *$ then $g^{\forall}_{\mathcal{O}}(R) = \{\langle *, * \rangle\}$*
- *Else If $e^{\forall}_{\mathcal{O}}(R) = \emptyset$ then $g^{\forall}_{\mathcal{O}}(R) = \{\langle \emptyset, \emptyset \rangle\}$*
- *Else $g^{\forall}_{\mathcal{O}}(R) = \{\langle C_i, Sub(C_i) \rangle\}$, with $C_i \in e^{\forall}_{\mathcal{O}}(R)$, and $Sub(C_i)$ is the set of all the concepts that C_i subsumes in the simple concept hierarchy H_S.*

We also denote $\pi_C(g^{\forall}_{\mathcal{O}}(R)) \equiv \{C_i\}$, the set of all C_i appears in $\{\langle C_i, Sub(C_i) \rangle\}$ (which is $e^{\forall}_{\mathcal{O}}(R)$); and $\pi_{Sub,C_i}(g^{\forall}_{\mathcal{O}}(R)) \equiv Sub(C_i)$.

Informally speaking, the reduced $\forall$-info structure contains only the bottommost concepts of the concept hierarchy branches that appears in $f^{\forall}_{\mathcal{O}}$, w.r.t. the simple concept hierarchy. In the other hand, extended $\forall$-info structure is a set, each element of which is a tuples of a concept in $e^{\forall}_{\mathcal{O}}$ and the set of all the children of that concept, w.r.t. the concept hierarchy.

We have the following important property of the reduced and extended $\forall$-info structure, concerning $\mathcal{O}$-separability of role assertions.

Lemma 3.2.2.1. *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and a role assertion $R(a,b) \in \mathcal{A}$, it holds that $R(a,b)$ is $\mathcal{O}$-separable if*

1. *For each $C \in e^{\forall}_{\mathcal{O}}(R)$*
   - *$C = \bot$ or*
   - *we can find a concept description $D \in \{E | b : E \in \mathcal{A}\}$, such that we have $D \sqsubseteq C$,*
2. *For each $C \in e^{\forall}_{\mathcal{O}}(R^-)$*
   - *$C = \bot$ or*
   - *we can find a concept description $D \in \{E | a : E \in \mathcal{A}\}$, such that we have $D \sqsubseteq C$.*

Proof. This lemma is almost the same with the lemma 3.1.3.1 in previous section. The only differences are in the set of considered concepts. From the definition of reduced $\forall$-info structure, we have $e^{\forall}_{\mathcal{O}}(R) \subseteq f^{\forall}_{\mathcal{O}}(R)$. Thus, there are some concept being considered in lemma 3.1.3.1, but not here. However, we will prove that those concepts also satisfy the separability condition, if the separability condition is hold in lemma 3.2.2.1.
Consider C_i is any of those unconsidered concepts, we have $C_i \in f^{\forall}_{\mathcal{O}}(R)$ and $C_i \notin e^{\forall}_{\mathcal{O}}(R)$. From the definition of reduced $\forall$-info structure we have $\exists C \in e^{\forall}_{\mathcal{O}}(R) : C \sqsubset C_i$. Since the separability condition is hold in lemma 3.2.2.1, $\exists D \in \{|b : E \in \mathcal{A}\}, s.t. D \sqsubseteq C$; which also means $D \sqsubseteq C_i$, hence C_i satisfies the separability condition. □

Lemma 3.2.2.2. *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$ and a role assertion $R(a,b) \in \mathcal{A}$, it holds that $R(a,b)$ is $\mathcal{O}$-separable if*

1. *For each $C \in \pi_C(g^{\forall}_{\mathcal{O}}(R))$*
   - *$C = \bot$ or*
   - *we can find a concept description $D \in \{E|b : E \in \mathcal{A}\}$, such that we have $D \in \pi_{Sub,C}(g^{\forall}_{\mathcal{O}}(R))$,*
2. *For each $C \in \pi_C(g^{\forall}_{\mathcal{O}}(R^-))$*
   - *$C = \bot$ or*
   - *we can find a concept description $D \in \{E|a : E \in \mathcal{A}\}$, such that we have $D \in \pi_{Sub,C}(g^{\forall}_{\mathcal{O}}(R))$,*

Proof. This lemma can easily be proved using lemma 3.2.2.1 and the definition of extended $\forall$-info structure. We have $\pi_C(g^{\forall}_{\mathcal{O}}(R)) \equiv e^{\forall}_{\mathcal{O}}(R)$, thus $D \sqsubseteq C; C \in e^{\forall}_{\mathcal{O}}(R) \Leftrightarrow D \subseteq \pi_{Sub,C}(g^{\forall}_{\mathcal{O}}(R)); C \in \pi_C(g^{\forall}_{\mathcal{O}}(R))$. □

Now we come back to the problem of updating TBox and RBox. As we discussed in previous section, updating ABox assertions can lead to the merging/reducing involving one or two specific partitions identified by the individuals in the updated assertions, while updating in TBox and RBox rather causes the merging/reducing in many pairs of partitions involving a certain set of role names. More formally speaking, updating w.r.t TBox and RBox can affects a set of role U_R, such that for each $R \in U_R$, and all individual pairs $\{a,b\}, s.t. R(a,b) \in \mathcal{A}$, the status of the role assertion $R(a,b)$ might be changed (separable to inseparable or vice versa). We call these role set U_R the *changable role set*, and each $R \in U_R$ *changable role*

The following lemma proposes the relation between extend $\forall$-info structure and the changable role.

Lemma 3.2.2.3. *Given an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$. A role name R is* changable *w.r.t. an terminology update only if there exists* changes *in the extended $\forall$-info structure of R ($g^{\forall}_{\mathcal{O}}(R)$). An extend $\forall$-info structure for role R is called have changes if, after the update, $\pi_C(g^{\forall}_{\mathcal{O}}(R))$ changes (has new elements or lost elements), or for any C_i, $\pi_{Sub,C_i}(g^{\forall}_{\mathcal{O}}(R))$ changes (has new elements or lost elements).*

Proof. This lemma can be directly derived from Lemma 3.2.2.2. In the Lemma 3.2.2.2, the separability of a role assertion depends on the concept assertions in ABox and the extended $\forall$-info structure. Since we have no change in $g^{\forall}_{\mathcal{O}}(R)$, and also no change in ABox assertions as well, there is no change on the separability of role assertions on role R. □

From above lemma, we can have the following algorithm for updating TBox and RBox:

- For each role R in new terminology $\mathcal{T}*$, calculate $g^{\forall}_{\mathcal{O}}(R)$ before updating and $g^{\forall}_{\mathcal{O}*}(R)$ after updating.
  - If($g^{\forall}_{\mathcal{O}}(R) \neq g^{\forall}_{\mathcal{O}} * (R)$) then $U_R = U_R \cup R$
- For each $R \in U_R$, and for each $R(a, b)$:
  - If $R(a, b)$ is $\mathcal{O}$-separable but not $\mathcal{O}*$-separable then $P = P \backslash \{\phi_P(a), \phi_P(b)\} \cup Merge(\phi_P(a), \phi_P(b))$
  - If $R(a, b)$ is not $\mathcal{O}$-separable but $\mathcal{O}*$-separable then $P = P \backslash \phi_P(a) \cup Reduce(\phi_P(a))$

(*) $\mathcal{O}*$-separable is denoted for separable with respected to new Ontology (after update), while $\mathcal{O}$-separable is denoted for separable with respected to old Ontology.

Following we will consider specific cases of updating TBox, and the effects they make to the extended $\forall$-info structure.

Updating TBox - concept inclusions

Updating TBox by adding/removing a concept inclusion might causes changes to the $g^{\forall}_{\mathcal{O}}$ because

- if the concept inclusion adds $A \sqsubseteq B$ to the Simple Concept Hierachy H_S, and since the extended $\forall$-info structure $g^{\forall}_{\mathcal{O}}$ is built based on H_S, there probably have changes in $g^{\forall}_{\mathcal{O}}$.
- if the shallow negation form of the added concept inclusion contains one (or more) $\forall$-bound for a role R that doesnt existed in the old terminology (or does not exist in updated terminology in case of removing concept inclusion), then there is changes in the $\forall$-info structure of the terminology, which also probably causes changes in the extended $\forall$-info structure.

Thus, instead of recalculating the extend $\forall$-info structure, if we know that the update is of a concept inclusion, then we just need to extract the infomation from the added/removed concept inclusion itself to check if it will cause changes in the $g^{\forall}_{\mathcal{O}}$.
Before go into details how to decide the udpate role set from the added/ removed concept inclusion, we introduce some useful definitions.

Definition 16. *A $\forall$-info structure for a concept inclusion $C \sqsubseteq D$ w.r.t $\mathcal{O}$, written as $f^{\forall}_{C \sqsubseteq D, \mathcal{O}}$, is a function that assigns to each role name R in $SNF(C \sqsubseteq D)$ one of the following entries:*

- *$\emptyset$ if we know that there is no $\forall$ constraint for R in $SNF(C \sqsubseteq D)$.*
- *a set S of atomic concept or negation atomic concept, s.t. there is no other than those in S that occurs $\forall$-bound on R in $SNF(C \sqsubseteq D)$.*
- *$*$, if there are arbitrary complex $\forall$ constraints on role R in $SNF(C \sqsubseteq D)$.*

This definition is literally similar to the definition of the $\forall$-info structure stated in section 3.1.2, but for only one axiom. From this, we also define the *reduced $\forall$-info structure for a concept inclusion w.r.t. ontology $\mathcal{O}$* and *extended $\forall$-info structure for a concept inclusion w.r.t. ontology $\mathcal{O}$* in the same manner

Definition 17. *A* reduced *$\forall$-info structure for a concept inclusion $C \sqsubseteq D$ w.r.t. ontology $\mathcal{O}$ is a function $e^{\forall}_{C \sqsubseteq D, \mathcal{O}}$ which is extend from $\forall$-info structure $f^{\forall}_{C \sqsubseteq D, \mathcal{O}}$ such that for every role R:*

$$e^{\forall}_{C \sqsubseteq D, \mathcal{O}}(R) = f^{\forall}_{C \sqsubseteq D, \mathcal{O}}(R) \backslash \{C_k | \exists C \in f^{\forall}_{C \sqsubseteq D, \mathcal{O}} : C \sqsubset C_k\}$$

Definition 18. *An* extended *$\forall$-info structure for a concept inclusion $C \sqsubseteq D$ w.r.t. ontology $\mathcal{O}$ is a function $g^{\forall}_{C\sqsubseteq D,\mathcal{O}}$ which is extended from reduced $\forall$-info structure $e^{\forall}_{C\sqsubseteq D,\mathcal{O}}$ as following:*

- *If $e^{\forall}_{C\sqsubseteq D,\mathcal{O}}(R) = *$ then $g^{\forall}_{C\sqsubseteq D,\mathcal{O}}(R) = \{\langle *, * \rangle\}$*
- *Else If $e^{\forall}_{C\sqsubseteq D,\mathcal{O}}(R) = \emptyset$ then $g^{\forall}_{C\sqsubseteq D,\mathcal{O}}(R) = \{\langle \emptyset, \emptyset \rangle\}$*
- *Else $g^{\forall}_{C\sqsubseteq D,\mathcal{O}}(R) = \{\langle C_i, Sub(C_i) \rangle\}$, with $C_i \in e^{\forall}_{C\sqsubseteq D,\mathcal{O}}(R)$, and $Sub(C_i)$ is the set of all the concepts that C_i subsumes in the simple concept hierarchy H_S.*

And we have the following detailed algorithm for calculating the update role set in case of adding/removing a concept inclusion:

- Adding a concept inclusion $C \sqsubseteq D$
  - For each $A \sqsubseteq B$ that is added to the concept hierarchy:
    - for any role R that $B \in g^{\forall}_{\mathcal{O}}(R)$, $U_R = U_R \cup R$
  - For each R s.t. $g^{\forall}_{C\sqsubseteq D,\mathcal{O}*}(R) \neq \emptyset \wedge g^{\forall}_{C\sqsubseteq D,\mathcal{O}*}(R) \nsubseteq g^{\forall}_{\mathcal{O}}(R)$, $U_R = U_R \cup R$
- Removing a concept inclusion $C \sqsubseteq D$
  - For each $A \sqsubseteq B$ that is removed to the concept hierarchy:
    - for any role R that $B \in g^{\forall}_{\mathcal{O}}(R)$, $U_R = U_R \cup R$
  - For each R s.t. $g^{\forall}_{C\sqsubseteq D,\mathcal{O}*}(R) \neq \emptyset \wedge g^{\forall}_{C\sqsubseteq D,\mathcal{O}*}(R) \nsubseteq g^{\forall}_{\mathcal{O}*}(R)$, $U_R = U_R \cup R$

Here we denote $\mathcal{O}$ the ontology before updating and $\mathcal{O}*$ the ontology after updating.

Updating RBox - role inclusions

Adding/removing a role inclusion has a quite obvious effect: it might change the role hierarchy. Since the $\forall$-info structure of the Ontology is calculated using role taxonomy, this will make the $\forall$-info structure, and also the extended $\forall$-info structure, to change. Following is the details algorithm for determining the udpate role set

- Adding a role inclusion $R \sqsubseteq S$

  - if $g^{\forall}_{\mathcal{O}}(S) \not\subseteq g^{\forall}_{\mathcal{O}}(R)$ then for all sub role V of R ($V \sqsubseteq R$), $U_R = U_R \cup V$

- Removing a role inclusion $R \sqsubseteq S$
  - if $g^{\forall}_{\mathcal{O}}(S) \not\subseteq g^{\forall}_{\mathcal{O}*}(R)$ then for all sub role V of R ($V \sqsubseteq R$), $U_R = U_R \cup V$

Updating RBox - role inverses

Adding/removing a role inverse, on the other hand, might change the $\forall$-bound for both roles involving in the role inverse. This causes the changes for the $\forall$-info structure of the both roles, which also alters their extend $\forall$-info structure, thus we have following algorithm for calcualting update role set

- Adding a role inverse pair $R = Inv(S)$
  - for all role $V \sqsubseteq R$, $U_R = U_R \cup V$
  - for all role $W \sqsubseteq S$, $U_R = U_R \cup W$
- Removing a role inverse pair $R = Inv(S)$
  - for all role $V \sqsubseteq R$, $U_R = U_R \cup V$
  - for all role $W \sqsubseteq S$, $U_R = U_R \cup W$

3.3 Reasoning with partitioned Knowledge base

In this paper, we consider following basic reasoning problems w.r.t. Description Logics: the concept subsumption problem, and the instance checking problem for concept assertion and role assertion.
For concept subsumption problem, it can be done without being concerned about the ABox. We can do the reasoning on any partition.
For role assertion checking, i.e. checking of $R(a, b)$, the reasoning can be done on the partition containing either a or b. Since we dont allow complex role in $\mathcal{ALCHI}$, it is quite simple to execute this inference.
The biggest concern with reasoning w.r.t. partitioned knowledge base is the instance checking for concept assertion. With atomic concept assertion, $C(a)$, it is already shown that the checking can be done on the partition containing a alone, using normal tableau algorithm. However, we want to be able to perform the checking also in the case of C being some arbitrary

complex concept (i.e. we want to check $john : Man \sqcap \forall hasChild.Man$). For this kind of checking, we will need to merge several partitions in order to do the reasoning. However, in this thesis we will not propose a method for finding the exact minimal set of partitions needed for reasoning, but rather a set of partitions which are easy to determined and relevant for solving the problem.
Let say our instance checking problem is performed w.r.t. ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$, with the partitioning $OP_{\mathcal{O}} = \langle \mathcal{T}, \mathcal{R}, P \rangle$, where P is an ABox Partitioning for $\mathcal{A}$. We transform our problem into knowledge base consistency problem for $\mathcal{O}' = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \cup \{\neg C(a)\} \rangle$, and execute Tableau algorithm, starting at the partition of a: $\phi_P(a)$. Whenever the application of Tableau algorithm goes outside the current partition, we merge the other respective partition into current partition, and continue expanding the competion tree.

Lemma 3.3.0.4. *If during expanding the compeltion tree, there is a merging of two partitions on role assertion $R(x, y)$, then there exists $\forall$ constraint of R in $\neg C$.*

Proof. If there is merging at $R(x, y)$, meaning $R(x, y)$ changes from $\mathcal{O}$-separable to $\mathcal{O}$-inseparable when we add $\neg C(a)$ to the ABox, there must exists changes in the $\forall$-info structure of R. Since there is no change in the terminolgy, then the change is causes by the presence of $\neg C$. This means there must be $\forall$ constraint of R in $\neg C$. $\square$

We will use this result to determine which partitions need to be merged. We start with the partition of a, P_a, and merge any partition that are separated with P_a by a role assertion $R(x, y)$ with R has $\forall$ constraint in $\neg C$ and any pair $\{x, y\}$. The merging then is recursively executed, until there is no more partition needed to be merged. After this procedure, the resulted partition is the partition that is relevant for solving the given instance checking problem. Notice that because we exhaustedly merged all the relating partitions, the resulted partition is not the minimal solution.

Chapter 4

Distributed Storage System for DL Knowledge bases

4.1 Extending problem of Knowledge bases

In the previous chapter, the algorithm for partitioning ABox based on $\mathcal{O}$-separability of role assertions was already discussed in details. The main objectives of the algorithm is to avoid making main memory overloaded when working with relatively large knowledge base. The main result of the proposed algorithm is the dividing the assertionology into many small partitions such that only some of those are needed when solving a reasoning problem.
Further developing the idea, seeing the trend of increasing size for Description Logics Knowledge Base systems that can exceed the storage capablility of a single computer, and the possible demand for decentralizing the Knowledge Base system, we carried out a reaseach on developing a distributed system for storing the DL Knowledge Base data. The idea is rather simple: the small partitions created from partitioning ABox is grouped and distributed to storage nodes. Here we need a system to manage the distribution, and even more, to provide the capability of performing the Knowledge Base updating, introduced in previous chapter, distributedly.
We have developed a software system implementing above idea. In this chapter, the overall structure as well as the details implementation design of the system will be clearly illustrated.

4.2 Overall system components

Our system consists of a server and a set of nodes, as illustrated in Figure 4.1. The main task of the server is to manage all the nodes, distribute

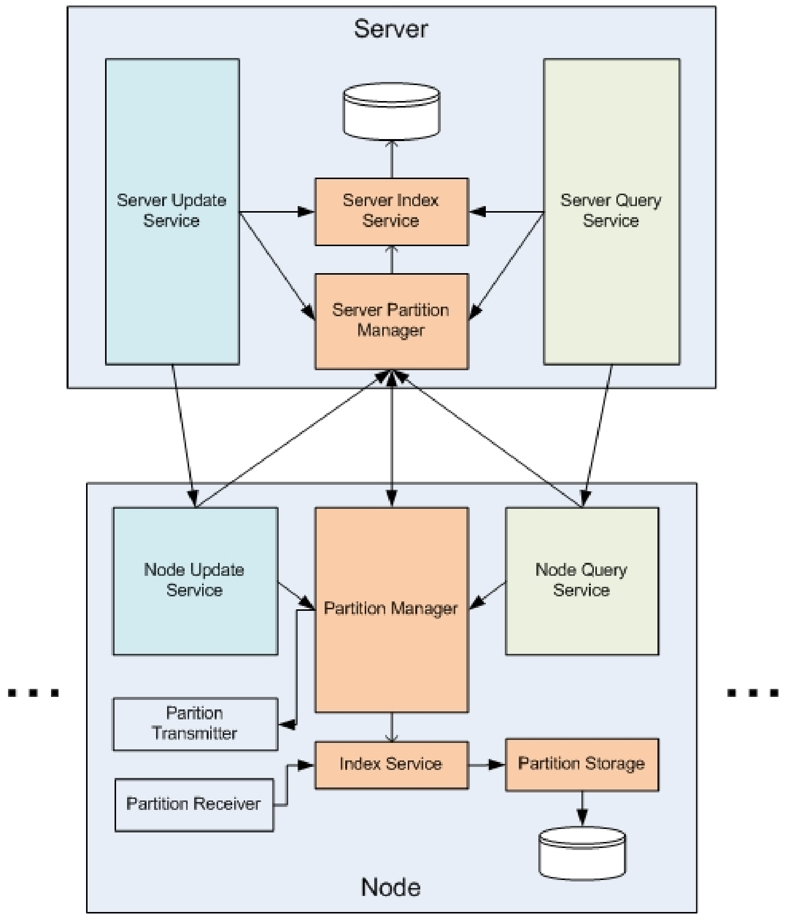


Figure 4.1: System architecture

the partitions on nodes, as well as receiving and scheduling the works requested from user. In the other hand, each node manages its own set of partitions using a database management system, and provides certain services to server. Since we only partition the ABox, and the size of TBox is much smaller compared to Abox, every partition contains a full version of the terminology. The terminology is also hard stored on the server, so that any request for TBox reasoning can be solved on server without requesting nodes' service.

4.2.1 The Server

Server partition manager

Server partition manager component on the server provides all the utility methods involving the partitions on the nodes. Some of the basic functions it provides are moving partition from node to node, adding a partition to a node, find the partition containing a given individual, etc. Server partition manager also provides methods for working directly on assertions, such as deleting or adding an assertion to a partition. Normally, these methods on assertions simply add/delete the assertion to the corresponding partition on the correct node without checking for repartitioning, assuming that the checking is done beforehand and these assertions cause no repartitioning of the ABox.

Server index service

Server index service monitors the id mapping as well as manages the terminology on the server. The id mapping is the mapping between the full uri naming of individuals, concept names and role names with an unique identifier. This mapping is stored in the database on the server, and is loaded by server index service component everytime the server is up and running. The terminology, including concept inclusions, role inclusions (role hierarchy), and role inverses is also stored in the database on the server. Server index service loads the terminology from database when the server is started, after loading the idmap so that all the concepts and roles can be indentified.
Another important function of the server index service is to provide the mapping between core indidividuals and nodes. This allow users or other services to be able to indentify the node that contains the partition having a specific individual as core individual. This mapping is not hard stored on the server, but is initialized when the system starts. Whenever a node is up, it connects to the server and send its set of individuals. The mapping on the server is built from those set and is stored in the main memory.

Server update service

Server update service component is to provide users the functionalities of updating the knowledge base. Upon given an update request, this component execute certain parts of the request, using the support from Server partition manager and server index service. For the parts which need the

information from specific nodes, it invokes the corresponding node update service to do the work.
For each type of updating, the parts that are done on server and the parts that need to be done on node will be discussed in more details in later sections.

Server query service

This component provides users the query service. If the query concerns only terminology reasoning, then it will be executed only on the server. Other than that, the node query service will be invoked as needed.

4.2.2 The Node

Partition manager

The partition manager on each node manages all the operations on partitions, such as adding, deleting, etc. It also supports two important methods on partitions: merging and reducing. Unlike reducing which can be performed solely on a node, the merging might need partitions from other nodes, in that case the request for moving partition will be sent to the server partiton manager.

Index service and partition storage

These components are the parts of the node that connect directly to the database. The partition storage manages all the partitions in the database and allows other component to retrieve the partition given the partition's id. One important property of the partition storage is that it maintains the number of the partitions loaded into main memory (as a cache) to speed up the accessing to those partitions, while also prevent the memory from being overloaded.
The index service component, in the other hand, mapping each individual to its partition. It is also the only component that can directly access to the partition storage. All the requests from other components to partition storage have to go through the index service.

Partition transmitter and receiver

As the name implied, a partition will be transfered to other node by partition transmitter component if there is a request from partition manager component. The partition arrives at the destination's partition receiver and is added to the partition storage via index service afterward.

Node update service

The node update service do its part of updating which are passed down by server update service. The node update service component uses functionalities provided by partition manager and index service when needed.

Node query service

The node query service do its part of querying which are passed down by server query service. The node query service component uses functionalities provided by partition manager and index service when needed.

4.3 System Implementation Design

Our system is implemented based on an on-developing description logics representation infrastructure, adding more functionalities, and performing decentralization using the system architecture mentioned above. The details about implementation will be described in following sections.

4.3.1 Description Logics representation

To represent the description logics individuals, concepts or axioms, a Java based infrastructure was already implemented. The classes diagram for this system can be found in Appendix A.
Besides that, an implementation for centralized partitioning ABox based on ∀-info structure is under development, and is already tested with LUBM data, as depicted in [16].

4.3.2 Network Implementation using RMI

Network communication is always the first concern when developing distributed system. In our system, we decide to use RMI - Remote Method

Invocation, a java based simple remote interface between different JVMs. RMI, at the most basic level, is Java's remote procedure call (RPC) mechanism, however it has many advantages over traditional RPC, some of which are:

- Object Oriented: Besides predefined data types, RMI can also pass object as the parameter or return value to the remote method directly without needs of extra code in server and client to marshal/unmarshal data.

- Dynamic Code Loading: RMI has capability to download the definition of an object's class if the class is not defined in the receiver's JVM. That means all the types and behaviors of an object, previously only available for a single JVM, can be transmitted to another, possibly remote, JVM.

- Security: Using java built-in security mechanism, RMI guarantees the security of the server when users load class implememtation. The rights applied for all the connection are defined in the policy file of the server, and managed by security management.

- Thread based: The RMI remote interface is totally thread-based, with the utilization of thread pool. This allow better processing time without wasting resources.

- Distributed Garbage Collection: RMI has a distributed garbage collection that free the objects that has no reference from remote JVM.

RMI distributed application

A RMI application is often composed of two components: a server and a client. Basically, the server creates objects implementing the services it offers, and makes the objects accessible by the client. The client, when requestting for services, obtains a reference to one of those objects and invokes its methods.
The Figure 4.2 illustrated the scenario of a RMI application. The server call the RMI Registry to associate (bind) a name with a service object. The client later obtains the service object by looking up by its name in the server's RMI registry, and invokes the methods in the object. In the figure, a webserver is used to stored the class definition to be loaded by client and server. In real system, this webserver can be any place accessible by both

client and server.

In the RMI application, an object that has methods to be invoked re-

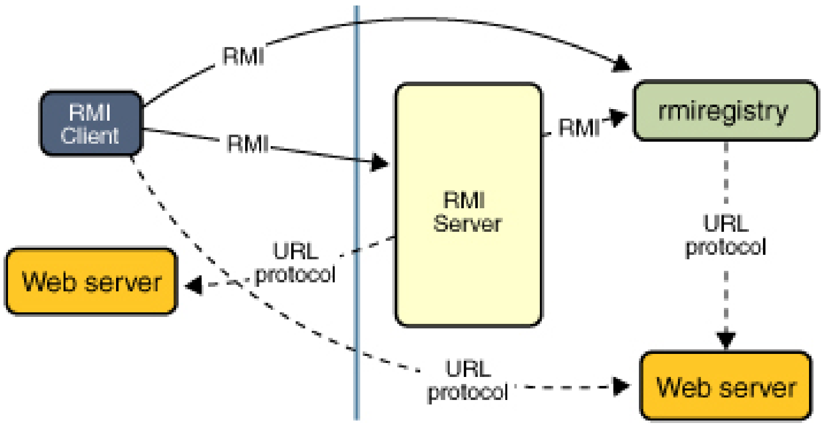


Figure 4.2: RMI distributed application

motely is a *remote object*, and the methods are called *remote methods*. An object becomes remote object if it implements a *remote interface*, which has following characteristics:

- A remote interface is a sub interface of `java.rmi.Remote`.
- Every remote methods in the interface must declare `java.rmi.RemoteException` in its throws clause.

The remote method is, when invoked, not performed on client but on the server. The invocation is passed from client to server using skeleton-stub mechanism.

A remote object, when being deployed, is compiled by RMI and result in a skeleton-stub objects pair. When the client requests for the service, it receives the stub. The stub acts as the local representative, or proxy, for the remote object. It implements the same remote interfaces that the remote object implements, yet not the real implementation but forwarding everything to skeleton to be perfomed on the server.

RMI Security policy

The security of the remote access on the server is managed using a Security Manager. There are several ways to specify the security policy, one of which is to define them in some policy files, and load them upon running application.

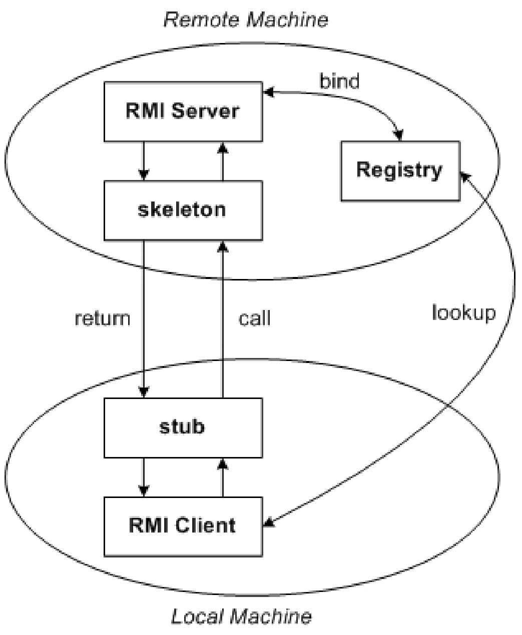


Figure 4.3: RMI skeleton-stub architecture

4.3.3 Multi thread based

Our system aims to be not only an extendable distributed Description Logics knowledge base storage system, but also a services provider that offers multiple updating, multiple retriving and also reasoning services. With the scalability in mind, we have designed the system totally thread based so that it can serve multiple requests. Also, the whole system architecture allows the number of data storage nodes to be extended easily.

Every remote service in our system has the structure as illustrated in Figure 4.4. To manage the threads, we use a so called *thread pool*, with a fixed number of working threads. These working threads get the task (`Runnable` object) from a *queue*. The main service thread puts a new task in the queue when receiving a request from client, creates a new handler for the task, and returns the handler to client. The handler, which is also a remote object, provides methods allowing users to wait until the work finishs, retrieve the result, etc., remotely.

The thread pool is created and managed by the main service thread. It maintains a list of working threads to perform the work. Each thread takes a new task from the queue when finishing previous task. When there is no task in the queue, the threads go into sleep, and are waken up when new tasks go in the queue.

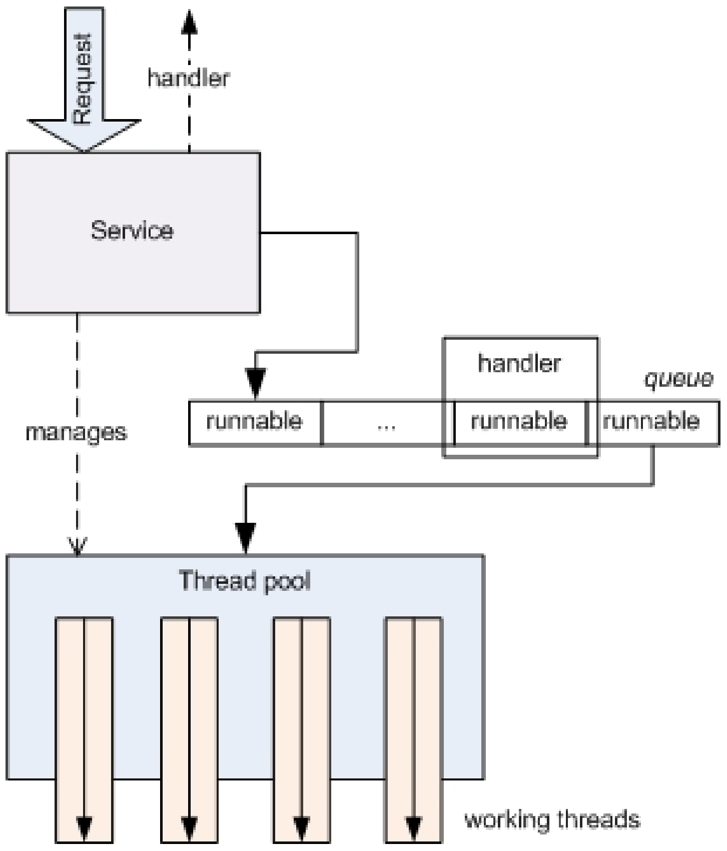


Figure 4.4: Structure of remote service

Refering to Figure 4.1, the components that implement this structure are:

- Server Update Service
- Server Query Service
- Server Partition Manager
- Node Update Service
- Node Query Service
- Partition Manager

4.3.4 Plugable implementation for services

During the development of the system, we saw the possibility of existing many different algorithms for updating the assertionology as well as terminology, and even for partitioning of knowledge base. From that, we decided to design the application so that it is easy to extend the application with adding new algorithms. The simplified design is depicted in the figure 4.5. We defined one interface and one abstract class, which are to be imple-

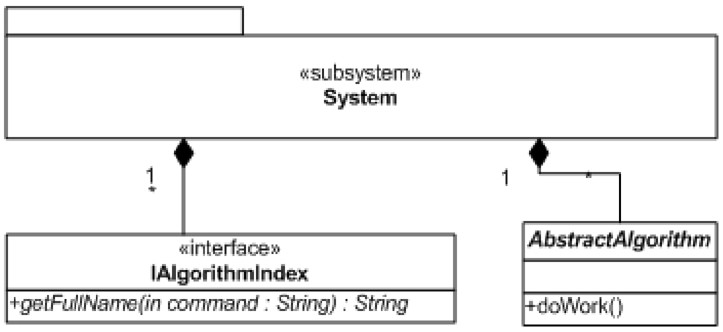


Figure 4.5: Plugable design for algorithms

mented by the implementation of the new algorithm. The implementation should include:

- One index class implementing the `IAlgorithmIndex` interface, and return the full qualified name for the service class of the algorithm, given the command input from users or external system.
- A set of algorithm classes extending `AbstractAlgorithm`. Each of these class is corresponding to a command which is given as the input to the index class to get full qualified name. This full qualified name is then used to create an instance of the algorithm class. Each algorithm class implements its services in the `doWork` method, which will be called by the system.

We can make a simple example here to illustrate the implemetations for the index interface and the algorithm class. Consider a developer want to create his own set of algorithm for updating, then he has to create his index class, named `MyIndex`, and a set of update algorithm class, one of them named `MyAddCA`, both are in the package `myalgorithm`. The class `MyIndex` needs to define a set of command for all the algorithm classes, i.e. "Add concept assertion" for the algorithm class `MyAddCA`, and implement the method `getFullName` to return the full qualified name of the algorithm class, given the corresponding command. This means the method invocation `getFullName("Add␣concept␣assertion")` will return the string `myalgorithm.MyAddCA`. The system can use this string to create the algorithm instance, and invokes the `doWork` method.

This design allows developer to add new algorithm by simply implementing his index class and algorithm classes, then specify the full qualified class name of the index class to the system. The system does not need to be shut down to perform the changes.

4.3.5 Database

The database of the system is divided into two main parts: one is on the server, storing the terminology of the knowledge base and the id mapping, and the other part is distributed among nodes, containing data about assertionology.

Database for Server

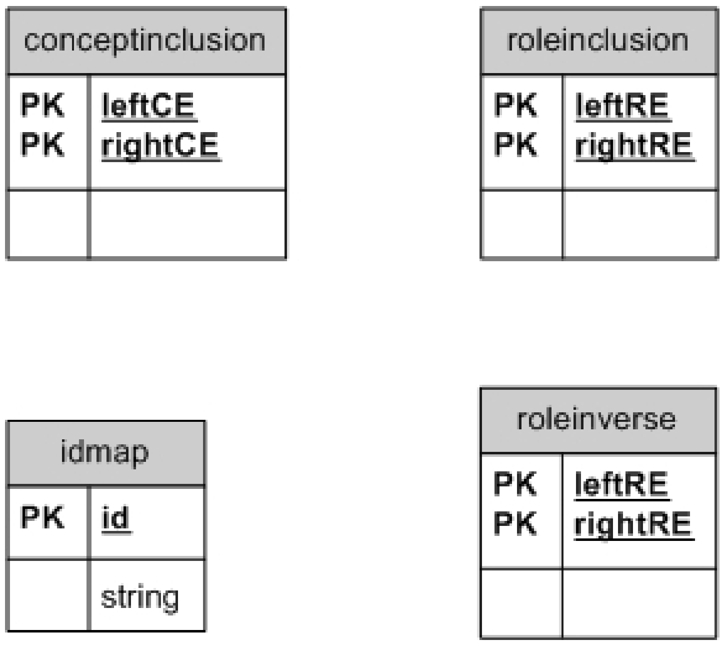


Figure 4.6: Server database

The database designed for the server is relatively simple. The most complex part in the terminology are the concept inclusions, which are possibly composed of a large combination of concept expressions. However, real implementation for that combination in the database is really complicated and the retrieval of an inclusion might need to join many tables, which is costly. To avoid that, instead of designing a database model for conjunction, disjunction, etc., we decide to store the whole complex concept description as a single preformatted string so that it can be parsed into a complete conplex description. For example, if we have a concept inclusion

$$Father \sqsubseteq \forall hasParent.(Man \sqcup Woman) \sqcap Man$$

we can store it in the database as a string `Father` in the *leftCE* table field and `FORALL:hasParent:(:Man:OR:Woman:):AND:Man` in the *rightCE* table field. The stored strings are loaded by the server when needed, are validated and intergrity checked before being ready to be used.

The table `idmap` is used to map between an id number and a name string. The name string, being unique, can be the name of an individual, a concept or a role. This table containing the name-id mapping for the whole system. We have to notice here that in our system, the names need to be unique for all individuals, concepts and roles.

Database for Node

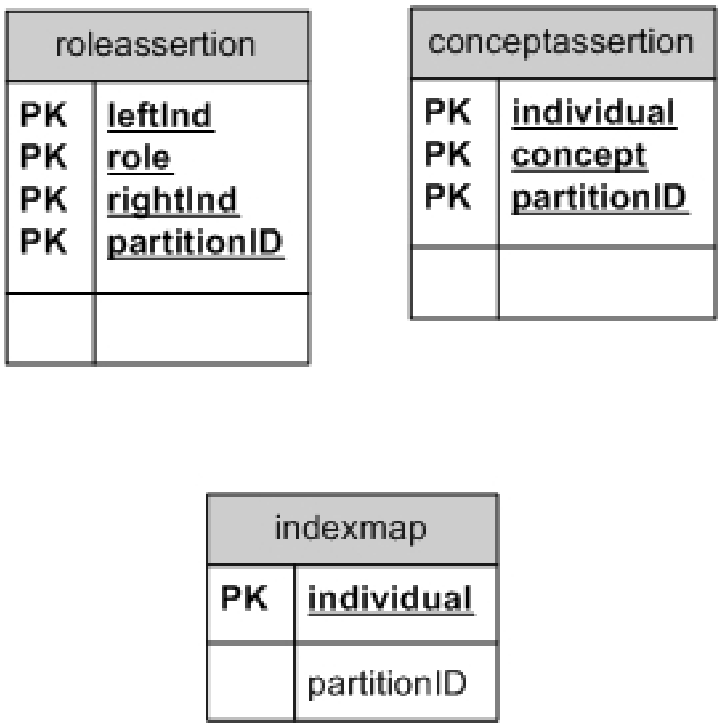


Figure 4.7: Node database

Since the assertions in the assertionology are only in simple form (i.e. $C(a)$, C is atomic concept or negation of atomic concept, or $R(a, b)$ with R

atomic role or inverse of atomic role), it is really simple to store them in the database. The only integrity constraints on the database is the existance of the id for individual/concept/role in the `idmap` table on the server. However, it is a complex issue when checking for integrity distributedly using database built-in mechanism, so here we ignore the integrity constraints on the table, and perform the validation in the application level instead.

4.4 Algorithm Implementation

4.4.1 Distributedly updating ABox

In this section we will discuss more details about performing the different types of updating in our distributed system.

Adding a concept assertion

From chapter 3, we have the algorithm for adding a concept assertion $C(a)$ is as following:

1. If $\phi_P(a) = \emptyset$ then add $\langle\{a\}, \{\}\rangle$ to P.
2. $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \cup \{a : C\}$
3. For each $ap_i \in P$ do: if $a \in Ind(\pi_S(ap_i))$ then $\pi_S(ap_i) = \pi_S(ap_i) \cup \{a : C\}$
4. $P = P \backslash \{\phi_P(a)\} \cup Reduce(\phi_P(a))$.

As we can see, the step 1,2 and 4 can be done solely on partition $\phi_P(a)$. The step 3, in contrast, needs to be executed on all the partitions that contain any assertion involving a. From the definition of partitioning algorithm, we know that if a partition ap_i contains any assertion involving a, then either a is a core individual of ap_i, or there must exists an individual b, s.t. b is the core individual of ap_i and there exists a role R that $(R(a, b) \in \mathcal{A}) \vee (R(b, a) \in \mathcal{A})$. So, from the partition of a, we look for all b that is not core individual of $\phi_P(a)$ and there exists a role assertion involving a and b. These b will be the core individuals of the partitions needed to be updated, which we call the *external partitions of* $\phi_P(a)$ *w.r.t* a

Then we have the following steps for performing the algorithm on our system:

1. When server update service receives an update request w.r.t. adding a concept assertion $C(a)$, it finds the node containing the partition for a and forwards the request to the node update service.

2. The node update service add the concept assertion into the correct partition, as well as all the external partitions of that partition w.r.t a.

3. The node update service request the partition manager to reduce the partition $\phi_P(a)$.

Removing a concept assertion

We have following algoritgm for removing a concept assertion from previous chapter:

1. $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \backslash \{a : C\}$

2. For each $ap_i \in P$ s.t. $a \in Ind(\pi_S(ap_i))$ then $\pi_s(ap_i) = \pi_s(ap_i) \backslash \{a : C\}$

3. For each $R(a, b)\phi_P(a)$, if $R(a, b)$ is not $\mathcal{O}$-separable then
   $P = P \backslash \{\phi_P(a), \phi_P(b)\} \cup \{Merge(\phi_P(a), \phi_P(b))\}$

4. For each $R(b, a)\phi_P(a)$, if $R(b, a)$ is not $\mathcal{O}$-separable then
   $P = P \backslash \{\phi_P(a), \phi_P(b)\} \cup \{Merge(\phi_P(a), \phi_P(b))\}$

Unlike in the case of adding a concept assertion, in this case only step 1 can be done solely on the partition of a. The step 2, where the update needs to be executed on all the external partitions of $\phi_P(a)$ w.r.t. a, can also be done as in the case of adding concept assertion. However, the steps 3 and 4 will require merging this partition with some other partitions, even the partitions from other nodes which will trigger transferring partitions from node to node.
We have the update is done as following steps:

1. When server update service receives an update request w.r.t. removing a concept assertion $C(a)$, it finds the node containing the partition for a and forwards the request to the node update service.

2. The node update service remove the concept assertion From the correct partitioin, as well as all the external partitions of that partition w.r.t a.

3. The node update service test the $\mathcal{O}$-separability of every roles involving a in $\phi_P(a)$, and ask the partition manager to merge partitions as needed.

Adding a role assertion

The algorithm for adding a role assertion from the last chapter is as following:

1. If $\phi_P(a) = \emptyset$ then add $\langle \{a\}, \{R(a,b)\} \rangle$ to P.
2. If $\phi_P(b) = \emptyset$ then add $\langle \{b\}, \{R(a,b)\} \rangle$ to P.
3. If $\phi_P(a) = \phi_P(b)$ then $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \cup \{R(a,b)\}$
4. Else If $R(a,b)$ is $\mathcal{O}$-separable w.r.t. $\pi_S(\phi_P(a))$ then
   a) Add $R(a,b)$ to $\pi_S(\phi_P(a))$ and to $\pi_S(\phi_P(b))$
   b) Add $\{b : C | b : C \in \pi_S(\phi_P(b))\}$ to $\pi_S(\phi_P(a))$
   c) Add $\{a : C | a : C \in \pi_S(\phi_P(a))\}$ to $\pi_S(\phi_P(b))$
5. Else
   a) Add $R(a,b)$ to $\pi_S(\phi_P(a))$
   b) $P = P \backslash \{\phi_P(a), \phi_P(b)\} \cup \{Merge(\phi_P(a), \phi_P(b))\}$

We can easily see that the problem in case of adding role assertion is not as simple as in case of concept assertion, since it might involves with two different partitions, and in worse case, of the two different nodes. Thus, we will perform the steps 1,2,3 and also the separability testing on the server. The corresponding node only executes the updating in steps 4 and 5. The details are as following:

1. When server update service receives update request for adding role assertion $R(a,b)$, it creates new partition if needed and requests to add it to a node.

2. If the two partitions for a and b are the same, then server update service request server partition manager to add the role assertion to the corresponding partition.

3. Else, server update service test the $\mathcal{O}$-separability of $R(a,b)$ w.r.t. $\phi_P(a)$, and forwards update work to node of b and node of a.

4. Each node update service check for the separability passed from server, and execute the update part corresponding to it.

Removing a role assertion

The algorithm for removing a role assertion is

1. If $\phi_P(a) \neq \phi_P(b)$ then
   a) $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \backslash \{R(a,b)\}$
   b) $\pi_S(\phi_P(b)) = \pi_S(\phi_P(b)) \backslash \{R(a,b)\}$
2. Else
   a) If $R(a,b)$ is $\mathcal{O}$-separable w.r.t $\pi_S(\phi_P(a))$ then
      - $\pi_S(\phi_P(a)) = \pi_S(\phi_P(a)) \backslash \{R(a,b)\}$
      - $\pi_S(\phi_P(b)) = \pi_S(\phi_P(b)) \backslash \{R(a,b)\}$

   b) Else $P = P \backslash \{\phi_P(a), \phi_P(b)\} \cup \mathit{Reduce}(\mathit{Merge}(\phi_P(a), \phi_P(b)))$

Same as the case of adding role assertion, removing role assertion also involving updating on two partitions, and part of work will be done on the server before forwarding to nodes.
The steps for removing role assertion are as following:

1. When receive the removing $R(a,b)$ request, server update service check the two partitions for a and b.
2. If the two partitions are not the same then server update service requests server partition manager to remove the role assertion from 2 partitions.
3. Else, if the role assertion is $\mathcal{O}$-separable then the updating is forwarded two the nodes that containing the two partitions. If not, the work is only forwarded to the node of a.
4. The node do the updating.

4.4.2 Distributedly updating TBox

As we discussed from previous chapter, the updating for TBox is much more costly than updating ABox since it can involve changes on every partitions. From the algorithm for updating

1. For each $R \in \mathcal{T}$, calculate $g^{\forall}_{\mathcal{O}}(R)$ before updating and $g^{\forall}_{\mathcal{O}} * (R)$ after updating.
   - If$(g^{\forall}_{\mathcal{O}}(R) \neq g^{\forall}_{\mathcal{O}} * (R))$ then $U_R = U_R \cup R$
2. For each $R \in U_R$, and for each $R(a, b)$:
   - If $R(a, b)$ is $\mathcal{O}$-separable but not $\mathcal{O}*$-separable then $P = P \backslash \{\phi_P(a), \phi_P(b)\} \cup Merge(\phi_P(a), \phi_P(b))$
   - If $R(a, b)$ is not $\mathcal{O}$-separable but $\mathcal{O}*$-separable then $P = P \backslash \phi_P(a) \cup Reduce(\phi_P(a))$

We can see that it is rather simple to divide the works between server and node. The steps are as following:

1. When server udpate service receives request for updating terminology, it calculate the update-role-set base on the information from terminology. Then it forwards this set to every nodes.
2. Every nodes checks every role in the set to see if there is any partitions need to be merged or reduced.

4.4.3 Distributedly reasoning

The reasoning service for our system is not much different from the algorithm proposed in previous chapters.

- If the query is terminological reasoning, it is done on the server.
- If the query is instance checking problem, it is forwarded to node. Node query service execute the reasoning using algorithm proposed in previous chapters. If there is a need of merging partitions, it ask the partition manager to carry out the merging.

4.4.4 Partitions allocation mechanism

An important aspect needed to be considered in the implementation of our system is the partitions allocation policy. This is a policy concerning with

- choosing a node to store a newly created partition.
- choosing the node to store the newly partition which are merged from different partitions from different nodes.

The main objective in designing a policy is to minimize the network communication, that is, trying to allocate the partitions so that all the merging operations will involve only the partitions on the same node. However, this totally depends on the application domains of each Descripion Logics Knowledge Base.
In our application, we implemented a simple policy that

- places a newly created partition on the node that has the least number of core individuals.
- places the merged partition on the node that
  - contains the individual a if the merging is triggered by updating concept assertion $C(a)$
  - contains the invididual a if the merging is triggered by updating role assertion $R(a, b)$
  - appears first in the nodes list managed by the server, if the merging is triggered by updating terminology, since this updating affect all the partitions.

This policy somehow guarantee the equally distribution of the data in the nodes, but not satisfy the main objective of minimizing the network communication. However, as we stated before, to develop a suitable policy for a Knowledge Base needs a deep researching on its application domains.

Chapter 5

Evaluation

5.1 System performance

We have tested our system on a real test data. The first testing system is composed of a server and 3 nodes, all being run in one laptop. The test data is the LUBM with 1 university and 1 department, and the total number of assertions in the data is 5738. After running the test several time, we got the average time needed to load all the data is around 200 seconds, which means it can load approximately 30 assertions per second. This seems to be slow, since the localized implementation for our update algorithm can load from 200-500 assertions per seconds, and the approach in [11] can load up to 1000 assertions per seconds.
To find out what is the reason that makes the system running slow, we have perfomed profiling the system, using JProfiler. The overall usefull running time on the whole system (the waiting time of the working threads are excluded) is listed in the table 5.1. Here, the real process is the time system

	Real process	Database process	Network process
seconds	65.966	4.63	129.404
%	32.98%	2.32%	64.70%

Table 5.1: The time usage in the system

really worked on the actual algorithm about updating, partitioning, etc. The database process time is the total time used for accessing database, and the network process time is the time needed for network communication between nodes. It shows in the table that the network communication

is really costly; it takes more than 60% of the total processing time, while the real processing time takes only 33%.
We also notice that the database accessing time is really small: only around 4.6 seconds, which is 2.32% total time. This is because in our system, almost all the database accessing is running in the background.
Considering only the real processing time, we will have that our system is capable of loading around 100 assertions per second. This is closer to the centralized implementation. However, we also need to notice that in our system, we introduced the implementation for updating terminology, which is much more time consuming comparing to updating assertionology only.

We also ran the testing on the same data with the different number of nodes. In the table 5.2, the times need to load all the data using 3 nodes, 4 nodes, 5 nodes and 6 nodes are listed.

	3 nodes	4 nodes	5 nodes	6 nodes
time(ms)	200437	209441	216146	223682

Table 5.2: Times for using different number of nodes

As we expected, increasing number of nodes used will increases the time needed to load the test data. This is caused by the increasing number of network communications, the most costly part of the system.

After testing with different number of node, we tested with the same number of nodes (3 nodes) but with different loading data (different universities and deparments in LUBM). The result is shown in the table 5.3. We can see that the time needed for loading the data is somehow linear with the number of assertions need to be loaded.

n assertions	3985	4499	4979	14407
time(s)	142.047	160.328	178.89	510.188

Table 5.3: System performance when loading different test data

There is also a big factors that we need to take care of. The test data we used, LUBM, has a very simple $\forall$-info structure. This leads to the fact that there are not many $\mathcal{O}$-inseparable role assertions in the assertionology, and so there are not many merging/reducing on the network. Considering another test data with a lot of merging/reducing during the loading, the time will greatly increase because of the possible suddenly large number of network communications and also database accessing.

From the data we collected about loading times using different number of nodes, we can see that increasing number of nodes used has slightly bad effect on the performance of the system. Thus, the number of node needs to be well considered, and new node should be added only when necessary.

5.2 Data distribution

Besides system performance, another factor we want to evaluate is the distribution of the data among nodes. The data collected using 3 nodes is listed in the table 5.4.

As we already stated in previous section, our test data did not trigger

Node	Total Partitions	Total Assertions	Assertions/partition	min	max
1	518	6089	11.7548	3	72
2	518	6822	13.1699	3	1596
3	518	5702	11.0077	3	77

Table 5.4: Partitions and assertions distribution among 3 nodes

many merging/reducing of the partitions, and because of our partition allocation policy, the number of partitions in the 3 nodes are somehow equally distributed.

The table also shows that we have a lot of stored assertions on the 3 nodes (18613) comparing to the test data, which has only about 6000 assertions. This is because we have alot of partitions created from test data, providing there is not many $\mathcal{O}$-inseparable role assertions. This factor was also mentioned in [16] as a down side of the algorithm.

The figure 5.1 illustrates the distribution of the assertions in the partitions on the first node. As shown in the figure, the number of assertions is really different between partitions. These differences actually illustrate the structure of the test data. From the table **??**, we notice that there is a partition in node 2 that has 1596 assertions. This is because that partition contains an individual that has many role assertion involving (an individual of `Deparment`).

We also ran the testing with 4, 5 and 6 nodes to collect distribution data. The distribution is somehow similar to the case of 3 nodes. Table 5.5 listed the data collected for 6 nodes.

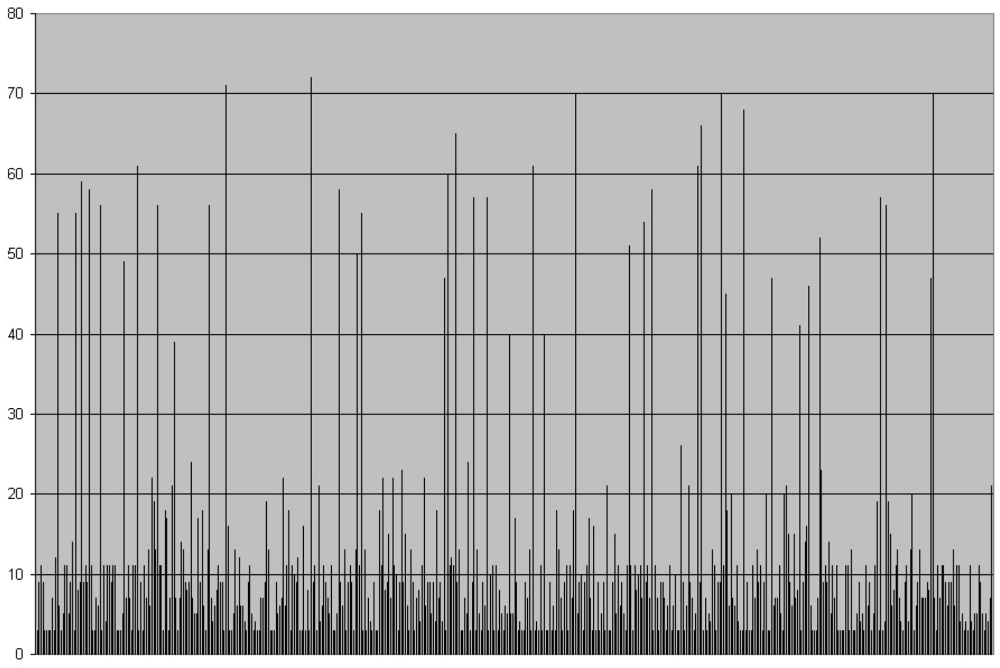


Figure 5.1: Assertion distribution among partitions in node 1 (3 nodes)

Node	Total Partition	Total Assertion	Assertion/partition	min	max
1	260	2989	11.4962	3	70
2	259	4129	15.9421	3	1596
3	259	2864	11.0579	3	77
4	258	3100	12.0155	3	72
5	259	2693	10.3977	3	76
6	259	2838	10.9575	3	74

Table 5.5: Partitions and assertions distribution among 6 nodes

The data distribution in our test is somehow nice, with the equally distribution of the partitions among nodes. However, this is the result of a simple testing data which does not instroduce many merging between partitions. Running test with more complex data, the partition allocation policy can be a critical factor deciding the system performance.

5.3 System performance with updating TBox

After testing the performance of the system for loading LUBM data, we also carried out the tests on the data using TBox updating. As we already discussed, updating TBox is an expensive kind of updates, since it might cause the merging/reducing the the whole ABox. In our experiments here, we investigated the performance of the system with different TBox updates that triggered different numbers of merging on the system. The results are shown in the table 5.6. As we see in the table, the time for updating is

n mergings	58	82	82	374	510	1650	3756
time (s)	6.73	8.73	6.22	15.63	105.49	1450.16	2715.34

Table 5.6: System performance when updating TBox

some how linear with the number of mergings when still in a low number range. However, when the number of mergings is higher (374 - 1650), the time growns exponentially. After that, it is some how linear again between 1650 mergings to 3756 mergings. This is because in the mid-range (374 - 1650), the update triggered the merging of more than 1000 partitions into 1 big partitioin, and with our merging strategy, the system tried to merge the first two partitions, saved new partition into database, then merged the new partition with the third one, then again saved into database, and this kept going on until all 1000 partitions was merged. This caused a very intensive database accessing, and made the performance drop dramatically.
From the result we can see that the database accessing time, in the case of intensive merging/reducing, have a really bad effect on the system's performance. This is even worse than the effect caused by network communication. However, it will be reduced if we can develop a better strategy for merging/reducing that optimizes the database accessing.

Chapter 6

Summary

Conclusions

In this master thesis, we have developed a distributed system for storing, updating and retrieving data relating to Description Logics knowledge base. The system was designed and implemented with the support for scalability and extendability. The number of database nodes in the system can easily be increased without changes in code or restating of the server and other nodes.
The functionalities of the system can also be conviniently extended with other implementations or algorithms, using the plugable design of the services, without recompiling the whole system.
The performance of our system is tested with an experiment data. There is a trade off between the performance and the scalability. With a well consideration for the system structure, the result is acceptable and promising for real applications with some refinement on the algorithms used.

Future works

The current system has all the basic functionalities and is capable of running with real data now. It is also can be used as a frame to develop further algorithm for updating. To improve the performance of the system, in the near future we will run the application with more complex data and develop a better partition allocation algorithm. However, we need to keep in mind that any partition allocation algorithm might be good only for Description Logics Knowledge base in certain application domains.

Another aspect we would like to improve in the near future is the resoning service in our system. The currently implementation only supports a very straightforward algorithm, which is impracticle for inferencing on real data. There are two approaches we would consider when develop the reasoning service:

- Reasoning based on current partitioning algorithm. In this approach, to perform the instance checking with complec concept description, we merge all the involving partitions and carry out the reasoning algorithm. This is the currently approach in our thesis, and it need more heuristic improvement in tableau algorithm to make it feasible in real time.

- Reasoning based on propagation between partitions. This is an approach proposed during the development of our thesis, but yet successfully to be proved. The idea of the approach is that given the complex instance checking, the reasoning is performed by one specific partition. The outcome of the first partition is then feeded into the second partition, and that keeps going on until we get the final result. This approach has an advantage that it doesnt need the transmission of the partitions over the network for the merging. However, a sound algorithm and its feasibility still need much more consideration.

Bibliography

[1] Remote method invocation white paper. java.sun.com. [-]

[2] Wikipedia, Description Logics. Internet. [8]

[3] F. Baader, I. Horrocks, and U. Sattler. Description logics for the semantic web. Internet, 2002. [-]

[4] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. Internet, 2000. [13]

[5] Franz Baader. *The Description Logic Handbook, Theory, Implementation and Applications.* University Press, Cambridge, 2003. [8, 13, 14]

[6] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation.* Elsevier, 2007. [-]

[7] Franz Baader and Ulrike Sattler. Expressive number restrictions in description logics. *Journal of Logic and Computation*, 9:319–350, 1999. [13]

[8] Francesco M. Donini, Maurizio lenzerini, Daniele Nardi, and Andrea Schaerf. *Principles of knowledge representation*, chapter Reasoning in description logics, pages 191–236. Center for the Study of Language and Information, Stanford, CA, USA, 1996. [-]

[9] Achille Fokoue, Aaron Kershenbaum, Li Ma, Chintan Patel, Edith Schonberg, , and Kavitha Srinivas. Using abstract evaluation in abox reasoning. *SSWS2006*, 1:61–74, 2006. [17]

[10] Enrico Franconi. Description logics online tutorial course. Internet. [-]

[11] Yuanbo Guo and Jeff Heflin. A scalable approach for partitioning owl knowledge bases. In *In: 2nd Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2006)*, 2006. [17, 55]

[12] Christian Halashek-wiener, Bijan Parsia, and Evren Sirin. Description logics reasoning with syntactic updates. In *In Proc. of the 5th Int. Conf. on Ontologies, Databases, and Applications of Semantics (ODBASE 2006*. Sringer Verlag, 2006. [24]

[13] I. Horrocks and U. Sattler. A description logic with transitive and inverse roles and role hierarchies, 1998. [14]

[14] Ian Horrocks. Reasoning with expressive description logics: Theory and practice. In *In: Andrei Voronkov, (ed) Proc. 18th Int. Conf. on Automated Deduction (CADE-18)*, pages 1–15. Springer, 2002. [-]

[15] Ralf Moeller Sebastian Wandelt. Island reasoning for ontologies. In *Proceedings of the Fifth International Conference on Formal Ontology in Information Systems (FOIS'08)*, 2008. [3, 17, 19, 21]

[16] Ralf Moeller Sebastian Wandelt. Updatable partitioning for alchi - ontologies. Ongoing, 2008. [3, 22, 26, 41, 57]

Appendices

Appendix A

Class diagrams

This appendix lists all the class diagrams for our system, including diagrams for:

DL language representation

Server Update Service and Server Query Service

Node Update Service and Node Query Service

Utility Service

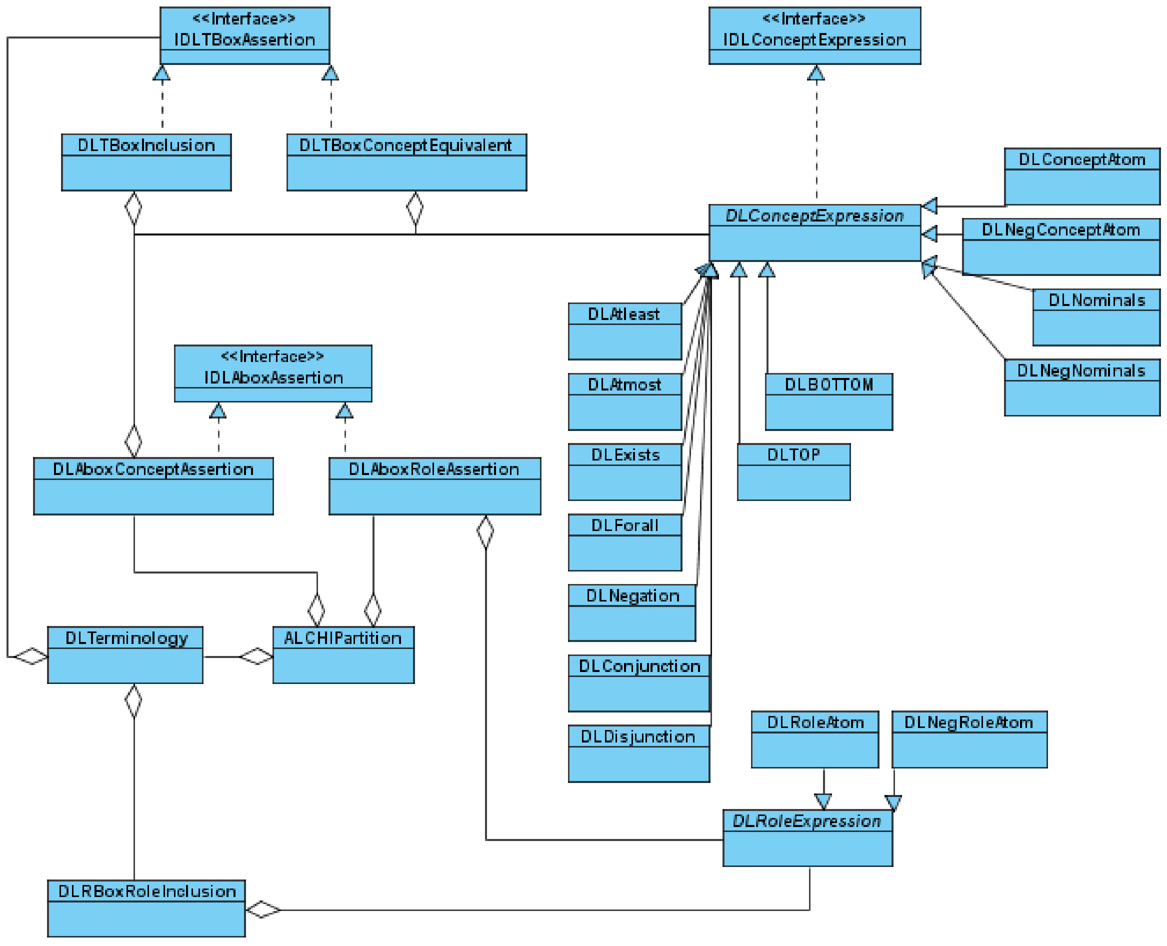
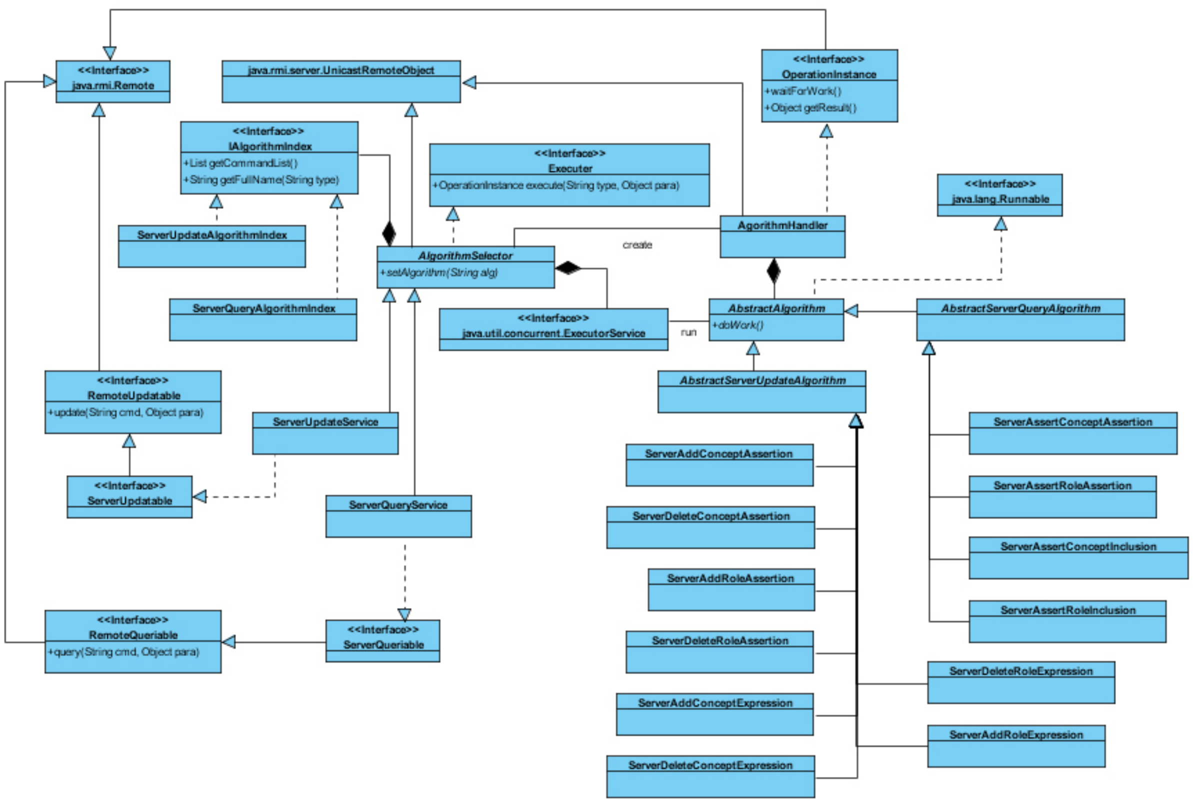


Figure A.1: Class diagram for description language representation



<<Interface>>
java.rmi.Remote
java.rmi.server.UnicastRemoteObject
<<Interface>>
OperationInstance
+waitForWork()
+Object getResult()
<<Interface>>
IAlgorithmIndex
+List getCommandList()
+String getFullName(String type)
<<Interface>>
Executer
+OperationInstance execute(String type, Object para)
<<Interface>>
java.lang.Runnable
ServerUpdateAlgorithmIndex
AgorithmHandler
create
AlgorithmSelector
+setAlgorithm(String alg)
ServerQueryAlgorithmIndex
<<Interface>>
java.util.concurrent.ExecutorService
run
AbstractAlgorithm
+doWork()
AbstractServerQueryAlgorithm
<<Interface>>
RemoteUpdatable
+update(String cmd, Object para)
AbstractServerUpdateAlgorithm
ServerUpdateService
ServerAssertConceptAssertion
ServerAddConceptAssertion
<<Interface>>
ServerUpdatable
ServerAssertRoleAssertion
ServerQueryService
ServerDeleteConceptAssertion
ServerAssertConceptInclusion
ServerAddRoleAssertion
ServerAssertRoleInclusion
<<Interface>>
RemoteQueriable
+query(String cmd, Object para)
<<Interface>>
ServerQueriable
ServerDeleteRoleAssertion
ServerDeleteRoleExpression
ServerAddConceptExpression
ServerAddRoleExpression
ServerDeleteConceptExpression

Figure A.2: Class diagram for Server Update and Query Services

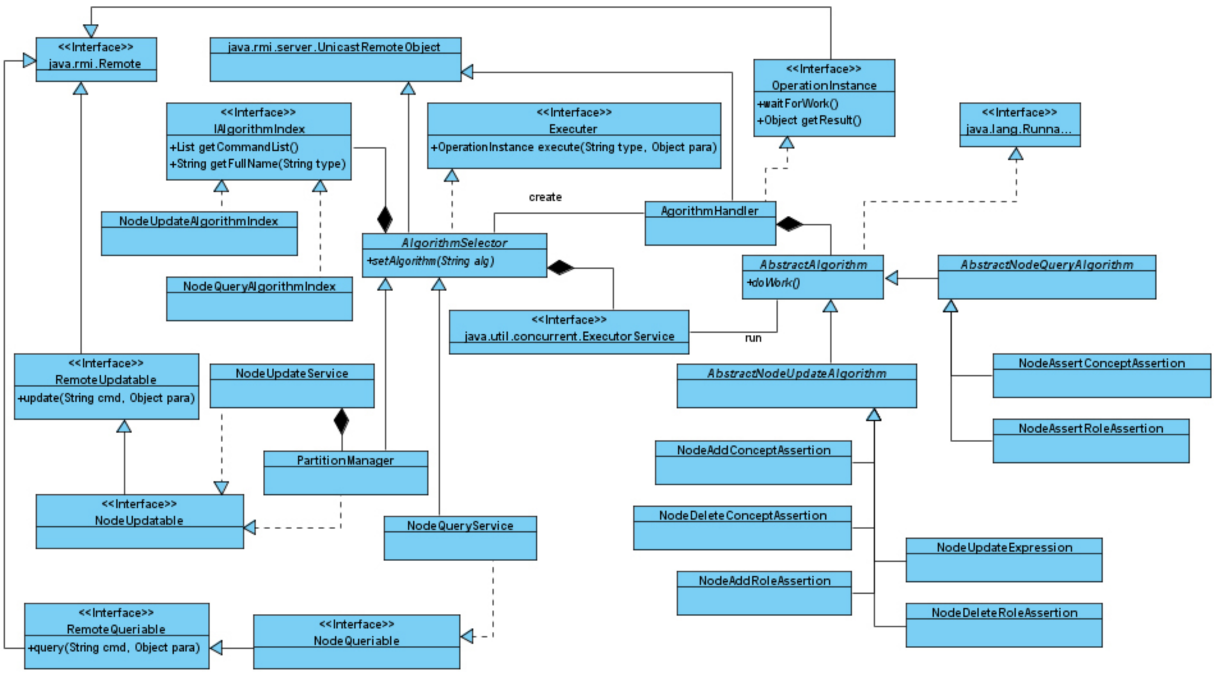


Figure A.3: Class diagram for Node Update and Query Services

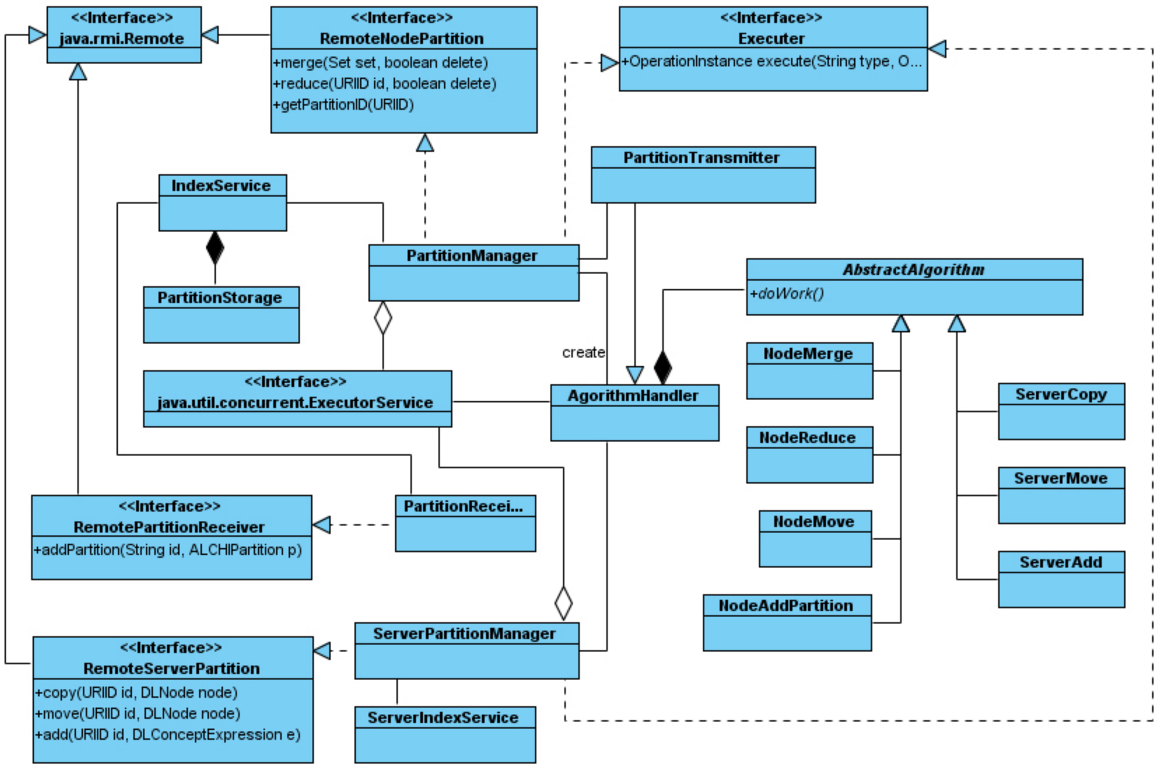


Figure A.4: Class diagram for the Partition Manager for server and nodes

Appendix B

Experiment data

This appendix lists all the experiment results running LUBM test data. Figure B.1 illustrates the distribution of the ABox assertions over all the partitions in the system. Data is collected from all nodes.
Besides, the overall distribution of partitions and assertions, running for 3 nodes, 4 nodes, 5 and 6 nodes are also listed in corresponding tables.

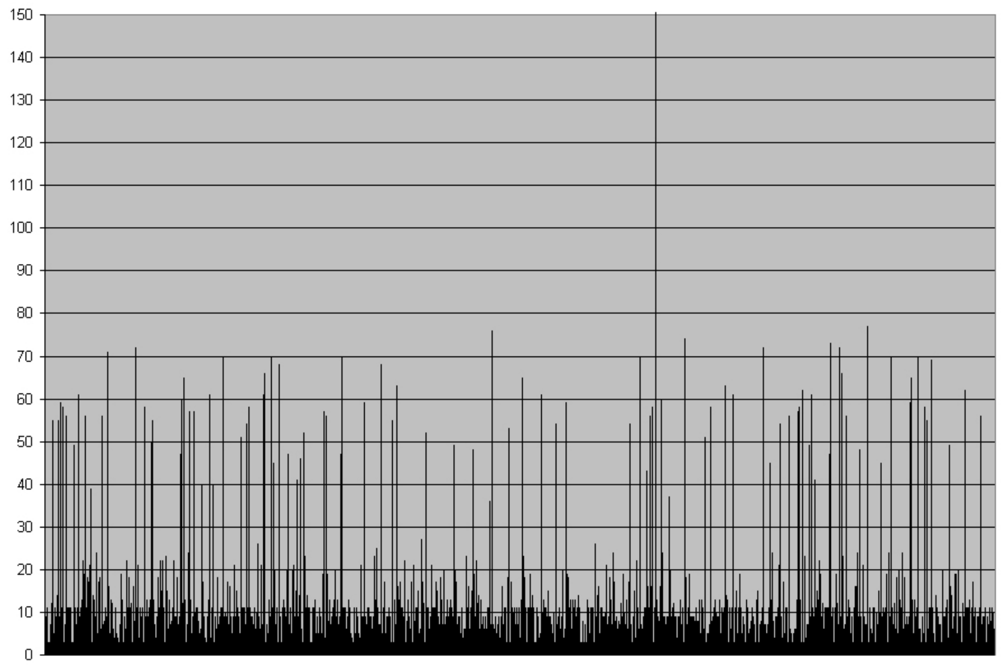


Figure B.1: The distribution of assertions amongs partitions

Node	Total Partitions	Total Assertions	Assertions/partition	min	max
1	518	6089	11.7548	3	72
2	518	6822	13.1699	3	1596
3	518	5702	11.0077	3	77

Table B.1: Partition distributions in system running with 3 nodes

Node	Total partitions	Total assertions	assertions/partition	min	max
1	389	4344	11.1671	3	77
2	388	4253	10.9613	3	73
3	389	4202	10.8021	3	76
4	388	5814	14.9845	3	1596

Table B.2: Partition distributions in system running with 4 nodes

Node	Total partitions	Total assertions	assertions/partition	min	max
1	311	3687	11.8553	3	72
2	311	3093	9.9453	3	68
3	311	3488	11.2154	3	76
4	311	3323	10.6849	3	73
5	310	5022	16.2	3	1596

Table B.3: Partition distributions in system running with 5 nodes

Node	Total Partition	Total Assertion	Assertion/partition	min	max
1	260	2989	11.4962	3	70
2	259	4129	15.9421	3	1596
3	259	2864	11.0579	3	77
4	258	3100	12.0155	3	72
5	259	2693	10.3977	3	76
6	259	2838	10.9575	3	74

Table B.4: Partition distributions in system running with 6 nodes

List of Figures

List of Tables