Scalable Reasoning over Large Ontologies

By

Md Rokan Uddin Faruqui

Submitted in partial fulfillment of the
requirements for the degree of
Masters of Science in Computer Science
at
Saint Francis Xavier University
Antigonish, Nova Scotia
April 2012

Saint Francis Xavier University
Department of
Mathematics, Statistics and Computer Science

The undersigned hereby certify that they have read a thesis entitled "**Scalable Reasoning over Large Ontologies**" by **Md Rokan Uddin Faruqui** in partial fulfillment of the requirements for the degree of **Masters of Science**.

Dated: _____

Supervisor: _____
Dr. Wendy MacCaull

Committee Member: _____
Dr. Man Lin

Committee Member: _____
Dr. Laurence T. Yang

St. Francis Xavier University

April 2012

Author:	Md Rokan Uddin Faruqui
Title:	Scalable Reasoning over Large Ontologies
Department:	Mathematics, Statistics and Computer Science
Faculty:	Science
Convocation	April 2012

Md Rokan Uddin Faruqui

Abstract

Ontologies are becoming increasingly important in various large-scale information systems such as health care systems. Ontologies give a coherent user-centric view of application domains. A major obstacle in developing ontology-based applications is the poor capability of current techniques to handle large ontologies. The web ontology language (OWL) is a semantic markup language for ontologies. Description logic (DL)-based OWL is used for developing ontologies where automated reasoning services are required. Classical DL reasoners such as *FaCT++*, *HermiT*, *Pellet* and *Racer* are main memory oriented and are not scalable. One of the approaches to improve the scalability is the direct manipulation of ontologies into databases by reasoning systems. However, using this method, DL-Lite is the maximal fragment that can be used for reasoning over ontologies stored into databases by query rewriting algorithms. We propose a hybrid approach which combines a logic-based reasoning strategy with materialization of knowledge into a relational database to enable reasoning over large ontologies based on the OWL 2 RL profile, a sublanguage of OWL 2, which is more expressive than DL-Lite and amenable to rule-based implementation. We also develop a restriction checker to check the ABox consistency of OWL 2 RL ontologies.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivation

An ontology is an explicit formal specification of a conceptualization that defines certain concepts of a domain and relationships among them [Gru93]. Ontologies are becoming of increasing importance in various large-scale information systems such as health care systems [RGJDGC+11, TZH07, Hor10]. Ontologies can play an important role in developing decision support systems. For example, guidelines for clinical practises are becoming more popular in the health care sector and ontologies can capture knowledge from clinical guidelines, standards, and practices. There is a number of biomedical ontologies, including GALEN [RR06], FMA [RJ03], NCI-Thesaurus [GFH+03], and SNOMED-CT [SPSW01] and these may be used as knowledge bases to drive decision support systems for healthcare systems. However, the effective use of ontologies requires not only a well-designed and well-defined ontology language, but also adequate support from reasoning tools.

Ontology reasoning is a methodology for extracting and inferring knowledge from ontologies. A formal specification for ontologies is necessary for processing and reasoning over ontologies. The Web Ontology Language (OWL) is a semantic markup language for ontologies that provides a formal syntax and semantics to represent ontologies. OWL also paves the way for manipulating ontologies effectively. The World Wide Web Consortium (W3C) declared two different standardizations for OWL: OWL 1 [MvH04] and OWL 2 [MGH+09]. The first standardization has three profiles: OWL Lite, OWL DL, and OWL Full. The OWL Lite profile is less expressive than the other two profiles and the OWL DL profile is based on description logic (DL) [MvH04]. Description logic-based OWL is a good candidate for defining ontologies where automated reasoning is required because it can utilize DL-based reasoning algorithms. The OWL Full profile has the maximal expressive power. However, reasoning over an ontology containing all the features of OWL Full is undecidable [MvH04]. Therefore, there is no polynomial time algorithms for performing reasoning over ontologies based on OWL Full. The second standardization, OWL 2, was declared for efficient and tractable reasoning. It trades some expressive power for more efficient reasoning [KMR10]. OWL 2 has three profiles: OWL 2 EL, OWL 2 QL, and OWL 2 RL; each of the profiles is useful in different application scenarios. Moreover, all the profiles exhibit a polynomial time complexity for standard reasoning tasks.

Numerous heuristics for reasoning were developed in the past and these were implemented in the classical DL reasoners such as *Racer* [HM01] , *HermiT* [MSH09], *FaCT++* [TH06], and *Pellet* [SPG+07]. These reasoners are main memory oriented, i.e., only the main memory of the computers is used to perform reasoning. However, in order to cover complex knowledge domains at various levels of detail, the size of real-life ontolo-

gies is becoming very large (millions of instances; for example, SNOMED[SPSW01] is a large biomedical ontology). Main memory-based reasoners are not particularly suitable for reasoning over ontologies with a large numbers of instances due to the inherent complexity of reasoning algorithms [TPR10].

Several approaches have been applied to improve the scalability of the reasoners. One of the most widely used approaches for scalability is database integration, i.e., utilizing secondary memory to increase efficiency. A number of reasoners such as OWLGres [SS08], OntMinD [AJPS10], QuOnto [ACG+05] use database integration by direct manipulation of ontologies into databases. However, these scalable reasoners support only a small fragment of DL logic called DL-Lite. DL-Lite is a subset of OWL-Lite and it is the maximal tractable fragment that supports efficient query answering using a relational database. So scalable reasoning with more expressive DL fragments is still a challenging problem. The goal of this thesis is to develop a scalable reasoning system for ontologies with large numbers of instances which supports a more expressive language fragment than DL-Lite. We developed a scalable reasoning system OwlOntDB based on the OWL 2 RL profile which combines two well-known methods: (i) the logic based approach - for efficient reasoning algorithms, and (ii) database technology - to handle a large amount of data.

Our work is based on the work from the several different communities, namely, the FOL, DL, OWL and relational databases communities and each community uses different jargons to refer the same things. We have attempted to clarify the relationships among the jargons from different communities.

1.2 Contributions

The newly standardized OWL 2 RL profile paves the way for designing a polynomial time reasoning system for large ontologies, which is our major contribution. In particularly, we have accomplished the following in this thesis:

- We developed a complete translation of the OWL 2 RL profile to datalog rules by extending the DLP[GHVD03] mapping, a mapping between OWL 1 ontologies and logic programs, to accommodate the new features of the OWL 2 RL profile (see Section 4.2). We implemented the translation that takes an OWL 2 RL ontology and generates a datalog program by translating each OWL 2 RL axiom to its equivalent datalog rule(s). We also developed a restrictions checker for some OWL 2 RL axioms that cannot, in general, be handled using the logic program (see Subsection 4.2.4). The restrictions checker is used to check the ABox consistency, one of the major reasoning tasks, of an ontology.

- We developed and implemented an algorithm to materialize the datalog version of an OWL 2 RL ontology into relational databases which uses forward chaining. The algorithm (see Algorithm 1) takes a datalog program and then performs forward chaining over the datalog program to infer implicit knowledge and translates the resulting datalog program to equivalent SQL statements (see Section 5.3.2) to store knowledge in a relational database.

- We developed a standard query interface for our scalable reasoning system by modifying an existing SPARQL-DL API. Since the SPARQL-DL API is built to interface with main-memory-based OWL 2 reasoners, we need some modifications to integrate with our system. We modified all the interfaces of the SPARQL-DL

API to integrate with our system (see Section 6.3). We also added a new feature to our query interface from SPARQL 1.1, a newly proposed standard for the SPARQL query (see Section 6.4).

1.3 Content Guide

The thesis contains the following chapters:

- Chapter 2 introduces the basic knowledge about Description Logics (DL) and web ontology languages. We focus on the basic DL $\mathcal{ALC}$ and the $\mathcal{SH}$ family of DL languages. We also introduce the OWL web ontology languages: OWL 1 and OWL 2 and their syntax and semantics, and ontology reasoning.

- Chapter 3 focuses on the scalability issues and also discusses some related works. An outline of our proposed solution is also presented in this chapter.

- Chapter 4 describes one of the major components of the thesis, namely the translation of ontologies into logic based programs.

- Chapter 5 presents the theory and database structures for storing and maintaining the inferred information from the translated ontologies.

- Chapter 6 describes the query interface that we developed by modifying the SPARQL-DL API to extract knowledge from ontologies.

- In Chapter 7, we present the programmatic description of the tool to store, infer, and query OWL 2 RL ontologies. We also evaluate some standard queries to verify the correctness of our proposed method and compare the results with other reasoners.

- Chapter 8 summarizes the contributions of this thesis, and then discusses the limitations of our work and a few future directions.

- In Appendix I, we present a datalog program generated by our translator from an ontology about people.

Chapter 2

Ontology Representation and Reasoning

This chapter introduces the basic knowledge about Description Logics (DL) and web ontology languages. It focuses on the basic DL $\mathcal{ALC}$ and the $\mathcal{SH}$ family of DL languages and also introduces the OWL web ontology languages OWL 1 and OWL 2 and their syntax and semantics.

2.1 Ontologies

An ontology is an explicit specification of a conceptualization [Gru93]. An ontology defines a model of some aspect of the world and introduces vocabulary relevant to a domain. Assume that we want to describe a domain - the *Heart*; the vocabulary associated within this domain is shown in Figure 2.1. An ontology adds specific meaning (semantics) of terms. For instance - *The heart is a muscular organ that is part of the circulatory system* - is a concept that has some terms with relevant semantics.

This concept can be formalized using first order logic as follows:

$$\forall x.[Heart(x) \rightarrow MuscularOrgan(x) \wedge \exists y.[isPartOf(x,y) \wedge CirculatorySystem(y)]]$$

The web ontology language (OWL) is an ontology representation language and the foundation of OWL is description logic (DL).

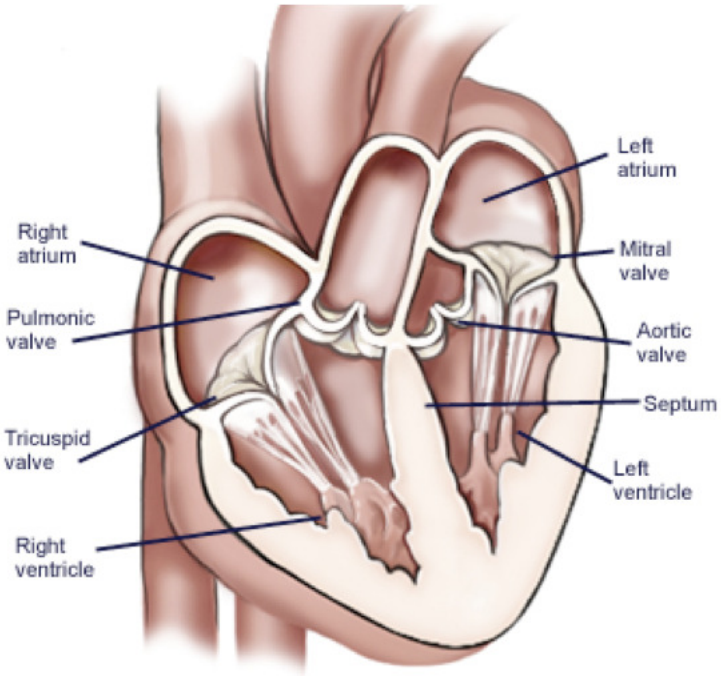

Figure 2.1: Domain: anatomy of a heart

2.2 Description Logic

2.2.1 Basic Notions

Description Logic (DL) is a fragment of first order logic designed for knowledge representation. DL defines relevant concepts of a domain, roles that are the relationships among

$$Heart \sqsubseteq MuscularOrgan \sqcap \exists\ isPartOf.CirculatorySystem \tag{2.1}$$

$$HeartDisease \equiv Disease \sqcap \exists\ affects.Heart \tag{2.2}$$

$$VascularDisease \equiv Disease \sqcap \exists\ affects.(\exists\ isPartOf.CirculatorySystem) \tag{2.3}$$

$$Patient(Mary) \tag{2.4}$$

Figure 2.2: An example of a DL-based ontology

the concepts, and individuals of the domain [BN03]. The basic syntactic building blocks are atomic concepts, atomic roles, and individuals[1]. It also provides a set of constructors to formalize complex concepts and relationships among the concepts of a domain. For example, a simple ontology about heart disease is represented in Description Logic (2.1) - (2.4). The first three axioms define the concepts *the heart is a muscular organ that is a part of the circulatory system*, *heart diseases are exactly those diseases that affects the heart*, and *vascular diseases are exactly those diseases that effects some parts of the circulatory system*, respectively. The fourth axiom asserts the fact that *Mary is a Patient*.

A DL knowledge base (KB) has two components: the Terminology Box (TBox) and the Assertion Box (ABox). The TBox introduces the terminology of a domain, while the ABox contains assertions about individuals in terms of this vocabulary. The TBox is a finite set of general concept inclusions (GCI) and role inclusions. A GCI is of the form $C \sqsubseteq D$ where C, D are DL-concepts and a role inclusion is of the form $R \sqsubseteq S$ where R, S are DL-roles. We may use $C \equiv D$ (concept equivalence) as an abbreviation for the two GCIs $C \sqsubseteq D$ and $D \sqsubseteq C$ and $R \equiv S$ (role equivalence) as an abbreviation

[1]Concepts and roles in DL are known as classes and properties in OWL, and unary and binary predicates in FOL.

for $R \sqsubseteq S$ and $S \sqsubseteq R$. The ABox is a finite set of concept assertions in the form of $C(a)$ and role assertions in the form of $R(a, b)$. In the ontology given in Figure 2.2, the TBox contains axioms (2.1) - (2.3) and the ABox contains an axiom (2.4).

2.2.2 The Basic Description Logics $\mathcal{ALC}$

The basic description language is $\mathcal{A}$ttributive $\mathcal{L}$anguage with $\mathcal{C}$omplements [SSS91]; in BNF form,

$$\mathcal{ALC} ::= \bot \mid \top \mid A \mid \neg C \mid C \sqcup D \mid C \sqcap D \mid \forall R.C \mid \exists R.C$$

where A, C, D are atomic concepts and R is an atomic role. $\mathcal{ALC}$ provides the top concept ($\top \equiv A \sqcup \neg A$), the bottom concept ($\bot \equiv A \sqcap \neg A$), boolean concept constructors ($\neg$: negation, $\sqcap$: intersection, $\sqcup$: union), as well as the universal quantifier ($\forall$) and the existential quantifier ($\exists$).

2.2.3 Extension of $\mathcal{ALC}$

$\mathcal{ALC}$ can be extended using one or more of the constructors found in the Table 2.1. One of the most important extensions is the $\mathcal{SH}$ DL family, which is extended from $\mathcal{ALC}$ with *transitive roles* (denoted by $\mathcal{R}_+$ in the Table 2.1) and *role inclusions* ($\mathcal{H}$). For example, we may have a transitive role *hasSibling*, a role *hasSister*, and a role inclusion axiom in an ontology about people using the logic of $\mathcal{SH}$. The role inclusion axiom in DL for this ontology is

$$hasSister \sqsubseteq hasSibling \tag{2.5}$$

Hence, if the ABox of an ontology contains assertions $hasSister(a, b)$ and $hasSibling(b, c)$

then $hasSibling(a, b)$ (from the role inclusion) and $hasSibling(a, c)$ (from the transitivity of $hasSibling$) will be asserted as axioms into the ontology.

The description logic - $\mathcal{SHOIN}$. The logic $\mathcal{SHOIN}$ is the extension of $\mathcal{SH}$ with nominals ($\mathcal{O}$), inverse roles ($\mathcal{I}$), and unqualified number restrictions ($\mathcal{N}$), and it serves as the logic foundation of the web ontology language OWL 1 [MvH04].

A nominal is a singleton set as its interpretation; i.e., there is one and only one individual in the interpretation of a nominal. For example, the country name CANADA can be modelled as a nominal, such that the cardinality $\#(CANADA)$ will always be 1 in any interpretation $\mathcal{I}$.

Inverse roles allow us to use a role in *both directions*. For example, suppose we have a role *advises* to denote the relation between a faculty member and a student; if we wish to define the concept of students who are advised by a faculty member, then we can describe it with the inverse role advises$^-$ as

$$Student \sqcap \exists\ advises^-.Faculty \tag{2.6}$$

A qualified number restriction ($\mathcal{Q}$) is one of the following form: $\geqslant nR.C$, $\leqslant nR.C$ or $= nR.C$, where R is an atomic role. For example, to make the assertion that every person must have exactly two parents who are also persons, or that a woman is a mother if she has at least one child which is a person, or that one person is married to at most one person, we have the following axioms:

$$Person \sqsubseteq (= 2\ hasParent.Person) \tag{2.7}$$

$$Mother \sqsubseteq Woman \sqcap (\geqslant 1\ hasChild.Person) \tag{2.8}$$

$$Person \sqsubseteq \leqslant 1\ marries.Person \tag{2.9}$$

Unqualified number restriction (denoted by $\mathcal{N}$ in Table 2.1) is a special case of a

qualified number restriction $\mathcal{Q}$ such that the qualification concept C is always the top concept $\top$. Therefore, an unqualified number restriction is of the following form: $\geqslant nR$, $\leqslant nR$ or $= nR$. For example, it may not be necessary to specify that a child is a person to define motherhood, hence, we can reformulate the axiom (2.8) as

$$Mother \sqsubseteq Woman \sqcap (\geqslant 1\ hasChild) \tag{2.10}$$

A functional role (denoted by $\mathcal{F}$ in Table 2) is an unqualified number restriction when $n = 1$. Therefore, a concept like $(= 2\ hasFather.Person)$ is an invalid DL concept if the role $hasFather$ is functional.

The description logic - $\mathcal{SROIQ}$. The most prominent extension of $\mathcal{SHOIN}$ is the logic $\mathcal{SROIQ}$ [HKS06], which is the basis of the newly standard web ontology language OWL 2 [MGH+09]. It is obtained from $\mathcal{SHOIQ}$ by allowing composite role inclusions (roles chain), and qualified number restrictions instead of unqualified number restrictions. A composite role inclusion has one of the forms: (i) $R \circ S \sqsubseteq R$, and (ii) $S \sqsubseteq R \circ R$, where $\circ$ is composition of binary relations. Composite role inclusions allow one to express an important feature of a knowledge base, called property chain - propagation of one property along another property [HS04]. For example, if we consider that each car contains an engine (i.e., $Car \sqsubseteq \exists\ hasPart.Engine$) and an owner of a car is also an owner of an engine (i.e., $\exists\ owns.Car \sqsubseteq \exists\ owns.Engine$), the we can use the following composite role inclusion axiom to express that an owner of a which contains b, also owns b.

$$owns \circ hasPart \sqsubseteq owns \tag{2.11}$$

In addition to the expressivity obtained from the logic $\mathcal{SHOIN}$ and composite role inclusions, the logic $\mathcal{SROIQ}$ allows for the following:

1. *disjoint roles.* $\mathcal{SHOIQ}$ allows us to express not only the disjointness of concepts but also the disjointness of roles. For example, the roles *sister* and *mother* can be declared as disjoint; if we assert $sister(a, b)$ and $mother(a, b)$ then the knowledge base will be inconsistent.

2. *reflexive and irreflexive roles.* $\mathcal{SROIQ}$ adds some useful constraints on ABoxes such as reflexivity and irreflexivity. For example, the roles *knows* may be declared as being reflexive and the role *hasSibling* may be declared as irreflexive.

3. *negated role asserstions.* In addition to positive role assertions, $\mathcal{SROIQ}$ allows negated role assertions like $\neg\ loves(Mac, Mary)$.

4. *local reflexivity.* $\mathcal{SROIQ}$ allows us to express concepts of the form $\exists R.Self$ to express local reflexivity. For example, *narcissist* can be defined as $\exists likes.Self$.

A model theory-based definition of $\mathcal{SROIQ}$ - the most expressive DL is given in Definition 2.2.1.

Definition 2.2.1. ***(Syntax and Semantics of $\mathcal{SROIQ}$-knowledge bases)*** *An Interpretation of a $\mathcal{SROIQ}$ knowledge base is a pair $\mathcal{I} =< \Delta^{\mathcal{I}}, .^{\mathcal{I}} >$ where $\Delta^{\mathcal{I}}$ is a non-empty set (the domain of interpretation) and $.^{\mathcal{I}}$ is a function that maps every concept to a subset of $\Delta^{\mathcal{I}}$, every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each individual name to an element of the domain $\Delta^{\mathcal{I}}$. The function $.^{\mathcal{I}}$ can be inductively extended to map the complex concepts and roles using the constructors shown in Table 2.1.*

An interpretation $\mathcal{I}$ satisfies a GCI, denoted by $\mathcal{I} \models C \sqsubseteq D$, iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. An interpretation $\mathcal{I}$ satisfies a role inclusion axiom $R \sqsubseteq S$ iff $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$. $\mathcal{I}$ is a model of a TBox $\mathcal{T}$, denoted by $\mathcal{I} \models \mathcal{T}$, if $\mathcal{I}$ satisfies all the concept inclusions and role inclusions in $\mathcal{T}$.

Constructor	Syntax	Semantics	
Top	$\top$	$\Delta^{\mathcal{I}}$	
Bottom	$\bot$	$\emptyset$	
Conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$	$\mathcal{ALC}$
Disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$	
Negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus D^{\mathcal{I}}$	
Exist Restriction	$\exists\ R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \exists b((a,b) \in R^{\mathcal{I}} \text{ and } b \in C^{\mathcal{I}})\}$	
Value Restriction	$\forall\ R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \forall b((a,b) \in R^{\mathcal{I}} \text{ implies } b \in C^{\mathcal{I}})\}$	
Transitive Role	R_+	$(a,b) \in R^{\mathcal{I}}$ and $(b,c) \in R^{\mathcal{I}}$ implies $(a,c) \in R^{\mathcal{I}}$	R_+
Composite Role	$R \circ S$	$\{(a,c) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \exists b((a,b) \in R^{\mathcal{I}} \text{ and } (b,c) \in S^{\mathcal{I}})\}$	
Nominal	$\{o\}$	$\{o^{\mathcal{I}}\}$	$\mathcal{O}$
Inverse Role	R^-	$\{(a,b) \mid (b,a) \in R^{\mathcal{I}}\}$	$\mathcal{I}$
Number	$\geqslant n\ R$	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\} \geqslant n\}$	
Restriction	$\leqslant n\ R$	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\} \leqslant n\}$	$\mathcal{N}$
Qualified Number	$\geqslant n\ R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \text{ and } b \in C^{\mathcal{I}}\} \geqslant n\}$	
Restriction	$\leqslant n\ R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \text{ and } b \in C^{\mathcal{I}}\} \leqslant n\}$	$\mathcal{Q}$
Functional	$\geqslant 1\ R$	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\} \geqslant 1\}$	
Role	$< 2\ R$	$\{a \in \Delta^{\mathcal{I}} \mid \#\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\} < 2\}$	$\mathcal{F}$
Reflexive Role	$Reflexive(R)$	$a \in \Delta^{\mathcal{I}}$ implies $(a,a) \in R^{\mathcal{I}}$	
Irreflexive Role	$Irreflexive(R)$	$a \in \Delta^{\mathcal{I}}$ implies $(a,a) \notin R^{\mathcal{I}}$	
Symmetric Role	$Symmetric(R)$	$(a,b) \in R^{\mathcal{I}}$ implies $(b,a) \in R^{\mathcal{I}}$	
Asymmetric Role	$Asymmetric(R)$	$(a,b) \in R^{\mathcal{I}}$ implies $(b,a) \notin R^{\mathcal{I}}$	
Disjoint Roles	$Disjoint(R_1, \ldots R_n)$	$R_j^{\mathcal{I}} \cap R_k^{\mathcal{I}}$ is empty for each $i \leq j$, $k \leq n$ and $j \neq k$	

Table 2.1: Syntax and semantics of $\mathcal{SROIQ}$ constructors

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

An interpretation $\mathcal{I}$ is a model of a knowledge base $\mathcal{K} =< \mathcal{T}, \mathcal{A} >$, denoted by $\mathcal{I} \models \mathcal{K}$, where $\mathcal{T}$ is the TBox, $\mathcal{A}$ is the ABox, iff $\mathcal{I}$ is model of $\mathcal{T}$ and $\mathcal{A}$. A knowledge base $\mathcal{K}$ entails an axiom γ, denoted by $\mathcal{K} \models \gamma$, if every model $\mathcal{I}$ of $\mathcal{K}$ satisfies γ.

2.2.4 FOL equivalence of DL

We presented the model theory-based interpretation of DL semantics in Definition 2.2.1 and in Table 2.1. However, DL is a decidable fragment of FOL, so it can also be represented in terms of a correspondence between DL and FOL, where classes correspond to FOL formulae with one free variable, properties correspond to FOL formulae with two free variables, and individuals correspond to FOL constants. The correspondence between DL and FOL is well-known. We present a translation of $\mathcal{SROIQ}$ logic into FOL in Table 2.2 which is taken from [MSS05]. In this DL to FOL translation, a translator operator π translates each DL statement into an equivalent FOL statement.

2.3 The Web Ontology Language

The Web Ontology Language (OWL) is a formal language for representing ontologies. Its predecessors DAML+OIL, DL, RDF, and RDFS have influenced the design of OWL. In particular, DL influenced the formal specification of the language, and the RDF/XML exchange syntax was influenced by a requirement to utilize the RDF syntax in the later version of the languages. OWL plays a vital role in realizing the Semantic Web vision of Tim Barners-Lee [TBLL01]. The Semantic Web defined by Barners-Lee is a web of machine processable data. Ontologies can be used to capture web data and relationships among them (semantics). Recently, OWL has become a de facto standard for publishing ontologies online. In this subsection, we present a brief history of the development of OWL the details of which can be found in [HPS10, HPSvH03].

The Resource Description Framework (RDF) [MM04] was the foundation of the semantic web. The RDF syntax allows one to express concepts (classes) and roles (prop-

Constructor	DL to FOL
Concept Inclusion	$\pi(C \sqsubseteq D) = (\forall a)(\pi_a(C) \rightarrow \pi_a(D))$
Atomic Concept	$\pi_a(A) = A(a)$
Atomic Role	$\pi_{a,b}R = R(a,b)$
Conjunction	$\pi_a(C \sqcap D) = \pi_a(C) \wedge \pi_a(D)$
Disjunction	$\pi_a(C \sqcup D) = \pi_a(C) \vee \pi_a(D)$
Negation	$\pi_a(\neg C) = \neg\pi_a(C)$
Exist Restriction	$\pi_a(\exists\ R.C) = (\exists b)(R(a,b) \wedge \pi_b(C))$
Value Restriction	$\pi_a(\forall\ R.C) = (\forall b)(R(a,b) \rightarrow \pi_b(C))$
Role Inclusion	$\pi_{a,b}(R \sqsubseteq S) = (\forall a)(\forall b)(\pi_{a,b}(R) \rightarrow \pi_{a,b}(S)$
Composition of Roles	$\pi_{a,b}(R_1 \circ \ldots \circ R_n) = (\exists b_1)\ldots(\exists b_{n-1})(\pi_{a,b_1}(R_1) \wedge \bigwedge_{i=1}^{n-2} \pi_{a_i,a_{i+1}}(R_{i+1} \wedge \pi_{a_{n-1},b}(R_n))$
Nominal	$\pi_a\{o\} = (a = o)$
Inverse Role	$\pi_{a,b}(R^-) = \pi_{b,a}(R)$
Number Restriction	$\pi_a(\geqslant n\ R) = (\exists b_1), \ldots, (\exists b_n)(\bigwedge_{i \neq j}(b_i \neq b_j) \wedge \bigwedge_i(R(a,b_i) \wedge \pi_{b_i}(C)))$ $\pi_a(\leqslant n\ R) = \neg(\exists b_1), \ldots, (\exists b_n)(\bigwedge_{i \neq j}(b_i \neq b_j) \wedge \bigwedge_i(R(a,b_i) \wedge \pi_{b_i}(C)))$
Transitive Role	$\pi_{a,b}(Transitive(R)) = \forall a,b,c(R(a,b) \wedge R(b,c) \rightarrow R(a,c))$
Local Reflexivity	$\pi_a(\exists R.Self) = R(a,\ a)$
Reflexive Role	$\pi(Reflexive(R)) = (\forall a)\pi_{a,a}(R)$
Irreflexive Role	$\pi(Ireflexive(R)) = \neg(\forall a)\pi_{a,a}(R)$
Symmetric Role	$\pi(Symmetric(R)) = (\forall a)(\forall b)(\pi_{a,b}(R) \rightarrow \pi_{b,a}(R))$
Asymmetric Role	$\pi(Asymmetric(R)) = (\forall a)(\forall b)(\pi_{a,b}(R) \rightarrow \neg\pi_{b,a}(R))$
Disjoint Roles	$\pi(Disjoint(R,S)) = \neg(\exists a)(\exists b)(\pi_{a,b}(R) \wedge \neg\pi_{a,b}(S))$

Table 2.2: Translation of $\mathcal{SROIQ}$ into FOL

erties). An RDF knowledge base is a collections of *(subject, predicate, object)* triples where each triple represents a binary relation (*predicate*) between individuals (*subject* and *object*). Here, subjects and objects are the individuals in DL and predicates are the roles in DL. The RDF Schema (RDFS) is an extension of RDF; it adds additional abilities to assert subclass and subproperty relationships.

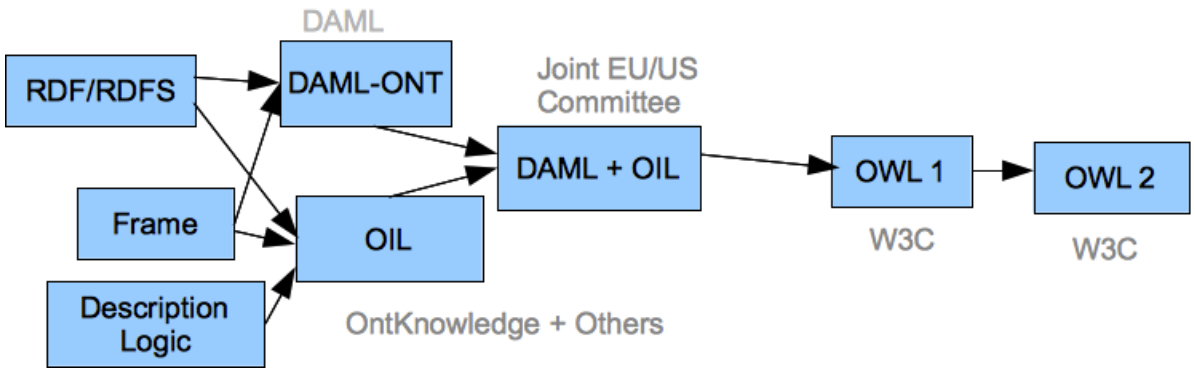

Figure 2.3: The OWL family tree

The DARPA Agent Markup Language (DAML) was developed by extending RDF with language constructors from object-oriented and frame-based knowledge representation languages. It was tightly integrated with RDFS, and while this was useful from a compatibility viewpoint, it suffered from an inadequate semantic specification [HM00] .

A large group of researchers, mostly Europeans, also developed a language called OIL (the Ontology Inference Layer) [FvHH+01] at the same time as DARPA was developed. OIL was the first ontology language to combine elements from description logics, frame languages, and web standards such as XML and RDF. The foundation of the language is the $\mathcal{SHIQ}$ (see Table 2) description logic and it indeed both XML and RDF syntaxes.

Later, the DAML and OIL groups jointly developed a new language, DAML+OIL [Hor02] by merging DAML and OIL. DL based model theory influenced its formal semantics. The DL derived language constructors of OIL were retained in DAML+OIL, but the frame structure was changed to allow the integration of RDF syntax more easily.

The first standardization by W3C that gives formal syntax and semantics for ontologies is the Web Ontology Language (originally called OWL and now referred to as OWL 1). DAML+OIL was the basis for OWL [MvH04]. OWL takes the basic fact-stating

ability of RDF and the class- and property-structuring capabilities of RDF Schema. Like RDF Schema, OWL can declare classes and can make a complete class hierarchy. OWL has several class constructors such as intersections, unions, and complements, etc., for building complex classes from atomic classes that go beyond the capabilities of RDFS. OWL can also declare properties, organize these properties into a subproperty hierarchy, and provide domains and ranges for these properties again as in RDFS. The domains of OWL properties are OWL classes, and ranges can be either OWL classes for the so called *OWL object properties* or externally-defined datatypes, such as string or integer, for the so called *OWL data properties*. OWL can state that a property is transitive, symmetric, asymmetric, functional, inverse functional, reflexive, irreflexive, or is the inverse of another property, here again extending RDFS.

We show the OWL family tree in Figure 2.3. OWL has two species: OWL 1 and OWL 2. OWL 1 [MvH04] was released as a W3C (World Wide Consortium) recommendation in February 2004 and OWL 2 [MGH+09] in October 2009. We describe OWL 1 and OWL 2 in the following subsections.

2.3.1 OWL 1

The foundation of the OWL 1 is $\mathcal{SHOIN}$. It comes with three sublanguages[2] with increasing expressive power.

- OWL-Lite sublanguage corresponds to the DL $\mathcal{SHIF}$ (see Table 2.1); it provides a useful subset of the OWL language features such as concept hierarchies and property restrictions. Reasoning over OWL-Lite is more efficient than over the other two profiles due to the expressive restriction imposed on OWL-Lite.

[2]In logic, profiles or sublanguages are known as *fragments*.

- OWL-DL sublanguage corresponds to the DL $\mathcal{SHOIN}$. Before the second standardization, it was the maximal decidable subset of OWL.

- OWL-Full sublanguage contains all the OWL language constructs and provides free, unconstrained use of RDF constructs. OWL Full also allows classes to be treated as individuals. However, use of OWL Full features generally leads to the loss of decidability.

2.3.2 OWL 2

The foundation of OWL 2 is $\mathcal{SROIQ}$. Therefore, it is more expressive than OWL 1 ($\mathcal{SHOIN}$); its additional expressive power was described in the subsection 2.2.3. OWL 2 has three sublanguages (profiles) which are discussed below:

- OWL 2 EL is aimed at application that uses ontologies with large TBoxes. So it is particularly suitable for ontologies that contain very large numbers of classes and properties. This profile captures the expressive power for which the basic reasoning problems can be performed in time that is polynomial with respect to the size of the ontology. A complete description of the profile can be found in [MGH+09].

- OWL 2 QL is particularly suitable for developing ontologies with a large number of Abox instances and is aimed at applications where query answering is the most important reasoning task (reasoning tasks for ontologies are discussed in Section 2.4). OWL 2 QL ontologies can be directly manipulated (i.e., create, insert, delete, update) by relational database management system. Therefore, conjunctive query answering can be performed using conventional relational database systems. It is also possible to design polynomial time algorithms for consistency, subsumption,

and classification reasoning as in OWL 2 EL. Although OWL 2 RL does include most of the main features of conceptual models, such as UML class diagrams and ER diagrams, the expressive power of the profile is quite limited. A complete description of OWL 2 QL can be found in [MGH+09].

- OWL 2 RL is more expressive than OWL 2 EL and OWL 2 QL. It is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It trades some expressivity of OWL 2 for efficiency. OWL 2 RL reasoning systems can be implemented using rule-based reasoning engines. Our work is based on this profile and we discuss this profile in Chapter 4

OWL 2 has several syntaxes. The standard syntax of the Semantic Web is RDF/XML. However, as RDF/XML is verbose and very hard to read, we use a syntax called the Manchester syntax which is easier to read. We use the Manchester syntax in the remainder of this thesis, precisely because it is designed for presentation to humans, rather than computers. Tables 2.3 and 2.4 summarize OWL 2 class constructors and axioms in Manchester syntax and their DL correspondences.

2.4 Ontology Reasoning

The effective use of ontologies requires not only a well-designed and well-defined ontology language, but also adequate support from reasoning tools. The users of ontologies are typically interested in obtaining information about relationships between concepts described in ontologies. For this task, reasoning tools are necessary to derive new knowledge from the knowledge explicitly stated in ontologies. We need ontology reasoners in two different phases:

OWL 2 Axioms in Manchester Syntax	DL Syntax
Individual: i Type C	i:C
Individual: i_1 Facts: P i_2	$< i_1, i_2 >: P$
Individual: i Facts: R v	$< i, v >: R$
SameIndividual: $i_1, \ldots, i_n$	$i_i \equiv i_{i+1}$ for $1 \leq i < n$
DifferentIndividual: $i_1, \ldots, i_n$	$i_i \neq i_{i+1}$ for $1 \leq i < n$
Class: C SubClassOf D	$C \sqsubseteq D$
Equivalent Classes: $C_1, \ldots, C_n$	$C_i \equiv C_{i+1}$ for $1 \leq i < n$
Disjoint Classes: $C_1, \ldots, C_n$	$C_i \sqsubseteq \neg C_{i+1}$ for $1 \leq i < n$
Class: C DisjointUnionOf: $C_1, \ldots, C_n$	$C \equiv C_1 \sqcup \ldots \sqcup C_n$
ObjectProperty: P SubPropertyOf Q	$P \sqsubseteq Q$
ObjectProperty: P EquivalentTo: Q	$P \equiv Q$
Equivalent Properties: $P_1, \ldots, P_n$	$P_i \equiv P_{i+1}$ for $1 \leq i < n$
Disjoint Properties: $P_1, \ldots, P_n$	$P_i \sqsubseteq \neg P_{i+1}$ for $1 \leq i < n$
ObjectProperty: P InverseOf Q	$P \equiv Q^-$
ObjectProperty: P Range C	$\top \sqsubseteq \forall P.C$
ObjectProperty: P Domain C	$\top \sqsubseteq \forall P^-.C$
ObjectProperty: P Characteristics: Transitive	$P^+ \sqsubseteq P$
ObjectProperty: P Characteristics: Functional	$\top \sqsubseteqslant 1\ P$
ObjectProperty: P Characteristics: Inverse Functional	$\top \sqsubseteqslant 1\ P^-$
ObjectProperty: P Characteristics: Reflexive	$\top \sqsubseteq \exists\ P.self$
ObjectProperty: P Characteristics: Irreflexive	$\exists\ P.self \sqsubseteq \bot$
ObjectProperty: P Characteristics: Symmetric	$P \equiv P^-$
ObjectProperty: P Characteristics: Asymmetric	$P \equiv \neg P^-$
DataProperty: R SubPropertyOf S	$R \sqsubseteq S$
DataProperty: R EquivalentTo: S	$R \equiv S$
Equivalent Properties: $R_1, \ldots, R_n$	$R_i \equiv R_{i+1}$ for $1 \leq i < n$
Disjoint Properties: $R_1, \ldots, R_n$	$R_i \sqsubseteq \neg R_{i+1}$ for $1 \leq i < n$
DataProperty: R Range C	$\top \sqsubseteq \forall R.C$
DataProperty: R Domain C	$\top \sqsubseteq \forall R^-.C$
DataProperty: R Characteristics: Functional	$\top \sqsubseteqslant 1\ T$

Table 2.3: OWL 2 axioms in Manchester Syntax and their corresponding DL semantics

OWL 2 Axioms in Manchester Syntax	DL Syntax
ObjectPropertyExpression: ObjectIntersectionOf: C_1 and ... and C_n	$C_1 \sqcap \ldots C_n$
ObjectPropertyExpression: ObjectUnionOf: C_1 or ... or C_n	$C_1 \sqcup \ldots C_n$
ObjectPropertyExpression: ObjectComplementOf: not C	$\neg C$
ObjectPropertyExpression: ObjectOneOf: $\{i_1 \ldots i_n\}$	$\{i_1 \ldots i_n\}$
ObjectPropertyExpression: ObjectSomeValuesFrom: P some C	$\exists P.C$
ObjectPropertyExpression: ObjectAllValuesFrom: P only C	$\forall P.C$
ObjectPropertyExpression: ObjectHasValue: P some $\{i\}$	$\exists P.\{i\}$
ObjectPropertyExpression: ObjectMinCardinality: P min n $[C]$	$\geqslant nP.C$
ObjectPropertyExpression: ObjectMaxCardinality: P max n $[C]$	$\leqslant nP.C$
ObjectPropertyExpression: ObjecExactCardinality: P exactly n $[C]$	$= nP.C$
DataPropertyExpression: DataSomeValuesFrom: R some C	$\exists R.C$
DataPropertyExpression: DataAllValuesFrom: R only C	$\forall R.C$
DataPropertyExpression: DataHasValue: P value $\{i\}$	$\exists P.\{i\}$
DataPropertyExpression: DataMinCardinality: R min n $[C]$	$\geqslant nR.C$
DataPropertyExpression: DataMaxCardinality: R max n $[C]$	$\leqslant nR.C$
DataPropertyExpression: DataExactCardinality: R exactly n $[C]$	$= nR.C$

Table 2.4: OWL 2 class expressions in Manchester Syntax and their corresponding DL semantics

- **reasoning at design time.** Ontologies may be very large and complex, like the well known SNOMED-CT [SPSW01] ontology that contains more than 400,000 classes. A reasoner is a must in order to develop such a large ontology. During ontology design, a reasoner determines whether concepts are consistent, i.e., non-contradictory and derives implied relationships. One can compute the concept hierarchy to check whether one concept is a subconcept of another and which concepts are synonyms. The computed concept hierarchy can be used in the design phase to test whether the concept definitions in the ontology have the intended consequences or not. Therefore, reasoners help to develop meaningful (i.e., all named classes can have instances), correct (i.e., captured intuitions of domain experts), minimally redundant (i.e., no unintended synonyms) ontologies. Reasoners also play an important role in the interoperability and integration of different ontologies.

- **reasoning in deployment.** In addition to the reasoning tasks important at design time, query answering to retrieve data or infer relationships between instances and ontology classes is important reasoning task at runtime.

2.4.1 Reasoning Tasks

Many reasoning tasks for OWL correspond to standard reasoning tasks, i.e. tasks that allow one to draw new conclusions about the knowledge base or check its consistency. Typical reasoning tasks are related to the TBox and the ABox of an ontology. The reasoning tasks related to the TBox and ABox of an ontology are given below:

- **TBox reasoning tasks**. Reasoning tasks typically considered for TBoxes are the following:
  - *satisfiability checking* - checks whether a class C can have instances according to the current ontology.
  - *subsumption checking* - checks whether a class D (respectively, property R) subsumes a class C (respectively, property S) according to the current ontology.
- **ABox reasoning tasks**. ABox reasoning tasks usually come into play at runtime of the ontology. Reasoning tasks typically considered for ABoxes are the following:
  - *consistency checking* - checks whether the ABox is consistent with respect to the TBox. ABox consistency checking is an important reasoning task. ABox consistency can be explained as follows: suppose, we have a TBox axiom *IrreflexiveObjectProperty(hasSibling)* (*hasSibling* is an irreflexive object property) and then we assert an Abox axiom *hasSibling(Bob, Bob)*. Now the ABox of the ontology will be inconsistent with respect to the Tbox axiom because *Bob* can not be the sibling of himself (*irreflexivity*).
  - *instance checking* - checks whether a given individual a is an instance of C.
  - *class instances retrival* - retrieves all individuals that instantiate a class C or finds all named classes C that an individual a belongs to.
  - *property instances retrival* - retrieves all individuals x which are related with an individual a via a property P. Similarly, we can retrieve the set of all named properties P between two individuals a and b, ask whether the pair (a, b) is an instance of P, or ask for all pairs (a, b) that are instances of P.

We developed a scalable reasoning system for reasoning over large Aboxes. For ABox reasoning, we use logic-based inferencing to infer implicit knowledge from ontologies. We use an existing DL reasoner for reasoning over TBoxes.

Chapter 3

Scalable Reasoning System

This chapter focuses on the scalability issues and also discusses some related works. We also present an outline of the proposed solution in this chapter.

3.1 The Importance of Scalability

Ontologies are becoming of increasing importance in various systems such as large-scale information systems for health care. Ontologies give a coherent user-centric view of application domains. Reasoning over ontologies with large numbers of instances is an important aspect of developing large-scale ontology-based scalable information systems. Numerous heuristics for reasoning were developed in past. One of the most prominent reasoning strategies is the Tableau decision procedure, as first introduced in [SSS91]. The Tableau algorithm is implemented in many reasoners including *Racer* [HM01], *FaCT++* [TH06], and *Pellet* [SPG$^+$07]. These reasoning algorithms have two major drawbacks; one is the high computational complexity (time complexity) and another is the lack of scalability (high space complexity). The worst case time complexity for OWL 1-

DL ($\mathcal{SHOIN}$) is NExpTime and for OWL 2-DL ($\mathcal{SROIQ}$) is 2NexpTime [TPR10]. However, several optimization techniques are already developed to improve the reasoning efficiency of Tableau-based reasoners. For instance, Hyper-Tableau is an optimized Tableau-based decision procedure [MSH09]. These optimized reasoners can handle effectively ontologies with large TBoxes. But, due to in-memory computation, the computation performance degrades when the ABox of an ontology has a large number of instances [dMRGAM08]. Therefore, reasoning over ontologies with large ABoxes is still challenging.

Several approaches have been applied to improve the scalability of the reasoners. One of the most widely used approaches is the database integration – utilizing secondary memory to increase efficiency. Relational databases are tailored to manipulate large amount of data. The database integration technique to improve the scalability is used in many reasoners including OWLGres [SS08], OntMinD [AJPS10], QuOnto [ACG$^+$05]. A classical approach was used to reduce the computational complexity namely, profiling or defining a tractable subset of OWL 1. DL-Lite is the maximal tractable subset of OWL 1 that can be used to develop relational database-based scalable reasoners [CGL$^+$07]. DL-Lite-based scalable reasoners directly map ontologies into relational databases and apply query rewriting techniques to perform reasoning over the ontologies [CGL$^+$09, ACKZ09]. A simple example of the direct mapping of some classes and properties of an ontology to a relational database is given in Figure 3.1 where eight classes, namely, "Person", "Caregiver", "Patient", "InPatient", "OutPatient", "Medication", "Pain", and "PainLevel" and four object properties, namely, "isFeeling", "hasMedication", "hasCareer", and "hasPainLevel" are mapped into twelve relational database tables.

However, DL-Lite is a very inexpressive language; it was designed to capture basic

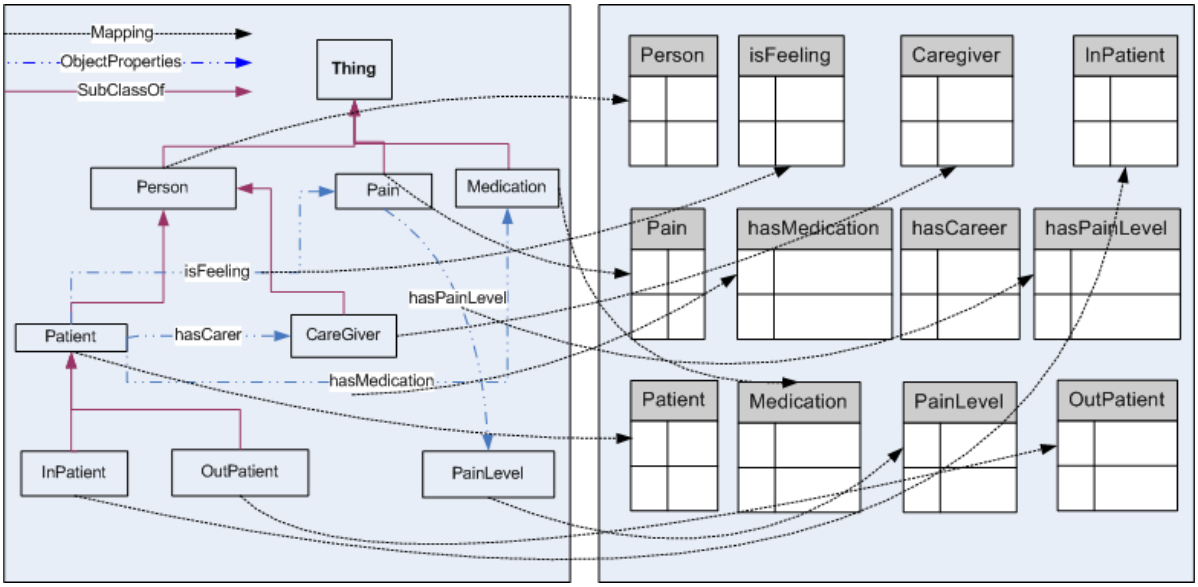

Figure 3.1: A direct mapping between an ontology and a relational database

ontologies, while the complexity of reasoning and answering complex queries over DL-Lite is very low [CGL+09]. The new standardization of the Web Ontology Language OWL 2 [MGH+09] which includes the 3 profiles as discussed in 2.3 addresses the trade-off between logical expressivity and scalability that is inherent to formal knowledge representation. The OWL 2 RL profile, which is based on $\mathcal{SROIQ}$, is more expressive than DL-Lite and suitable for scalable implementations without sacrificing too much expressive power. Instead of direct mapping between ontologies and relational databases, we used a logic-based approach to develop a scalable reasoning system for OWL 2 RL. In the following section, we present some related works and then describe our approach.

3.2 Related Works

There has been considerable interest recently in the design and development of scalable storage and inference systems for ontologies with large numbers of instances. Considerable work has already been completed on designing optimal database schemas for storing large ontologies in relational databases which have been implemented in various systems such as SOR [LMZ+07], Oracle Semantic, etc. Designing optimal database schemas to reduce redundant data, improving query performance, etc., is not the focus of this thesis; we used one of the already proposed database schemas and focused only on reasoning over large ontologies. There are two main approaches to developing scalable reasoning systems for large ontologies: (i) database integration [AJPS10]; and (ii) modularization of large ontologies [GPSK06]. We discuss each below.

3.2.1 Database Integration

In this subsection, we discuss different database integration methods used for reasoning over large ontologies:

- *Mapping onto Logic Programming (LP) rules.* The goal of this approach is to reuse existing LP inference algorithms and implementations that are efficient for reasoning over large amounts of data. Basically, the approach consists in translating DL axioms into LP rules in datalog. One of the major disadvantages is the use of deductive databases that are not widely used in real applications because there are no industry standard matured tools, like Oracle and MySQL, for deductive databases. This approach is implemented in the Coral Deductive DBMS [RSSS93].

- *KBMS-DBMS two-phase coupling.* A two-phase loose coupling approach combines a knowledge-base system (KBMS) with a database system where data is stored [BB93]. Inferences about stored database objects are performed by first querying the relational databases. The results are then sent to the KBMS to detect inconsistencies and perform deductions that the DBMS cannot compute. In this way, database objects are periodically input into the KBMS reasoner. This system still suffers from the problem of taking a long time for reasoning over large ontologies because for each query, it invokes the in-memory-based KBMS.

- *DL reasoner-DBMS coupling.* A DL reasoner and DBMS coupling-based system combines a DL reasoner to generate class hierarchies for the TBox and a relational DBMS to store the ABox. A retrieval query to compute the set of instances of a class is answered using a combination of database queries and TBox reasoning. Coupling approaches face potentially prohibitive costs for communication between the two components. The InstanceStore system [HLTB04] uses this approach.

- *Embedding reasoning in DBMS.* In this method, a forward-chaining inference engine is embedded with a relational DBMS. A number of different implementations embed rule-based reasoners in DBMS. For example, Oracle Semantic and OntoMinD [AJPS10]. Unlike Oracle Semantic, OntoMinD performs reasoning incrementally at each update; however, it only supports a DL-lite language. Oracle semantic is a proprietary solution that supports OWL Prime.

- *Performing reasoning during query evaluation.* In this approach, reasoning is performed as part of query processing. In a first step, it reformulates the user's query as a new query that results from reasoning over the axioms defined in the TBox.

The QuOnto [ACG+05] reasoner uses this method. A drawback of this approach is that inferencing for the same query is frequently repeated because inferred information is not stored in the database.

- *KAON2* is a DL reasoner developed at the University of Manchester and the University of Karlsruhe [Mot08]. In KAON2, the ontology is translated into a logic program and then it is materialized into a deductive database for querying and storing the information. The system can handle $\mathcal{SHIQ}$ knowledge bases, which is equivalent to OWL Lite without nominals. This approach is similar to our approach, except we develop a scalable reasoner for OWL 2 RL which is more expressive than OWL Lite and materialize the information to a relational database rather than to a deductive database.

3.2.2 Modularization of Ontologies

The main use of ontologies is to formalize the vocabulary of an application domain. In many cases, only a fragment of the defined vocabulary is of interest, which is knows as a module. A module can be considered to be a subset of an ontology that makes sense (i.e., is not an arbitrary subset randomly built) and can somehow exist separated from the whole ontology, although not necessarily supporting the same functionality as the whole. The extraction of the module of interest is an alternative way for using large ontologies. There are various approaches to extract modules; we discuss only two main categories:

- *Logic-Based Approach.* In a logic-based approach, all the techniques are based on rigorous logical foundations for extracting modules. Research in modular on-

tologies has drawn significant attention in the recent past and many approaches and formalisms have been developed for modular ontologies. In many approaches, formal algorithms are developed based on sound logical foundations for modules extraction that are correct and complete [PJC09].

- *Graph Theory-Based Approach.* All the graph theory-based approaches adopt graph traversal algorithms for traversing the ontology hierarchy and applying heuristics to extract relevant modules. This approach is very simple and useful. However, the approach does not take into consideration the underlying semantics of the ontology, and hence does not guarantee the generation of sound and complete modules.

3.3 The Proposed Solution

We develop a scalable storage and reasoning system for OWL 2 RL ontologies. In our approach, we combine the following approaches:

- *Translation to logic programming (LP).* Logic programming has efficient inference algorithms that are suitable for reasoning over large ontologies. The main advantage of this approach is to reuse existing inference algorithms and implementations. In this approach, an ontology is translated into a logic program, then inference algorithms for logic programs are used for reasoning.

- *Materialization and query evaluation through relational databases.* Materialization is an approach of storing inferred information from a knowledge base. In this method, an inferred knowledge base is obtained by performing logic-based

inferencing over knowledge bases; the inferred knowledge is then stored in relational databases. Database Management Systems (DBMS) are tailored toward the processing of large amounts of data, and the efficient manipulation of such data. Hence, database integration with reasoning systems improves the scalability of the system. In this approach, user's queries are reformulated to extract inferred information stored in databases.

We present the architecture for our proposed scalable storage and reasoning system for OWL 2 RL ontologies in Fig 3.3. The process can be divided into four steps as follows:

- Step 1: *Expanding the terminology box (TBox).* We use a DL reasoner for TBox expansion. It mainly infers a complete subsumption relationship among classes; i.e., it generates a complete class hierarchy. Several DL reasoners, including *Pellet*, are mature enough for TBox reasoning in memory. Our main focus is ABox reasoning. An example of TBox reasoning is given in Figure 3.2 where the equivalence property of $Woman$ is used to get a new definition of the property $Mother$.

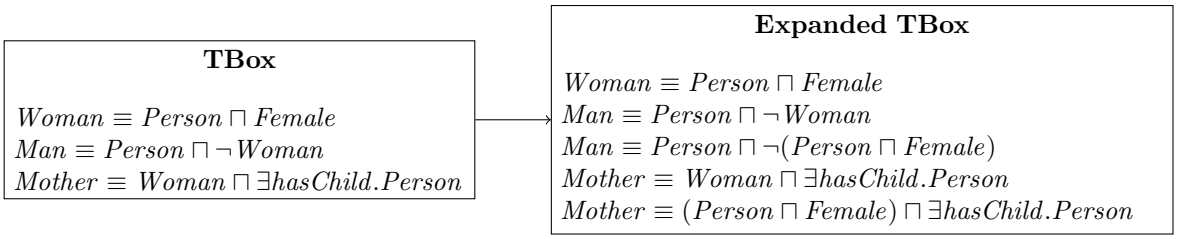

Figure 3.2: TBox reasoning

- Step 2: *Translating ontologies into datalog programs.* Our approach to reasoning is to express inference tasks for the OWL 2 RL ontology in terms of inference tasks for the rule language datalog. Datalog is a simple rule language stemming from

Manchester Syntax	Datalog
Class: Mammal *SubClassOf* Animal	$Animal(x) \leftarrow Mammal(x)$
Class: Cat *SubClassOf* Mammal	$Mammal(x) \leftarrow Cat(x)$
Individual: mr_whiskers *Type* Cat	$Cat(mr\_whiskers)$
ObjectProperty: likes *InverseOf* liked_by	$liked\_by(y, x) \leftarrow likes(x, y)$
ObjectProperty: ancestor *Characteristics* Transitive	$ancestor(x, z) \leftarrow ancestor(x, y)$ and $ancestor(y, z)$
Individual: Bob *Type* Human	$Human(Bob)$

Table 3.1: Datalog translation of a fragment of an OWL ontology

Prolog and is a decidable fragment of first order logic. In this step, we extend the description logic framework known as DLP [GHVD03] to translate an OWL 2 RL ontology into a datalog program. We develop a complete translation of OWL 2 RL into datalog. We use the OWL API to parse the inferred OWL 2 RL ontology and extract all the logical axioms from an ontology. Then, we translate each logical axiom into its equivalent datalog rule(s). We describe the translation procedure in Chapter 4. This step is demonstrated by an example given in Table 3.1. The table contains some OWL axioms and their equivalent datalog rule(s).

- Step 3: *Storing and materializing the ontology into a database.* The term materialization is used to characterize approaches where the information that is inferred by deduction is stored in the ontology and maintained when the ontology is updated.

  Datalog rules are closely related to operations in relational algebra. So materialization can be performed by representing datalog rules in the form of SQL commands. In this step, we use an efficient database schema to store the ontology, and develop a translation procedure to generate an equivalent SQL statement for each datalog

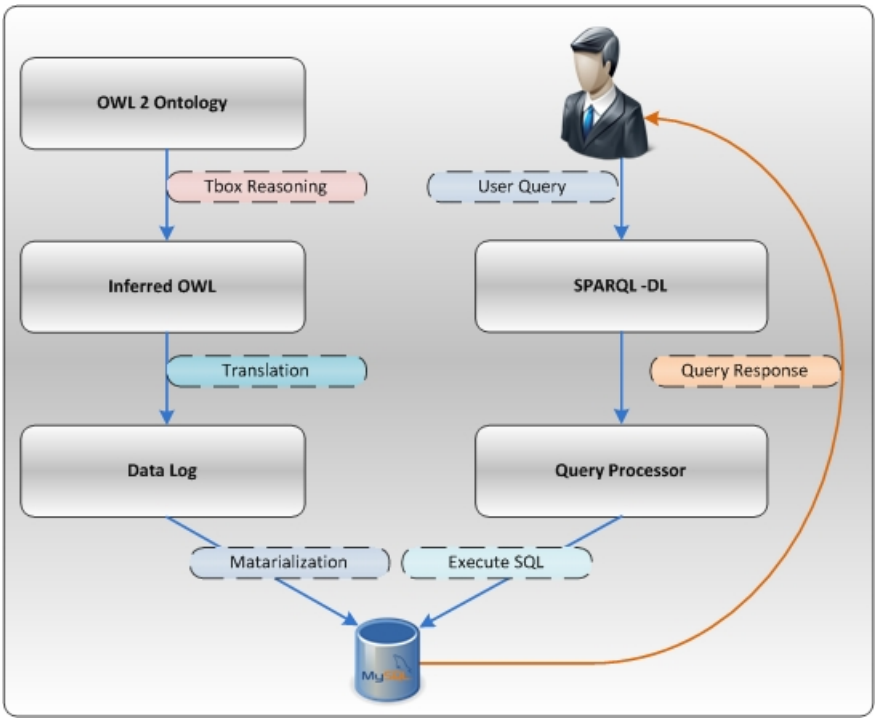

Figure 3.3: The system architecture of the scalable reasoning system

rule for materialization. We explain the details of the materialization technique in Chapter 5. For example, the ontology translated into the datalog program in step 2 can be materialized and stored into the database using the following SQL commands:

```
INSERT into animal(X) SELECT X FROM mammal.
INSERT into mammal(X) SELECT X FROM cat.
INSERT into cat(mr_whiskers).
INSERT into liked_by(X,Y) SELECT X,Y FROM likes.
INSERT into likes(X,Y) SELECT X,Y FROM liked_by.
INSERT into ancestor(x,z) SELECT t1.y AS x, t2.z AS z
      FROM ancestor AS t1 INNER JOIN ancestor AS t2 ON t1.y=t2.x.
INSERT into human(Bob).
```

- Step 4: *Query evaluation.* A standard query language is necessary to access the materialized knowledge without knowing the low level details of our database schema. SPARQL-DL [SP07] is a query language for OWL 1 and an extension of this work

is implemented in the *SPARQL-DL API* for querying OWL 2 ontologies [SPA11]. But this API is developed to integrate with standard main-memory based reasoners such as Fact++, Pellet etc. In this step, we developed all the necessary interfaces for standard reasoning tasks using SQL statements and integrated these interfaces with the SPARQL-DL API. We describe the query evaluation procedure in Chapter 6. An example of a SPARQL-DL query to find all the instances of the wine type *Chardonnay* from a materialized wine ontology is given below:

Query 3.3.1. *Find all the instances of the wine type Chardonnay.*

```
PREFIX wine: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>
SELECT ?i
    WHERE
      { Type(?i, wine:Chardonnay) }
```

In summary, our scalable reasoning system first takes an OWL 2 RL ontology as input and uses a DL reasoner to create a complete class hierarchy from the OWL 2 RL ontology, then it translates each axiom and fact of the ontology to its equivalent datalog rule(s) using our extended DLP mapping. The materialization technique is used to infer implicit knowledge from the datalog version of the ontology and to store asserted and inferred knowledge in a relational database. Our reasoning system uses SPARQL-DL as a query language to retrieve knowledge from the relational database.

Chapter 4

Translating Ontologies into Datalog Programs

This chapter focuses on the translation for OWL 2 RL ontologies into datalog programs and the representation of inference tasks of OWL 2 RL in terms of datalog queries.

4.1 Introduction

Our approach to reasoning is to express inference tasks for the OWL 2 RL profile in terms of inference tasks for the rule language datalog. We use the OWL 2 RL profile, an expressive language, for ontology specifications .The design of OWL 2 RL was influenced by Description Logic Programs (DLP) [GHVD03] and pD^* [tH05], which allows us to translate OWL 2 RL ontologies into logic programs. OWL 2 RL trades some of the expressivity of OWL 2 for efficiency [MGH$^+$09]; indeed, it is possible to get a polynomial time reasoning strategy for the OWL 2 RL profile.

The syntax of OWL 2 RL is asymmetric. From the Definition 2.2.1, we see that a TBox of an ontology is a set of GCIs of the form $C \sqsubseteq D$. By asymmetric, we mean that the syntactic restrictions allowed for subclass expression or LHS of inclusion, i.e., for C are different from those allowed for superclass expressions or RHS of inclusion i.e., for D. For instance, an existential quantification of a class expression (`ObjectSomeValuesFrom`) is allowed only in subclass expressions whereas universal quantification of a class expression (`ObjectAllValuesFrom`) is allowed only in superclass expressions. These restrictions facilitate the design of a rule-based reasoning strategy for OWL 2 RL ontologies. Details are discussed in Subsections 4.2.2, and 4.2.3. Rule-based reasoning strategies are very well-known and have the benefit of the many years of research efforts by the logic-based computing community. There is a number of scalable reasoning strategies for rule-based systems (see subsection 5.3.3).

We translated the OWL 2 RL profile to datalog. Datalog is a simple rule language stemming from the logic programming language Prolog. Datalog is a decidable fragment of first order logic. It is a declarative logic language in which each rule is a function-free Horn clause. A Horn clause is a rule of the form $h \leftarrow b_0, b_1, ...b_k$ where each h (head of the rule) and b_i, $0 \leq i \leq k$, (body of the rule) is an atomic predicate. All variables in each predicate are universally quantified. A datalog program is a collection of Horn rules. In datalog, each variable in the head of a rule must appear in the body of the rule [AHV95].

In [GHVD03], the authors present a mapping, which we refer to as the DLP mapping, between DL-based ontologies and datalog programs and vice versa. We extended the DLP mapping to translate an OWL 2 RL ontology to a set of datalog rules. The goal of that translation is to allow datalog engines to perform DL-based inferencing over

datalog programs. The DLP mapping was based on OWL 1 and so it supports a subset of the DL language fragments $\mathcal{SHIQ}$. We extended the mapping to accommodate all the features of $\mathcal{SROIQ}$-based OWL 2 RL profile. The translation is based on the fact that OWL 1 and OWL 2 are based on DL (see Table 2.3, 2.4). Since DL is a decidable subset of FOL (see Table 2.2), the translation of the OWL 2 RL ontologies to datalog programs results in a decidable subset of FOL. We summarize, the logical relationship among OWL 2 RL, DL, FOL and datalog in Figure 4.1. The following sections explain the translation.

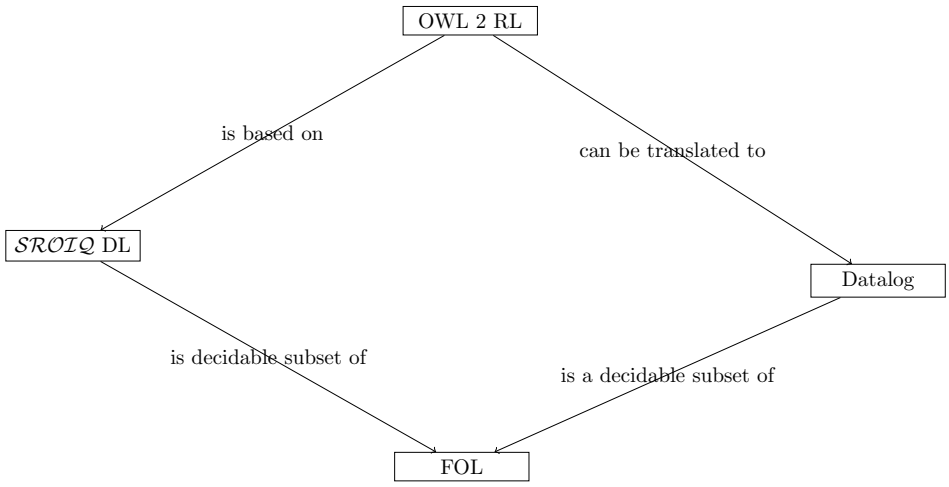

Figure 4.1: The logical relationship among OWL 2 RL, DL, FOL and datalog

4.2 Translation of OWL 2 RL to Datalog

OWL 2 RL describes the domain of an ontology in terms of classes, properties, individuals, as well as data types and values. We recall that individual names refer to elements of the domain, e.g., *Mary*; classes describe sets of individuals having similar characteristics, e.g., *Patient*. Properties describe relationships between pairs of individu-

als, e.g., *isFeeling* describes a relationship between two individuals *Mary* and *MildPain*. Notice that properties may be object or data properties depending on the types of individuals. In OWL 2 RL, object properties can be functional, inverse functional, irreflexive, symmetric, asymmetric, and transitive; however, data properties can only be functional [MGH+09]. Note that qualified cardinality restrictions, irreflexive, symmetric and antisymmetric properties, and property chain inclusion axioms are the new features of OWL 2 RL. The DLP [GHVD03] was restricted to OWL 1 axioms; our translation include the new features of OWL 2 RL.

4.2.1 OWL 2 RL Axioms

An OWL 2 RL axiom is a statement that describes what is true in the associated domain of an ontology. The set of axioms provide information about classes, properties, data ranges, and individuals. We translated OWL 2 RL axioms as follows:

- *Assertions.* OWL 2 RL supports a rich set of axioms about individuals that are often called as facts or assertions. OWL 2 RL has three different types of assertions:
  - the $ClassAssertion$ axiom allows one to state that an individual is an instance of a particular class.
  - the $ObjectPropertyAssertion$ axiom allows one to state that an individual is connected by an object property expression to an individual.
  - the $DataPropertyAssertion$ axiom allows one to state that an individual is connected by an data property expression to a literal.

  The datalog rules for OWL 2 RL axioms are given in Table 4.1. We first show the equivalent expressions for OWL 2 RL Axioms in DL and FOL, then we translate

each axiom into its equivalent datalog rule(s).

OWL 2 RL Constructors	DL Syntax	FOL	Datalog
`ClassAssertions`	a:C	C(a)	C(a)
`ObjectPropertyAssertion`	$\langle a, b\rangle : P$	$P(a, b)$	$P(a, b)$
`DataPropertyAssertion`	$\langle a, b\rangle : P$	$P(a, b)$	$P(a, b)$

Table 4.1: Translation of OWL 2 RL assertions axioms

- *Class Expression Axioms.* OWL 2 RL provides axioms that allow relationships to be established between class expressions. Class expressions are analogous to complex concepts in DL. Note that complex concepts in DL are built by using several constructors with atomic concepts. OWL 2 RL has two types of class expression axioms:

  – a *SubClassOf* axioms allows one to state that each instance of one class expression is also an instance of another class expression. A subclass axiom *SubClassOf(C, D)* states that class expression C is the subclass of the class expression D.

  – an *EquivalentClasses* axiom allows one to state that several class expressions are equivalent to each other. An equivalent class axiom *EquivalentClasses* $(C_1, \ldots, C_n)$ states that all the class expressions C_i, $1 \leq i \leq n$, are equivalent to each other.

  The translation of class expression axioms are given in the Table 4.2.

- *Object Property Axioms.* OWL 2 RL provides axioms that can be used to characterize and establish relationships between object property expressions. It has following object property axioms:

OWL 2 RL Constructors	DL Syntax	FOL	Datalog
`SubClassOf`	$C \sqsubseteq D$	$\forall a(C(a) \rightarrow D(a))$	$D(a) \leftarrow C(a)$
`EquivalentClassOf`	$C \equiv D$	$\forall a(D(a) \leftrightarrow C(a))$	$C(a) \leftarrow D(a)$ $D(a) \leftarrow C(a)$

Table 4.2: Translation of OWL 2 RL class expression axioms

– a *SubObjectPropertyOf* axiom allows one to state that the extension of one object property expression is included in the extension of another object property expression, where the extension of a property is the set of instances that is associated with the property. The *SubObjectPropertyOf* axiom is analogous to the *SubClassOf* axiom. An object property expression in the LHS of a *SubObjectPropertyOf* can be an *ObjectPropertyChain* expression. The *ObjectPropertyChain* expression is known as *roles chain or composite role inclusions* (see $\mathcal{SROIQ}$ in Section 2.2.3). A *SubObjectPropertyOf (ObjectPropertyChain* ($P_1, \ldots P_n$)*, P)* axiom states that if an individual a is connected by a sequence of object property expression $P_1, \ldots P_n$ with an individual b, then a is connected with b by the object property expression P.

– an *EquivalentObjectProperties* axiom allows one to state that the extensions of several object property expressions are the same. The *EquivalentObjectProperties* axiom is analogous to the *EquivalentClasses* axiom.

– an *InverseObjectProperties* axiom can be used to state that two object property expressions are the inverse of each other. An object property expression P is an inverse of an object property expression Q if an individual a is connected by P to an individual b, then b is also connected by Q to a, and vice versa.

- *ObjectPropertyDomain* and *ObjectPropertyRange* axioms can be used to restrict the first and the second individual, respectively, of an object property. For example, the domain and range of an object property *hasPet* are *Person* and *Pet* respectively.
- a *SymmetricObjectProperty* axiom allows one to state that an object property is symmetric. An object property expression P is symmetric if an individual a is connected by P to an individual b, then b is also connected by P to a.
- a *TransitiveObjectProperty* axiom allows one to state that an object property is transitive. An object property expression P is transitive if an individual a is connected by P to an individual b, and b is connected by P to an individual c, then a is also connected by P to c.

- *Data Property Axioms.* OWL 2 also provides axioms that can be used to characterize and establish relationships between data property expressions namely, *SubDataPropertyOf*, *EquivalentDataProperties*, *DataPropertyDomain*, and *DataPropertyRange*. The translation of a data property axioms is the same as that for the analogous object property axioms. Hence, we explain only the translation of object property axioms and give some example translations of data property axioms in the Table 4.6.

The equivalent datalog rules for OWL 2 object property axioms are given in Table 4.3.

4.2.2 Propositional Connectives.

OWL 2 RL also provides a set of propositional connectives to construct new class expressions from existing class expressions. It has following propositional connectives:

OWL 2 RL Constructors	DL Syntax	FOL	Datalog
`SubObjectPropertyOf`	$P \sqsubseteq Q$	$\forall a, b(P(a,b) \rightarrow Q(a,b))$	$Q(a,b) \leftarrow P(a,b)$
`ObjectPropertyChain`	$P \circ Q \sqsubseteq R$	$(\forall a)(\forall b)(\exists c(P(a,c) \wedge Q(c,b)) \rightarrow R(a,b))$	$R(a,b) \leftarrow (P(a,c) \wedge Q(c,b))$
`EquivalentObjectPropertyOf`	$P \equiv Q$	$\forall a, b(Q(a,b) \leftrightarrow P(a,b))$	$P(a,b) \leftarrow Q(a,b)$ $Q(a,b) \leftarrow P(a,b)$
`InverseObjectPropertyOf`	$P \equiv Q^-$	$\forall a, b(P(a,b) \leftrightarrow Q(b,a))$	$P(a,b) \leftarrow Q(b,a)$ $Q(b,a) \leftarrow P(a,b)$
`ObjectPropertyDomain`	$\top \sqsubseteq \forall P^-.C$	$\forall a, b(P(b,a) \rightarrow C(a))$	$C(a) \leftarrow P(a,b)$
`ObjectPropertyRange`	$\top \sqsubseteq \forall P.C$	$\forall a, b(P(a,b) \rightarrow C(b))$	$C(b) \leftarrow P(a,b).$
`SymmetricObjectProperty`	$P \equiv P^-$	$\forall a, b(P(a,b) \rightarrow P(b,a))$	$P(b,a) \leftarrow P(a,b)$
`TransitiveObjectProperty`	$P \circ P \sqsubseteq P$	$\forall a, b, c((P(a,b) \wedge P(b,c)) \rightarrow P(a,c))$	$P(a,c) \leftarrow (P(a,b) \wedge P(b,c))$

Table 4.3: Translation of OWL 2 RL object property axioms

- an *ObjectUnionOf* connective allows one to perform the disjunction operation on class expressions. A union class expression *ObjectUnionOf*$(C_1, \ldots, C_n)$ contains all individuals that are instances of at least one class expression C_i, $1 \leq i \leq n$. As we already mentioned that there are some syntactic restrictions in constructing OWL 2 RL class expressions. The *ObjectUnionOf* is allowed only in LHS of a subclass axiom. The datalog translation of OWL 2 RL class expressions consisting *ObjectUnionOf* connectives are given in the Table 4.4.

  From the table, we see that the datalog rule for the DL statement $C \sqsubseteq D_1 \sqcup D_2$ is $D_1(a) \vee D_2(a) \leftarrow C(a)$. But this is not a valid datalog rule because it has more than one predicate in the head of the rule. Fortunately, disjunction is not allowed in the RHS of a subclass expression in OWL 2 RL. The only valid form of disjunctive class expressions of OWL 2 RL in DL is $C_1 \sqcup C_2 \sqsubseteq D$. So the syntactic restrictions of OWL 2 RL facilitate the translation of OWL 2 RL ontologies into datalog programs.

- an *ObjectIntersectionOf* connective allows one to perform the conjunction operation on class expressions. An intersection class expression *ObjectIntersectionOf(*C_1, $\ldots, C_n$) contains all individuals that are instances of all class expression C_i, $1 \leq i \leq n$. The datalog translation of OWL 2 RL class expressions consisting *ObjectIntersectionOf* connectives are given in the Table 4.4.

OWL 2 RL Constructors	DL Syntax	FOL	Datalog
`ObjectUnionOf`	$C_1 \sqcup C_2 \sqsubseteq D$	$\forall a(C_1(a) \vee C_2(a) \rightarrow D(a))$	$D(a) \leftarrow C_1(a)$ $D(a) \leftarrow C_2(a)$
	$C \sqsubseteq D_1 \sqcup D_2$	$\forall a(C(a) \rightarrow D_1(a) \vee D_2(a))$	$D_1(a) \vee D_2(a) \leftarrow C(a)$
`ObjectIntersectionof`	$C_1 \sqcap C_2 \sqsubseteq D$	$\forall a(C_1(a) \wedge C_2(a) \rightarrow D(a))$	$D(a) \leftarrow C_1(a) \wedge C_2(a)$
	$C \sqsubseteq D_1 \sqcap D_2$	$\forall a(C(a) \rightarrow D_1(a) \wedge D_2(a))$	$D_1(a) \leftarrow C(a)$ $D_2(a) \leftarrow C(a)$

Table 4.4: Translation of OWL 2 RL class expressions consisting propositional connectives

4.2.3 Property Restrictions.

We can build up OWL 2 class expressions by placing restrictions on property expressions. OWL 2 RL has both existential and universal quantification restrictions for object property and data property expressions. Here we explain only the translation of object property restrictions.

- An *ObjectAllValuesFrom* class expression allows for universal quantification over an object property expression, and it contains those individuals that are connected through an object property expression only to instances of a given class expression. The datalog translation of OWL 2 RL class expression consisting *ObjectAllValuesFrom* restrictions is given in the Table 4.5.

- An *ObjectSomeValuesFrom* class expression allows for existential quantification over an object property expression, and it contains those individuals that are connected through an object property expression to at least one instance of a given class expression. The datalog translation of OWL 2 RL class expression consisting *ObjectSomeValuesFrom* restrictions is given in the Table 4.5.

OWL 2 RL Constructors	DL Syntax	FOL	Datalog
`ObjectAllValuesFrom`	$C \sqsubseteq \forall P.D$	$\forall a(C(a) \rightarrow (\forall b\ (P(a,b) \rightarrow D(b))))$	$D(b) \leftarrow C(a) \wedge P(a,b)$
`ObjectSomeValuesFrom`	$\exists P.C \sqsubseteq D$	$\forall a(\exists b\ (P(a,b) \wedge C(b)) \rightarrow D(a))$	$D(a) \leftarrow P(a,b) \wedge C(b)$

Table 4.5: Translation of OWL 2 RL class expressions consisting property restrictions

We give example translations OWL 2 RL axioms presented in Manchester Syntax to datalog rules in Table 4.6. In this table, we express some real-world concepts in OWL 2 RL axioms and translate each axiom to its equivalent datalog rule(s).

4.2.4 Expressive Restriction

The limitation of the DLP mapping is that axioms about cardinality restrictions and property restrictions - namely, functional, inverse functional, irreflexive, asymmetric - cannot be translated into datalog rules to perform reasoning. These axioms impose certain restrictions over the properties of an ontology and any violation of these restrictions results an inconsistent ABox. We developed a restriction checker to check ABox consistency for these axioms. In order to develop the restrictions checker, we represent each restriction using a datalog head (i.e., rules without bodies) and then, each datalog head is translated to an equivalent SQL statement to store the property and its restrictions to the relational database. The translation of datalog rules to SQL statements is discussed in Chapter 5. For each property assertion, the restrictions checker queries the

relational database to identify any violation, and if there is no violation, then it allows the insertion of the data into the relational database. So the restrictions checker keeps our ABox alway consistent. We discuss the different types of restrictions violations as follows:

- *Minimum Cardinality Violation.* A minimum cardinality expression *ObjectMinCardinality(n OPE CE)* consists of a nonnegative integer n, an object property expression OPE, and a class expression CE, and it contains all those individuals that are connected by OPE to at least n different individuals that are instances of CE. For example, an *ObjectMinCardinality(2 fatherOf Person)* contains those individual that are connected by *fatherOf* to at least two different instances of *Person*. The restrictions checker counts the number of instances of CE that are connected by OPE by a database query. If the number is less than n, then the ABox is inconsistent. The datalog rule is *ObjectMinCardinality(n OPE CE)*.

- *Maximum Cardinality Violation.* A maximum cardinality expression *ObjectMaxCardinality(n OPE CE)* consists of a nonnegative integer n, an object property expression OPE, and a class expression CE, and it contains all those individuals that are connected by OPE to at most n different individuals that are instances of CE and CE is optional. For example, an *ObjectMaxCardinality(2 hasDog Person)* contains those instances that are connected by *hasDog* to at most two instances of *Person*. Again the restrictions checker identifies the consistency of the ABox by determining the number of instances of CE. The datalog rule is *ObjectMaxCardinality(n OPE CE)*.

- *Functional Characteristics Violation.* An object property functionality axiom

FunctionalObjectProperty (OPE) states that for each individual a, there can be at most one distinct individual b such that a is connected by OPE to b. For example, the *FunctionalObjectProperty(hasMother)* states that each object can have at most one mother. An ontology may be inconsistent if a functional object property points to two different values. The datalog rules for materializing this type of property is *FunctionalObjectProperty (OPE)*. In OWL 2 RL, a data property can also be functional and the datalog for the functional data property axiom is *FunctionalDataProperty (DPE)*.

- *Inverse Functional Characteristics Violation.* An object property inverse functionality axiom *InverseFunctionalObjectProperty (OPE)* states for each individual a, there can be at most one individual b such that b is connected by OPE with a. For example, the *InverseFunctionalObjectProperty(motherOf)* states that each object can have at most one mother. An ontology containing the expression OPE may be inconsistent if there are two individuals connected to by OPE. The translation is *InverseFunctionalObjectProperty (OPE)*.

- *Irreflexive Characteristics Violation.* An object property irreflexivity axiom *IrreflexiveObjectProperty(OPE)* states that no individual is connected by OPE to itself. For example, a concept *nobody can be married to themselves* can be expressed as an irreflexive object property *marriedTo*. If we assert an ABox axiom *marriedTo(Bob, Bob)*, then the ontology will be inconsistent. The datalog rule for this axiom is *IrreflexiveObjectProperty (OPE)*.

- *Asymmatric Characteristics Violation.* An object property asymmetry axiom *AsymmetricObjectProperty(OPE)* states that if an individual a is connected by

OPE to an individual b, then b cannot be connected by OPE to a. Consider the ontology consisting of the following axioms:

- *parentOf* is an asymmetric object property: AsymmetricObjectProperty (parentOf)
- Peter is a parent of Bob: ObjectPropertyAssertion(parentOf Peter Bob)

If we assert an axiom *ObjectPropertyAssertion(parentOf Bob Peter)*, it will invalidate the asymmetric object property and the ontology would become inconsistent. The datalog rule is *AsymmetricObjectProperty(OPE)*.

Table 4.6: Example Translation: OWL 2 RL axioms to datalog rules.

OWL 2 RL Axioms in Manchester Syntax	**Datalog**
Individual: Peter *Type* Man (Peter is a Man)	Man(Peter)
Individual: Brain *Facts hasDog* Peter (ObjectProperty Assertion: Brain is a Dog of Peter)	hasDog(Peter,Brain)
Individual: 17 *Facts hasAge* Meg (DataProperty Assertion: Meg is seventeen years old)	hasAge(Meg,17)
Class: Child *SubClassOf* Person (Each child is a person.)	Person(a) ← Child(a)
Equivalent Classes: USPresident, PrincipalResidentOfWhiteHouse (USPresident and PrincipalResidentOfWhiteHouse both are identical concept)	USPresident(a) ← PrincipalResidentOfWhiteHouse(a) PrincipalResidentOfWhiteHouse(a) ← USPresident(a)
Continued on next page	

Table 4.6 – continued from previous page

OWL 2 RL Axioms in Manchester Syntax	Datalog
ObjectProperty: hasBiologicalFather *SubPropertyOf* hasFather (Having a biological father implies having a father.)	hasFather(a,b) ← hasBiologicalFather(a,b)
SubClassOf(ObjectPropertyChain(hasFather, hasBrother), hasUncle) (The brother of someone's father is that person's uncle.)	hasUncle(a,c) ← hasFather(a,b) ∧ hasBrother(b,c)
DataProperty: hasLastname *SubPropertyOf* hasName (A last name of someone is his/her name as well.)	hasName(a,b) ← hasLastname(a,b)
ObjectProperty: hasDog *Domain* Dog (Only people can own dogs.) *ObjectProperty:* hasDog *Range* (The range of the hasDog property is the class Dog.)	Person(a) ← hasDog(a,b) Dog(b) ← hasDog(a,b)
DataProperty: hasName *Domain* Person (Only people can have names.) *DataProperty:* hasName *Range* string (The range of the hasName property is string.)	Person(a) ← hasName(a,b) string(b) ← hasName(a,b)
Equivalent Properties: hasBrother, hasMaleSibling (Having a brother is the same as having a male sibling.)	hasBrother(a,b) ← hasMaleSibling(a,b) hasMaleSibling(a,b) ← hasBrother(a,b)
Equivalent Properties: hasName, seLlama (hasName and seLlama (in Spanish) are synonyms.)	hasName(a,b) ← seLlama(a,b) seLlama(a,b) ← hasName(a,b)

Continued on next page

Table 4.6 – continued from previous page

OWL 2 RL Axioms in Manchester Syntax	Datalog
ObjectProperty: hasFather *InverseOf* fatherOf (Having a father is the opposite of being a father of someone.)	fatherOf(b,a) ← hasFather(a,b)
ObjectProperty: friendOf *Characteristics* Symmetric (If a is a friend of b, then b is a friend of a.)	friendOf(b,a) ← friendOf(a,b)
ObjectProperty: ancestorOf *Characteristics* Transitive (If a is an ancestor of b and b is an ancestor of c, then a is an ancestor of c.)	ancestorOf(a,c) ← ancestorOf(a,b) ∧ ancestorOf(b,c)
ObjectProperty: *ObjectUnionOf:*(Mother *or* Father) *SubClassOf* Parent (Parents are either Mather or Women)	Parent(a) ← Mother(a) Parent(a) ← Father(a)
ObjectProperty: Mother *SubClassOf* *ObjectIntersectionOf* Woman, Parent (Mothers are exactly those who are women and parents)	Woman(a)← Mother(a) Parent(a) ← Mother(a)
Class: HappyPerson *SubClassOf* hasChild *only* Happy (A happy person is one whose all children are happy.) ($HappyPerson \sqsubseteq \forall\ hasChild.Happy$)	Happy(b) ← HappyPerson(a) ∧ hasChild(a,b)
ObjectProperty: hasChild *some* Person *SubClassOf* Parent (Those who have at least one child which is a Person are Parents.) ($\exists\ hasChild.Person \sqsubseteq Parent$)	Parent(a) ← hasChild(a,b) ∧ Person(b)

4.2.5 Implementing the Translation

First, we take an OWL 2 RL ontology as an input and then invoke a DL reasoner to compute a complete class hierarchy. Then, we parse the inferred ontology that generates a set of OWL 2 RL axioms and facts. After that, we translate the ontology by recursively applying the translation rule for each axiom and facts of an ontology. Let $\mathcal{T}$ be a translation function that takes an ontology $\mathcal{O}$, a set of OWL 2 RL axioms and fact. It applies the appropriate translation rules recursively for each axiom or fact of $\mathcal{O}$ and generates a datalog program. For example, $\mathcal{T}$ would translate the class expression $Disease \sqcap \exists affects.Heart \sqsubseteq HeartDisease$ into the datalog rule $HeartDisease(x) \leftarrow Disease(x) \wedge affects(x, y) \wedge Heart(x)$.

4.3 Inferencing

In this section, we show the representation of inference tasks of OWL 2 RL in terms of datalog queries from [GHVD03]. In our approach, we use a DL reasoner for inference tasks related to TBox, therefore we only discuss the reduction of inference tasks related to Abox into inference tasks in a datalog program.

In a DL reasoning system, several different kinds of query are typically supported with respect to a knowledge base $\mathcal{K}$. Note that $\mathcal{K}$ is representing an OWL 2 RL ontology. These include queries about classes:

- Class-instance membership queries: given a class C,
  - ground: determine whether a given individual a is an instance of C;
  - open: determine all the individuals in $\mathcal{K}$ that are instances of C;

  - all-classes: given an individual a, determine all the (named) classes in $\mathcal{K}$ that a is an instance of;

- Consistency checking: class satisfiability queries, i.e., given a class C, determine if C is satisfiable (consistent) with respect to $\mathcal{K}$.

In addition, there are similar queries about properties: property instance membership, property subsumption, property hierarchy, and property satisfiability. We assume that $\mathcal{D}$ is a datalog program derived from an OWL 2 RL ontology via translation $\mathcal{T}$, and that all ontology-based queries are with respect to $\mathcal{K}$. In logic programming-based reasoning engines, there is one basic kind of query supported with respect to a datalog program $\mathcal{D}$: datalog queries. These queries are either:

- ground: determine whether a ground atom A is entailed;

- open: determine, given an atom A (in which variables may appear), all the tuples of variable bindings (substitutions) for which the atom is entailed.

We use a DL reasoner for generating a complete class hierarchy (see Section 4) and checking TBox consistency. Queries related to ABox are reduced as follows:

Datalog queries (ground or open) can be used to answer ontology-based (ground or open) class-instance membership queries. For instance, a is an instance of C *iff* $\mathcal{D}$ (the datalog program) entails $\mathcal{T}(a : C)$. When C is an atomic class name, the mapping leads directly to a datalog query. When C is a conjunction, the result is a conjunction of datalog queries; i.e., a is an instance of $C \sqcap D$ *iff* $\mathcal{D}$ entails $\mathcal{T}(a : C)$ and $\mathcal{D}$ entails $\mathcal{T}(a : D)$.

When C is a universal restriction, the mapping $\mathcal{T}(a : \forall P : C)$ gives $\mathcal{T}(C, y) \leftarrow P(a, y)$. This can be transformed into a datalog query using a simple kind of substitution;

i.e., b is replaced with a constant b, where b is new in $\mathcal{D}$, and a is an instance of $\forall P.C$ *iff* $\mathcal{D} \cup \{P(a,b)\}$ entails $\mathcal{T}(b,C)$.

The case of property-instance membership queries is trivial as all properties are atomic: $< a,b >$ is an instance of P *iff* $\mathcal{D}$ entails $P(a,b)$. Complete information about class-instance relationships, to answer open or all-classes class-instance queries, can then be obtained via class-instance queries about all possible combinations of individuals and classes in $\mathcal{K}$.

Chapter 5

Materialization

This chapter discusses a method of storing inferred information from ontologies in relational databases, discusses the structure of the relational database required to store the information, and outlines a translation from datalog rules into SQL statements.

5.1 Motivation

Materialization is an approach of storing inferred information from ontologies, which is maintained when the TBox or the ABox is updated. If the ABox of an ontology is large and the query rate is high, the materialization technique is better than the approaches that perform reasoning during query evaluation. Materialization techniques are used in many scalable reasoners, including [AJPS10], [Mot08] and [LMZ+07].

Materialization can be used to increase the performance of query evaluations by making implicit information explicit. This avoids the need to recompute derived information for every query. In our approach, materialized information is stored in relational databases. Relational database management systems (RDBMS) are tailored to the pro-

cessing of large amounts of data, and the efficient manipulation of such data. As such they appear to be well-suited for implementing materialization on a persistent storage system.

In section 5.2, first we discuss the relationship between datalog and SQL and then we describe a database structure required to store OWL 2 RL ontologies in a relational database. In section 5.3, we focus on the materialization process, first we discuss the structure of our datalog program, then we explain how a datalog rule can be translated to a corresponding SQL statement. Finally, we discuss the inferencing over datalog program.

5.2 Datalog and SQL

Datalog rules are closely related to operations in relational algebra, and the foundation of SQL is also relational algebra [AHV95]. Analogies between Datalog and relational query languages such as SQL are well known and well studied. Both formalisms cover unions of conjunctive queries, but additionally datalog supports recursions. Datalog programs can be implemented on top of relational databases. In order to store the inferred information into databases, we use an optimal database schema developed in [LMZ$^+$07]. .

The tables of the database schema are categorized into 4 types: atomic tables, TBox axiom tables, ABox fact tables, and class constructor tables. The atomic tables include: Ontology, PrimitiveClass, Property, Datatype, Individual, Literal, and Resource. The Property table stores characteristics of the properties such as symmetry, transitivity, functionality, etc. There are three different type of assertions to populate ABoxes namely *ClassAssertions*, *ObjectPropertyAssertion*, and *DataPropetyAssertion*. Class,

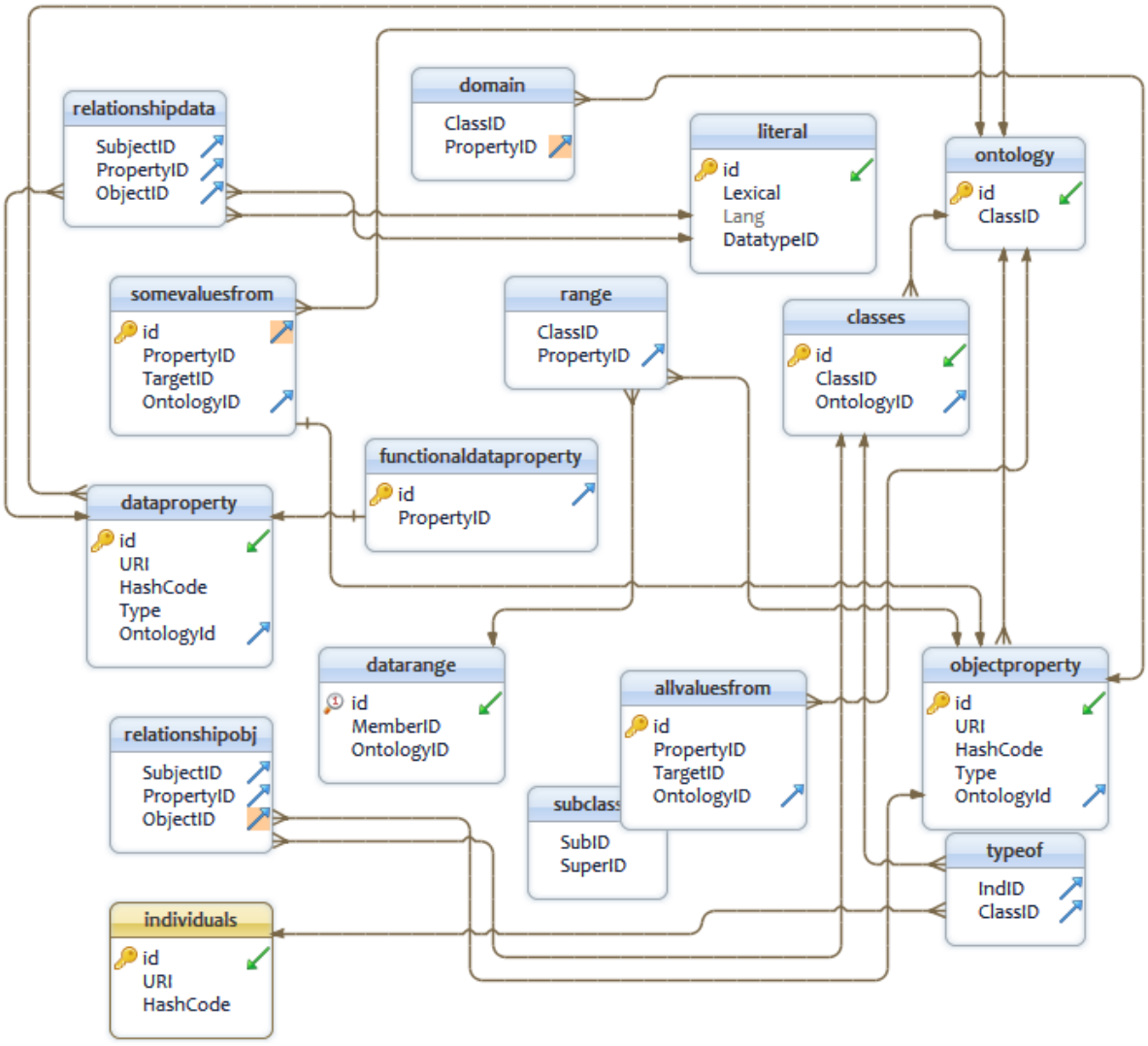

Figure 5.1: A fragment of the database schema

object property and data property assertions are stored in *TypeOf*, *RelationshipObj*, and *RelationshipData* tables respectively. We used class constructor tables to store class expressions and other relational tables are listed in Subsection 5.3.2. In total for an OWL 2 RL ontology, 33 relational tables are required to store it in the database. A fragment of the database structure is given in Figure 5.1 where some tables including

their column names are shown. An arrow between two tables represent a referential constraint (functional dependency) between the tables. Referential constraints are also known as foreign keys.

5.3 Inferencing and Storing information

We translate an OWL 2 RL ontology into a datalog program. We need a formal representation of the datalog version of the translated ontology in order to translate it to SQL statements. In this section, first we give an abstract syntax for datalog programs, then we describe the translation of datalog rules to SQL statements.

5.3.1 An abstract syntax for datalog programs

The abstract syntax for our datalog program is given in Listing 5.1 using a BNF. In this notation, the terminals are quoted, the non-terminals are not quoted, alternatives are separated by vertical bars, and components that can occur zero or more times are enclosed by braces followed by a superscript asterisk symbol ($\{\ldots\}^*$). A class atom represented by *class(i-object)* in the BNF consists of a class and a single argument representing an individual. For example, an atom *Person(x)* holds if x is an instance of the class *Person*. Similarly, an individual property atom represented by *ObjectProperty(i-object,i-object)* consists of an object property and two arguments representing individuals. For example, an atom *hasDog(x,y)* holds if x is related to y by property *hasDog*. Recall that in OWL 2 RL, object properties can be functional, inverse functional, irreflexive, symmetric, asymmetric, and transitive, and data properties can be only functional [MGH+09]. The characteristics of properties are encoded

as ground facts in datalog. For example, a functional object property *hasMother* is encoded as *FunctionalObjectProperty (hasMother)*. We define some atoms as restricted atoms which are basically ground facts; i.e., there are no variables in their argument list. These atoms are restricted to appear only in the head of datalog rules.

```
Program ::=Rule {Rule}*
Rule ::= Head | Head '←' Body
Head ::= Atom | RestrictedAtom
Body ::= Atom{∧ Atom}*
Atom ::= Class '(' i-object ')'
       | ObjectProperty '(' i-object ',' i-object ')'
       | DataProperty '(' i-object ',' d-object ')'

RestrictedAtom ::= 'InverseObjectProperty(' PropertyID ',' PropertyID')'
                 | 'FunctionalObjectProperty(' PropertyID ')'
                 | 'InverseFunctionalObjectProperty(' PropertyID ')'
                 | 'SymmetricObjectProperty(' PropertyID ')'
                 | 'AsymmetricObjectProperty(' PropertyID ')'
                 | 'TransitiveObjectProperty(' PropertyID ')'
                 | 'IrreflexiveObjectProperty(' PropertyID ')'
                 | 'FunctionalDataProperty(' PropertyID ')'
                 | 'ObjectMinCardinality(' n PropertyID ClassID ')'
                 | 'ObjectMaxCardinality(' n PropertyID ClassID')'
                 | 'ObjectPropertyDomain(' ClassID ')'
                 | 'ObjectPropertyRange(' ClassID ')'
                 | 'DataPropertyDomain(' ClassID ')'
                 | 'DataPropertyRange(' ClassID ')'

i-object ::= i-variable | individualID
d-object ::= d-variable | dataLiteral
i-variable ::= 'I-variable(' URIreference ')'
d-variable ::= 'D-variable(' URIreference ')'
```

Listing 5.1: Abstract syntax for datalog programs

5.3.2 Datalog to SQL translation

We translate each datalog rule to an SQL statement and the translated SQL statements are used to store the information to relational databases. There are two different mapping approaches to translate datalog rules to SQL statements, namely, direct mapping and meta mapping [WLS03]. In the direct mapping approach, each n-ary datalog pred-

icate $R(X_1, \ldots, X_n)$ corresponds to a relational table named R consisting n columns; the argument which appears in the i^{th} place in the predicate becomes the entity in the i^{th} column of R. For instance, if we translate a datalog fact *Person(Bob)*, then *Bob* will be added as a row of the *Person* table. Similarly, if we translate a datalog fact *hasBrother(Mary, John)*, then *(Mary, John)* will be added as a row of the *hasBrother* table.

The direct mapping approach is not suitable when the size of the ontology becomes large because it generates a single table for each predicate, so the number of tables linearly increases with the number of classes and properties of ontologies. The meta mapping approach is more efficient than the direct mapping approach. In the meta-mapping approach, instead of creating a table for each predicate, three tables are created, namely,

- *TypeOf(Class, Individual)* - stores the information of *all classes* of an ontology
- *RelationshipObj(Subject, Predicate, Object)* - stores the information of all object properties of an ontology
- *RelationshipData(Subject, Predicate, Object)* - stores the information of all data properties of an ontology

For instance, if we translate a datalog fact *Person(Bob)*, then *(Person,Bob)* will be added as a row of the *TypeOf(Class, Individual)* table. Similarly, if we translate a datalog fact *hasBrother(Mary, John)*, then *(Mary, hasBrother, John)* will be added as a row of the *RelationshipObj(Subject, Predicate, Object)* table.

Another advantage of using the meta-mapping approach is to store some useful information that is necessary for reasoning. For example, if we materialize a rule

hasBrother $\sqsubseteq$ *hasSibling*, then in the direct mapping approach, we only store all the instances of *hasBrother* and the instances of *hasSibling*, but the information that *hasBrother* is a sub object property of *hasSibling* is not stored. But in the case of the meta-mapping approach, the sub object property relations information is also stored. For this approach, we create additional relational tables SubClassOf, SubPropertyOf, Functional, Symmetric, Asymmetric, Irreflexive, Transitive, Domain, and Range, SomeValuesFrom, AllValuesFrom, Classes, Individuals, DisjointClass, InversePropertyOf, EquivalentClasses, and EquivalentProperties in order to store the respective information of OWL 2 RL axioms.

We used the meta-mapping approach to generate SQL statements from datalog rules. A datalog program is a set of ground facts (i.e., rules without bodies) and rules. Each predicate in a datalog rule can be either a class assertion e.g., Person(Bob), or a object property assertion e.g, hasMother(Bob, Jenny), or a functional object property e.g., FunctionalObjectProperty(hasMother), etc (See the abstract syntax of our datalog program from the Listing 5.1). The equivalent SQL statements for ground facts are SQL INSERT statements to insert the facts into corresponding tables. The appropriate table names are selected based on the type of predicate. For example, if a predicate is a class assertion, then the *TypeOf(Class, Individual)* table is selected, if a predicate is a data property assertion, then the *RelationshipData(Subject, Predicate, Object)* is selected. For instance, the SQL statement for the ground fact *Person(Bob)* is

```
INSERT INTO TypeOf(Class, Individual) VALUES(‘‘Person’’, ‘‘Bob’’)
```

The SQL statement for the ground fact hasMother(Bob, Jenny) is

```
INSERT INTO RelationshipObj(Subject, Predicate, Object)
            VALUES(‘‘Bob’’, ‘‘hasMother’’, ‘‘Jenny’’)
```

A datalog rule has one of the following forms

$$head(h_1, \ldots h_n) \leftarrow body(b_1, \ldots, b_n) \tag{5.1}$$

$$head(h_1, \ldots h_n) \leftarrow body_0(b_1, \ldots, b_n) \wedge \ldots \wedge body_n(b_1, \ldots, b_n) \tag{5.2}$$

The rule format in 5.1 has only one body predicate and such a rule corresponds to an SQL SELECT statement of the following form:

```
INSERT INTO <Table1> SELECT <Projectors> FROM <Tables> WHERE <SELECTORS>
```

The rule format in 5.2 has more than one body predicate. We need an additional database operation namely Join to formulate SQL statements for such a rule. In a relational database, the Join operation is used to match one table against another based on some condition, thereby creating a third table with data from the matching tables. For example, a customer table can be joined with an order table creating a table for all customers who purchased a particular product. The general form of SQL SELECT statements for rules of the form 5.2 is:

```
INSERT INTO <Table1> SELECT <Projectors> FROM <Table2> JOIN ... JOIN
                                          <TableN> WHERE <SELECTORS>
```

We need explicitly bound variables in the head predicate of each rule to fill the "Projectors" list. A variable occurring in the head predicate is explicitly bound if at least one body predicate contains the variable as an element. Note that all the variables in the head predicate of our datalog program are explicitly bound (see subsection 4.1). We filled the projector list of SQL statements by iterating over all body predicates and extracting the predicates explicitly binding at least one head variable. The names of these predicates are used to select appropriate database tables.

Let us consider following two rules of the forms (5.1) and (5.2):

$$Person(a) \leftarrow Child(a) \tag{5.3}$$

$$Dog(b) \leftarrow hasPet(a, b) \wedge Pet(b) \tag{5.4}$$

The SQL statement for the first rule ((5.3)) is

```
INSERT INTO TypeOf(Class, Individual) SELECT ‘‘Person’’, Ind
                    FROM TypeOf WHERE Class == ‘‘Child’’
```

Listing 5.2: The SQL statement for the datalog rule (5.3)

This SQL statement (Listing (5.2)), first selects the individuals(instances) of the "Child" class from the table "TypeOf(Class, Individual)", then adds all selected individuals as individuals of the "Person" class to the same table.

The SQL statement for the second rule ((5.4)) is

```
INSERT INTO TypeOf(Class, Individual) SELECT ‘‘Dog’’, r.Object
            FROM TypeOf t, RelationshipObj r WHERE r.Predicate == ‘‘hasPet’’
                        AND t.Class == ‘‘Pet’’ AND t.Individual == r.Object
```

Listing 5.3: The SQL statement for the datalog rule (5.4)

This SQL statement (Listing (5.3)), first selects the individuals (instances) of the "Pet" class by joining two tables "TypeOf(Class, Individual)" and "RelationshipObj(Subject, Predicate, Object)", then adds all selected individuals as individuals of the "Dog" class to the same table.

5.3.3 Inferencing

We discussed in subsection 4.3 how can we reduce a DL query to a datalog query, allowing us to perform inferencing over a datalog program instead of a DL knowledge base. In this chapter, we also discussed the analogy between datalog and relational databases and the construction of SQL statements from corresponding datalog rules.

Algorithm 1: Materialize($\mathcal{R}$, $\mathcal{F}$) - materialize a datalog program into the database

Data: $\mathcal{R}$- set of datalog rules, $\mathcal{F}$- set of ABox facts (ground fact/rules without body)..
Result: $\mathcal{S}$- set of SQL statements.

```
repeat
    inferred ⇐ false
    for ∀ r ∈ R do
        for ∀ f ∈ Consequences(r, F) do
            if f ∉ F then
                inferred ⇐ true
                F ⇐ f
                S ⇐ datalogToSQL(f)
            end
        end
    end
until ! inferred
for ∀ r ∈ R do
    S ⇐ datalogToSQL(r)
end
for ∀ s ∈ S do
    executeSQL(s)
end
```

In this subsection, we show how to infer implicit knowledge and store it in a database. After the translation of OWL 2 RL ontologies, we get a rule base - a datalog program with a set of datalog rules. There are two principle strategies for rule-based inferences:

- *Forward-chaining*. The basic idea in forward chaining is looking for unifications that will match each of the formulae in the body of a datalog rule to datalog facts, and then inferring the head of the rule under the specified substitution. After the inference, new facts are added to the knowledge base. The process is continued until there is nothing new to be derived.

- *Backward-chaining*. Starting with a particular fact or a query that we want to prove, we look for derivations that allow us to prove the goal. For example, if we look for a rule R or a fact F whose head unifies with the goal, we then recursively

try to prove all the formulae in the body of R.

Procedure Consequences($\mathcal{R}$, $\mathcal{F}$) - recursively applied for all the predicates of a rule body to derive the consequence or the value of the head of that rule.

Data: $\mathcal{R}$- set of datalog rules, $\mathcal{F}$- set of Abox facts.

```
1  if R is a fact then
2  |  return R
3  end
4  inferred ⇐ ∅
5  for ∀ f ∈ F do
6  |  Instantiate the rule R
7  |  if R is a fact then
8  |  |  inferred := inferred ∪ R
9  |  end
10 end
11 return inferred
```

We use the inferencing strategy known as materialization [Bro05]. In materialization, for each class/property assertion a forward-chaining is performed. The inferred knowledge is stored in relational databases for query evaluation or retrieval. So for each assertion, we select the number of rules to be fired by matching the fact with the body of each rule from the rulebase (i.e., the datalog program). Here, firing a rule means an execution of the corresponding SQL statements for those datalog rules. We developed a materialization algorithm which uses forward chaining. The pseudocode for the materialization algorithm is given in Algorithm 1. The $Materialize(\mathcal{R}, \mathcal{F})$ algorithm takes the datalog version of the OWL 2 RL ontology, then performs a forward chaining for inferring implicit knowledge. Then $Materialize(\mathcal{R}, \mathcal{F})$ algorithm invokes the procedure $Consequences(\mathcal{R}, \mathcal{F})$ (Procedure Consequences) to perform forward-chaining reasoning. The $Consequences(\mathcal{R}, \mathcal{F})$ procedure recursively applies all the values to the predicates of the body of a selected rule to derive the value of the head of that rule. The Algorithm 1 (Line 1 - 12), first determines all the consequences using the $Consequences(\mathcal{R}, \mathcal{F})$ pro-

cedure and resulting an ABox with asserted and inferred facts, and then (Line 13 -15), it translates each datalog rule to a corresponding SQL statement. After that using the $executeSQL(S)$ method (Line 16 - 18), it executes each SQL statement to store the information in relational databases. The Procedure Consequences is invoked by the Algorithm 1 for each rule. The Procedure Consequences (Line 1 -3) returns the datalog rule if it is a fact, otherwise (Line 4 - 11) it tries to instantiate the rule to derive the value of the head of that rule (i.e., determines the consequences of that rule using ABox facts). If the body of a rule is fully instantiated by the facts, then the value of the head is added as an inferred fact. In this way, our materialization algorithm performs a total forward chaining, and rule-base inferencing by translating each datalog rule to a SQL statement, and finally stores the information in a relational database.

Chapter 6

Query Processing

In previous chapters, we introduced ontologies as knowledge representation systems and also described how implicit knowledge can be extracted and materialized in a persistence system. However, a query language is necessary to access this knowledge. In this chapter, we describe a query interface for our reasoning system that we developed by modifying an existing SPARQL-DL API. We also added a new feature, namely, aggregation and grouping, to our query interface from the newly proposed standard for SPARQL query, SPARQL 1.1 to our query interface.

6.1 Motivation

SPARQL [PS08] is a W3C recommendation for querying RDF graphs. There are two major limitations of SPARQL that prevent us from using it directly as a query language for OWL 2. First, it is based on the triple patterns of RDF graphs. The RDF patterns do not match the well-defined OWL 2 syntax, so a modified version of SPARQL is necessary. Second, in our framework, we materialized ontologies into relational databases. So it

is necessary to translate each SPARQL query into a SQL query to retrieve data from relational databases.

For relational data, SQL is by far the most widely supported query language; it includes support for large data-storage, efficient indexing schemes, and query optimization. It would therefore be attractive if we could use SQL, a robust and widely available technology, for extracting knowledge from materialized ontologies. Unfortunately, this can only be done at the cost of learning the underlying relational schemas used for materializing ontologies. Moreover, it is essential for more real-world semantic web-based applications to extract data from relational sources along with ontologies. Therefore, a uniform query language is necessary for accessing both structured data (e.g., relational databases) and semi structured data (e.g., RDF, OWL).

In order to use SPARQL for querying ontologies based on OWL 1, Sirin and Parsia [SP07] designed a query language called SPARQL-DL, a substantial subset of SPARQL, by mapping RDF triple patterns using OWL 1 DL semantics. Therefore, SPARQL-DL supports only the semantics of OWL 1 ontologies. A query language for OWL 2 ontologies was developed by [SPA11] - we will refer it as $SPARQL - DL_E$. The SPARQL-DL API supports $SPARQL - DL_E$ queries over OWL 2 RL ontologies. However, the SPARQL-DL API is built to interface with main-memory-based OWL 2 reasoners, we need some modifications to integrate with our system. We modified the SPARQL-DL API to support queries over relational database-based reasoners. In Section 6.2, we discuss the syntax and semantics of the query language for OWL 2 RL, in Section 6.3, we discuss the modifications of SPARQL-DL API that we did to integrate it with our reasoning systems and in the last section of this chapter, we discuss the extension of $SPARQL - DL_E$.

6.2 The Query Language for OWL 2 RL

6.2.1 The SPARQL Query Language

SPARQL, a W3C recommendation, is a declarative query language primarily designed for extracting information from RDF graphs. To define the syntax and semantics of SPARQL, we follow the notation from [SML10]. Assume there are three pairwise disjointed infinite sets B(blank nodes), L(Literals), and U(URIs). Literals are quoted strings, e.g., "Bob", blank nodes represent anonymous classes/properties, and uniform resource identifiers (URI) are used to uniquely represent elements of ontologies. An RDF graph is defined as follows:

Definition 6.2.1. *(RDF Graph). An RDF triple* $(s, p, o) \in (B \cup I) \times I \times (I \cup B \cup L)$ *represents a link in a directed graph where subject* $\boldsymbol{s}$ *connects through predicate* $\boldsymbol{p}$ *to object* $\boldsymbol{o}$*. An RDF graph is a set of RDF triples.*

For example, a simple a graph pattern to represent the name of a professor of a university is

$$(< \text{http://logic.stfx.ca/StFX} >, < \text{http://logic.stfx.ca/Professor} >, \text{“Bob”}) \qquad (6.1)$$

where the subject is "StFX" and the predicate is "Professor" and the object is "Bob". Note that prefixes are used with subjects and predicates to represent the elements of ontologies in the URI format. In this example, we use "http://logic.stfx.ca/StFX" as a prefix for both subject and predicate.

We now introduce an abstract syntax for SPARQL expressions. Let V be a set of variables disjoint from $(B \cup L \cup U)$. We distinguish variable names by a leading question mark symbol, e.g., `?title`.

Definition 6.2.2. *(SPARQL Expression). A SPARQL expression is an expression that is built recursively as follows: (i) A triple pattern* $t \in (U \cup V) \times (U \cup V) \times (L \cup U \cup V)$ *is an expression; (ii) If* Q_1 *and* Q_2 *are expressions,* $(Q_1\ UNION\ Q_2)$,$(Q_1\ AND\ Q_2)$ *are expressions where* $UNION$ *and* AND *are binary operators.*

SPARQL expressions constitute SPARQL queries. SPARQL has four query forms, namely SELECT, ASK, CONSTRUCT, and DESCRIBE. A SELECT query returns a set of query results, while an ASK query is a boolean query that return true **iff** there is one or more result, and false otherwise. A CONSTRUCT query extracts the set of query results and returns the result as an RDF graph while a DESCRIBE query returns the description of the result in the form of an RDF graphs. We need only SELECT and ASK queries to extract knowledge from materialized ontologies. Hence, we restrict our discussion to SPARQL SELECT and ASK queries.

Definition 6.2.3. *(SELECT Query, ASK Query). Let* Q *be a SPARQL expression and let* $S \subseteq V$ *be a finite set of variables. A SPARQL SELECT query is an expression of the form* $SELECT_S(Q)$*. A SPARQL ASK query is an expression of the form* $ASK(Q)$*.*

A simple SELECT query to find the name of the professors from the graph pattern described in 6.1 is

```
SELECT  ?name
  WHERE {
          <http://logic.stfx.ca/StFX>, <http://logic.stfx.ca/Professor>, ?name
        }
```

This query, on the data in 6.1, has one solution:

Name
Bob

Ask queries return boolean results i.e., true or false. A simple ASK query to check whether "Bob" is a name of a professor or not can be written as

```
ASK {
      ?name <http://logic.stfx.ca/Professor> ''Bob''
      }
```

6.2.2 The semantics of $SPARQL - DL_E$

$SPARQL - DL_E$ is a query language for OWL 2 ontologies. It is an expressive query language that can combine TBox and ABox queries. Here we briefly describe the semantics of $SPARQL - DL_E$ adapted from [SP07].

Let $\mathcal{O}$ be an OWL 2 ontology, $V_O = (\mathcal{V}_{cls}, \mathcal{V}_{op}, \mathcal{V}_{dp}, \mathcal{V}_{ind}, \mathcal{V}_D, \mathcal{V}_{lit})$ and let the vocabulary for $\mathcal{O}$ and $I = (\Delta^I, .^I)$ be an interpretation for $\mathcal{O}$. A $SPARQL - DL_E$ query atom q can be defined as:

$$
\begin{aligned}
q ::= \ & Type(a, C) \mid PropertyValue(a, p, v) \mid SameAs(a, b) \mid DiffernetForm(a, b) \\
& \mid EquivalentClass(C_1, C_2) \mid SubClassOf(C_1, C_2) \mid DisjointWith(C_1, C_2) \\
& \mid ComplementOf(C_1, C_2) \mid EquivalentProperty(p_1, p_2) \mid SubPropertyOf(p_1, p_2) \\
& \mid InversOf(p_1, p_2) \mid ObjectProperty(p) \mid DataProperty(p) \mid Functional(p) \\
& \mid InverseFunctional(p) \mid Transitive(p) \mid Symmetric(p) \mid Irreflexive(a) \\
& \mid Class(a) \mid Property(a) \mid Individual(a) \mid Irreflexive(a) \\
& \mid StrictSubClass(a) \mid StrictSubProperty(a) \mid Reflexive(a)
\end{aligned}
$$

where $a_{(i)} \in \mathcal{V}_{uri} \cup \mathcal{V}_{var} \cup \mathcal{V}_{bnode}$,$d \in \mathcal{V}_{uri} \cup \mathcal{V}_{var} \cup \mathcal{V}_{bnode} \cup \mathcal{V}_{lit}$, $C_{(i)} \in \mathcal{V}_{var} \cup \mathcal{S}_c$, $p_{(i)} \in \mathcal{V}_{uri} \cup \mathcal{V}_{var}$, $\mathcal{V}_{cls}$ is the set of classes, $\mathcal{V}_{op}$ is the set of object properties, $\mathcal{V}_{dp}$ is the set of data properties, $\mathcal{V}_{ind}$ is the set of individuals, $\mathcal{V}_{lit}$ is the set of literals and $\mathcal{V}_D$ is the set of data types of $\mathcal{O}$. Note that query about two new property characteristics of OWL 2 namely, reflexivity and irreflexivity are the new query atoms of $SPARQL - DL_E$ (see section 2.2.3). However, OWL 2 RL does not include reflexivity, so our reasoning system does not support querying about reflexivity.

Let an evaluation $\delta : \mathcal{V}_{ind} \cup \mathcal{V}_{bnode} \cup \mathcal{V}_{lit} \rightarrow \Delta^{\mathcal{I}}$ be a mapping from the individual

Form of the Query atom $\mu(q)$	$\mathcal{I} \vDash_\delta \mu(q)$ if
$Type(a, C)$	$\delta(a) \in C^I$
$PropertyValue(a, p, v)$	$< \delta(a), \delta(v) > \in p^I$
$SameAs(a, b)$	$\delta(a) = \delta(b)$
$DifferentFrom(a, b)$	$\delta(a) \neq \delta(b)$
$SubClassOf$	$C_1^I \subseteq C_2^I$
$StrictSubClassOf$	$C_1^I \subseteq C_2^I$ and $C_2^I \subsetneq C_1^I$
$EquivalentClass$	$C_1^I = C_2^I$
$DisjointWith(C_1, C_2)$	$C_1^I \cap C_2^I = \emptyset$
$ComplementOf(C_1, C_2)$	$C_1^I = \Delta^I \setminus C_2^I$
$SubPropertyOf(p, q)$	$p^I \subseteq q^I$
$StrictSubPropertyOf(p, q)$	$p^I \subseteq q^I$ and $q^I \subsetneq p^I$
$EquivalentPropertyOf(p, q)$	$p^I = q^I$
$Functional(p)$	$< x, y > \in p^I$ and $< x, z > \in p^I$ implies $y = z$
$InverseFunctional(p)$	$< y, x > \in p^I$ and $< z, x > \in p^I$ implies $y = z$
$Transitive(p)$	$< x, y > \in p^I$ and $< y, z > \in p^I$ implies $x = z$
$Symmetric(p)$	$< x, y > \in p^I$ implies $< y, x > \in p^I$
$InverseOf(p_1, p_2)$	$< x, y > \in p_1^I$ implies $< y, x > \in p_2^I$
$Reflexive(p)$	$< x, y > \in p^I$ implies $x = y$
$Irreflexive(p)$	$< x, y > \in p^I$ implies $x \neq y$
$Class(c)$	$c \in \mathcal{V}_{cls}$
$Property(p)$	$p \in \mathcal{V}_{op}$ or $p \in \mathcal{V}_{dp}$
$Individual(i)$	$i \in \mathcal{V}_{ind}$

Table 6.1: Satisfaction of a $SPARQL - DL_E$ query atom with respect to an interpretation

names, blank nodes, and literals used in the query to the elements of interpretation domain $\Delta^{\mathcal{I}}$ with the requirement $\delta(a) = a^{\mathcal{I}}$ if $a \in \mathcal{V}_{ind}$ or $a \in \mathcal{V}_{ind}$. The interpretation $\mathcal{I}$ satisfies a query atom q if q is compatible with the corresponding condition listed in Table 6.1 and $\mathcal{I}$ satisfies a query $Q = q_1 \wedge \ldots \wedge q_n$. w.r.t. an evaluation δ **iff** $\mathcal{I} \models_\delta q_i$ for

every $i = 1, \ldots, n$.

A solution to query Q is a mapping $\mu : \mathcal{V}_{var} \rightarrow \mathcal{V}_{cls} \cup \mathcal{V}_{op} \cup \mathcal{V}_{dp} \cup \mathcal{V}_{lit}$ such that when all the variables in Q are substituted with the corresponding value from μ we get a ground query $\mu(Q)$ (i.e., atoms having variables) compatible with $\mathcal{V}_{\mathcal{O}}$ and $\mathcal{O} \vDash \mu(Q)$.

6.2.3 The $SPARQL - DL_E$ Query Format

We defined the query types for $SPARQL - DL_E$ queries in Subsection 6.2.2 and semantics in Table 6.1. Now we present an abstract syntax for $SPARQL - DL_E$ queries in BNF in Listing 6.1. We follow the same assumption for BNF described in Subsection 5.3.2.

```
Queries ::= [Prefixes] Query
Query ::= AskQuery | SelectQuery
AskQuery ::= 'ASK {' Atoms '}'
Atoms ::= Atom {,Atom}*
Primitive ::=Variable | Literal | IRI
selectQuery = 'SELECT' ['DISTINCT'][{Variable}*]['WHERE']'{'Atoms'}'
                                              ['OR~WHERE']'{'Atoms'}'

Atom= 'Type(' Primitive, Primitive ')' | 'SubClassOf(' Primitive, Primitive ')'
    | 'DiffernetForm(' Primitive, Primitive ')' | 'Transitive(' Primitive ')'
    | 'DisjointWith(' Primitive, Primitive ')' | 'Class(' Primitive ')'
    | 'EquivalentProperty(' Primitive, Primitive ')' | 'Individual(' Primitive ')'
    | 'InversOf(' Primitive, Primitive ')' | 'ObjectProperty(' Primitive ')'
    | 'Functional(' Primitive ')' | 'InverseFunctional(' Primitive ')'
    | 'Symmetric(' Primitive ')' | 'PropertyValue(' Primitive, Primitive, Primitive ')'
    | 'SubPropertyOf(' Primitive, Primitive ')' | 'Irreflexive(' Primitive ')'
    | 'SameAs(' Primitive, Primitive ')' | 'DataProperty(' Primitive ')'
    | 'Property(' Primitive ')' | 'EquivalentClass(' Primitive, Primitive ')'
    | 'ComplementOf(' Primitive, Primitive ')'

Variable = ?[a − zA − Z]+
Prefixes = Prefix':'Suffix
Literals = [a − zA − Z0 − 9]+
Prefix = Literals
Suffix = 'http://'Literals
IRI = Suffix | Prefixes
```

Listing 6.1: Abstract syntax for SPARQL DL queries

6.3 Implementation of $SPARQL - DL_E$

The SPARQL-DL API is built on top of the OWL API [HB09]. It is fully aligned with the OWL 2 standard. All the reasoners provide interfaces for standard reasoning tasks listed in Subsection 2.4.1. The SPARQL-DL API was designed in such a way that it can answer mixed TBox and ABox queries by invoking all of the required interfaces provided by the reasoners. The integration of the SPARQL-DL API with a reasoner is shown in Figure 6.1.

Figure 6.1: The integration of the SPARQL DL API between reasoners and application programs.

We integrated the SPARQL-DL API engine in our system. For the integration, we developed all the required interfaces for answering mixed ABox and TBox queries. In our system, all explicit and implicit knowledge of an ontology $\mathcal{O}$ is materialized into relational databases. The list of necessary interfaces $allC(\mathcal{O})$, $allDP(\mathcal{O})$, $allOP(\mathcal{O})$, $allI(\mathcal{O})$, etc., are identified in [KS08]. We discuss all the interfaces with implementation details as follows:

1. $allC(\mathcal{O})$, $allDP(\mathcal{O})$, $allOP(\mathcal{O})$, $allI(\mathcal{O})$ return all classes, data properties, object properties, and individuals respectively defined in $\mathcal{O}$. We stored asserted and inferred information from ontologies into databases and we also used meta mapping.

Therefore, we need SQL queries to retrieve relevant information from corresponding tables of the relational database. The SQL queries for these interfaces are

```
SELECT Class FROM TypeOf
SELECT Predicate FROM RelationshipData
SELECT Predicate FROM RelationshipObj
SELECT Ind FROM TypeOf
```

2. $subC(\mathcal{O}, \mathcal{C})$, $supC(\mathcal{O}, \mathcal{C})$, $eqC(\mathcal{O}, \mathcal{C})$ return all sub classes, super classes, and equivalent classes respectively of class C in $\mathcal{O}$. The SQL queries for these interfaces are

```
SELECT SubID FROM SubClassOf WHERE SuperID = C
SELECT SuperID FROM SubClassOf WHERE SubID = C
SELECT Class2 FROM EquivalentClasses WHERE Class1 = C
SELECT Ind FROM TypeOf WHERE Class = C
```

3. $subOP(\mathcal{O}, \mathcal{P})$, $supOP(\mathcal{O}, \mathcal{P})$, $eqOP(\mathcal{O}, \mathcal{P})$, $subDP(\mathcal{O}, \mathcal{P})$, $supDP(\mathcal{O}, \mathcal{P})$, $eqDP(\mathcal{O}, \mathcal{P})$ return all sub object properties, super object properties, equivalent object properties, sub data properties, super data properties, and equivalent data properties respectively of properties p in $\mathcal{O}$. The SQL queries for these interfaces are

```
SELECT SubPropertyID FROM SubPropertyOf WHERE SuperPropertyID = P
SELECT SuperPropertyID FROM SuperPropertyOf WHERE SubPropertyID = P
SELECT Property2 FROM EquivalentProperties WHERE Property1 = P
SELECT SubPropertyID FROM SubPropertyOf WHERE SuperPropertyID = P
SELECT SuperPropertyID FROM SuperPropertyOf WHERE SubPropertyID = P
SELECT Property2 FROM EquivalentProperties WHERE Property1 = P
```

4. $en(\mathcal{O}, q)$ checks whether $\mathcal{O} \models q$ for a $SPARQL - DL_E$ atom q. This query is evaluated by the SPARQL-DL API by invoking all the interfaces discussed in (1)-(3). For instance, if we consider $q = SubClassOf(''Person'', c)$, the SPARQL-DL API will invoke the interface $subC(\mathcal{O}, Person)$ to retrieve all the subclasses of "Person" from the database.

6.4 Adding the aggregation and grouping feature to $SPARQL - DL_E$

The W3C is working to extend the expressive power of SPARQL, and SPARQL 1.1 is the current proposed standard. The working draft of this proposed standard is available in [HP12]. The proposed standardization has recommended a number of new features, including aggregation and grouping. Aggregation and grouping are important operations in analyzing and sorting data. Aggregate functions compute a scalar value from a multi-set of values. These functions are regularly needed to count a number of values. For example, they are needed to identify the minimum or maximum of a set of values. Grouping additionally allows aggregates to be computed on groups of values. Our reasoning system supports aggregation, namely *COUNT, SUM, MIN, MAX, AVG,* and grouping *Group By*, which are defined in version 1.1 of SPARQL. To support these features, we add the query results into a relational database view (temporary table) and then extract the result from view by SQL queries with aggregation and grouping. In this way, we utilized the advantage of relational databases functionalities to support aggregation and grouping with $SPARQL - DL_E$.

We can explain this feature as follows: for instance, if we want to count the number of students who are also employees from an ontology about universities, then a $SPARQL - DL_E$ query to retrieve this information can be formulated as follows using the *count* aggregation operator:

```
SELECT count( ?x ) WHERE
    {
    Type( ?x, Student ), Type( ?x, ?E ), SubClassOf( ?E, Employee )
    }
```

Chapter 7

Implementation and Performance Analysis

We developed OwlOntDB, a Java-based framework for storing and reasoning with OWL 2 RL ontologies. The OwlOntDB software is written in Java and can be operated by a simple command-line interface.

7.1 System Description

An overview of the software architecture of the OwlOntDB application is shown in Figure 7.1 It consists of 4 components.

- *Classifier*. This takes an OWL 2 RL ontology in XML format, and invokes a DL reasoner to classify the input ontology. We use *Pellet* for TBox reasoning, which infers complete subsumption relationships between classes and properties.

- *Datalog Generator*. The datalog generator first parses the classified OWL file using

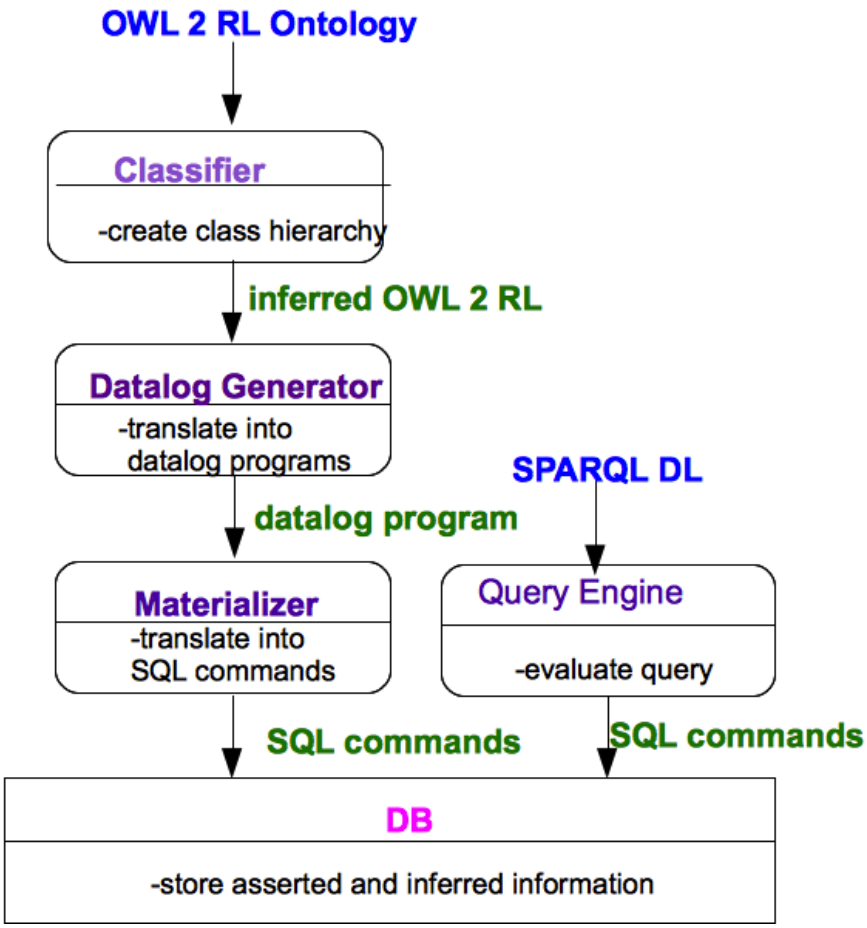

Figure 7.1: The OwlOntDB tool architecture

the OWL API [HB09]. The datalog generator extracts the set of axioms and facts from the ontology using this API. Finally, it recursively applies the DLP-based translation rules (Subsection 4.2) to generate equivalent datalog rules for each axiom and facts. The design of the OWL API was directly based on the OWL 2 Structural Specification and it treats an ontology as a set of axioms and facts

- *Materializer*. This module infers knowledge from the translated ontologies (i.e., from datalog programs). It applies a forward-chaining algorithm over the datalog program to infer knowledge. Then it translates each asserted or inferred fact into

its equivalent SQL statement. The first step of the translation is analyzing and parsing the input. We developed a parser based on the translation discussed in section 5.3.2. This parser generates equivalent SQL statements from the datalog program. The materializer module stores both original assertions and the assertions inferred by the DL reasoner and materialization by executing a sequence of SQL commands. This module also includes a restrictions checker submodule to check ABox consistency. Before inserting each assertion to the database, the materializer module invokes the restrictions checker to identify any violations and keep the ABox always consistent by allowing only the assertions that are consistent with the TBox of the ontology. Any RDBMS can be used as a back-end for our system OwlOntDB. We tested our system using the MySQL database as a back-end.

- *Query Engine.* The query language supported by OwlOntDB is $SPARQL - DL_E$. Application programs or users interact with OwlOntDB using $SPARQL - DL_E$ queries, and queries are answered by directly retrieving inferred results from the database using SQL statements. There is no inference during the query-answering stage because the inference has already been completed at the time the data is loaded. This technique improved the query response time.

7.2 Performance Analysis

We developed a scalable storage and reasoning system for OWL 2 RL ontologies. There are no widely accepted benchmarks for OWL 2 and not even for OWL 1 [MS06]. So comparing the performance of DL reasoning systems is difficult. In [MS06], the authors also identify that there are some ontologies that can be used as standards for testing

TBox reasoning; however, there are no such standard tests for ABox reasoning. Hence, they constructed their own data set. The data set is freely available from the KAON2 Web site. We use the wine ontology, an ontology containing a classification of wines, from the KAON2 site [1]. This ontology contains functional roles, disjunctions, and existential quantifiers. We used two datasets $wine_1$ - the original wine ontology, and $wine_5$ - which they synthetically generated by replicating 2^n times the ABox of $wine_1$. We evaluated the following query over the wine ontology:

Query 7.2.1. ($\mathbf{WQ_1}$). *Determine all the instances of "AmericanWine". This is an ABox query. This query can be written using SPARQL − DL_E as follows:*

```
PREFIX wine: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>
   SELECT ?i WHERE {
     Type(?i, wine:AmericanWine)
      }
```

Query 7.2.2. ($\mathbf{WQ_2}$). *Determine all the instances of wines which type is "Dry". This a conjunctive query and a mixed ABox and TBox query. The query is given below:*

```
PREFIX wine: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#>
   SELECT ?i WHERE {
     Type(?i,?x), SubClassOf(?x, wine:DryWine)
      }
```

We also used the LUBM benchmark ontology which is developed at the Lehigh University for testing performance of ontology management and reasoning systems [2]. This is an ontology about organizational structures of universities. LUBM provides a data generator. We used another two datasets from LUBM namely, $lubm_1$, and $lubm_{10}$,

[1] http://kaon2.semanticweb.org/download/test_ontologies.zip

[2] http://swat.cse.lehigh.edu/downloads/index.html

where 1 and 10 are the number of universities used to generate test data. We evaluated the following query over the LUBM ontology:

Query 7.2.3. (**LQ$_1$**). *Find all the students who are also employees and what kind of employee they are. (e.g., ResearchAssistant). This is a mixed ABox and TBox query. This query can be written using SPARQL − DL$_E$ as follows:*

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
   SELECT * WHERE {
     Type(?x, ub:Student), Type(?x, ?C), SubClassOf(?C, ub:Employee)
       }
```

Query 7.2.4. (**LQ$_2$**). *Find out the name of all students. This is an ABox. This query can be written as follows:*

```
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
   SELECT * WHERE {
     Type(?x, ub:Student)
       }
```

Table 7.1 show the number of axioms for each ontology. In this table, the TBox axioms are subclass axioms ($C \sqsubseteq D$), functional property axioms, domain axioms, range axioms, and subproperty axioms ($R \sqsubseteq S$) and the ABox axioms are class assertion axioms ($C(a)$), property assertion axioms ($P(a, b)$).

We evaluated LQ_1, LQ_2, WQ_1, and WQ_2 queries over the corresponding ontologies using OwlOntDB and also using a tableau-based in-memory reasoner namely *Pellet*. The OwlOntDB materializes the information to a database, so it needs an initial processing for query evaluation. The initial processing time (i.e., materialization time) for four datasets required for OwlOntDB are given in the Table 7.2.

Ontology	$C \sqsubseteq D$	Functional	Domain	Range	$R \sqsubseteq S$	$C(a)$	$P(a,b)$
$wine_1$	126	6	6	9	9	247	246
$wine_5$	126	6	6	9	9	2717	2706
$lubm_1$	36	0	25	18	9	20659	82415
$lubm_{10}$	36	0	25	18	9	5658	11096694

Table 7.1: Number of axioms in test ontologies

$lubm_1$	$lubm_{10}$	$wine_1$	$wine_5$
117.58 s	830.53 s	21.422 s	217.5 s

Table 7.2: Time required for materialization

The query evaluation time for both $\mathsf{O}_{\text{wl}}\mathsf{O}_{\text{nt}}\mathsf{DB}$ and *Pellet* is given in Table 7.3. We already mention in Chapter 3.1, Tableau-based reasoners support more expressive DL language and efficiently perform reasoning over ontologies with large TBoxes and small ABoxes. From our experiments, we found that for ontologies with smaller TBoxes but larger ABoxes ($lubm_{100},\ wine_5$), our reasoning outperformed its tableau counterparts; although we are using a tableau-based DL reasoner for TBox reasoning, we get better performance than tableau-based reasoners because we materialized the inferred information into databases. After the materialization, reasoning over materialized ontologies are simply SQL queries into a relational database. On the other hand, main-memory

based reasoners perform inferencing for each query. So it takes a longer time when the size of the ABoxes are large. The disadvantage of the materialization technique is that it takes a long time initially to materialize the ontology. For ontologies with larger TBoxes but smaller ABoxes, Tableau-based reasoners are still better than our system because materialization in our system is time consuming. From the test results, we can see that if we combine the initial processing time for ($lubm_1$, $wine_1$) with query answering time, then the performance of the Tableau-based reasoner is still better.

Dataset	Reasoner	LQ_1	LQ_2
$lubm_1$	*Pellet*	129.02 s	127.06 s
$lubm_1$	OwlOntDB	3.43s	0.79 s
$lubm_{100}$	*Pellet*	142.80 s	127.66 s
$lubm_{100}$	OwlOntDB	29.07 s	1.0 s

Dataset	Reasoner	WQ_1	WQ_2
$wine_1$	*Pellet*	2.95 s	2.98 s
$wine_1$	OwlOntDB	0.047 s	0.11 s
$wine_5$	*Pellet*	363.089 s	363.59 s
$wine_5$	OwlOntDB	0.3435 s	1.171 s

Table 7.3: Test results

The computational complexity for instance checking and answering conjunctive query over OWL 2 RL ontologies for rule-based implementations is PTime-complete [MGH+09]. The complexity of our forward-chaining-based materialization algorithm is at most polynomial with the total size of the axioms in the ontology.

Chapter 8

Conclusion and Discussion

In this chapter, we summarize the contributions of this thesis, followed by the limitations and a few future research directions.

8.1 Summary

Scalable reasoning is crucial for the development of large-scale ontology-driven applications. In this thesis, we propose a practical scalable ontology reasoning approach. This approach combines DL reasoners for TBox reasoning with datalog-based materialization using relational databases for ABox reasoning. The combination of DL reasoners with logic-based inferencing using datalog exploits the particular advantages of each one in order to support expressive ontologies and large amounts of instance data. Logic-based approaches give us scalable reasoning strategies, and database systems are well-known technology for handling large amounts of data. There is a number of advantages and disadvantage for materialization techniques. However, they are good for many applications where query answering is more frequent and updating is less frequent.

A brief summary of our main achievements are given below:

- We developed a translator to generate datalog programs from OWL 2 RL ontologies by extending the DLP mapping. This translator can be used in various applications, for example, to drive rule-based systems using OWL 2 RL ontologies. We used this translation to generate a multi-agent rule-based system from an ontology in [RFM12] to verify the properties of the rule-based system by Model Checking. To check ABox consistency, we developed a restrictions checker for some of OWL 2 RL axioms that cannot be handled using logic program. In Chapter 4, we discussed the translation.

- We developed a materialized algorithm that performs a forward chaining over a datalog program, then translates the resulting program to a set of SQL statements. The SQL statements are used to store the information in a relational database. In Chapter 5, we discussed the materialization.

- We modified an existing SPAQRL-DL API and integrated it with our system. We also added aggregation and grouping operators to the $SPARQL - DL_E$ language to query over the materialized ontologies. In Chapter 6, we discussed the query processing.

8.2 Limitations and Future Work

Our approach is still preliminary and some improvements can be made. The reasoning process does not address efficiently the ontology update problem (i.e., deleting/inserting facts or axioms in ontologies). The current strategy is to materialize the whole ontology

again if the ontology is updated. This is the simplest method but it brings a heavy overhead. Although ontology updates are infrequent in many real applications, incremental maintenance of inferred information will speed up the reasoning process. We have identified one possible approach to improve the materialization technique, namely modularization. For modularization technique, we have to develop an algorithm to find a subset of an ontology that will be affected by an update operation on the knowledge base. In this case, the materialization cost will be minimized. There are also some works in deductive database areas for incremental maintenance of truth in materialization [VSM05]; a further investigation can be made to check whether these techniques can be used for relational databases.

Another disadvantage of the materialization technique is the cost of initial processing. However, the query processing is very fast for materialized ontologies. Another interesting future direction can be the application of parallel and distributed computing for reducing the materialization time.

Appendix I

In this Appendix I, we present the datalog program in 8.2 generated from an ontology about people given in 8.1. The ontology about people is given in Manchester syntax.

```
 Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
 Prefix: owl: <http://www.w3.org/2002/07/owl#>
 Prefix: xml: <http://www.w3.org/XML/1998/namespace>
 Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
 Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
 Prefix: skos: <http://www.w3.org/2004/02/skos/core#>

Ontology: <http://logic.stfx.ca/ontologies/person.owl>

 Annotations:
   rdfs:comment "This is an ontology about people."^^rdf:PlainLiteral

 AnnotationProperty: rdfs:comment

 Datatype: xsd:string

 ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#friendOf>
     Characteristics:
         Symmetric

ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#hasFather>
     SubPropertyOf:
         <http://logic.stfx.ca/ontologies/person.owl#hasParent>
     Characteristics:
         Functional

 ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#hasMother>
     SubPropertyOf:
         <http://logic.stfx.ca/ontologies/person.owl#hasParent>
     Characteristics:
         Functional
     Domain:
         <http://logic.stfx.ca/ontologies/person.owl#Person>
     Range:
```

```
        <http://logic.stfx.ca/ontologies/person.owl#Woman>

ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#motherOf>
    SubPropertyOf:
        <http://logic.stfx.ca/ontologies/person.owl#parentOf>
    Characteristics:
        InverseFunctional

ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#fatherOf>
    SubPropertyOf:
        <http://logic.stfx.ca/ontologies/person.owl#parentOf>
    Characteristics:
        InverseFunctional


ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#hasSister>
    SubPropertyOf:
        <http://logic.stfx.ca/ontologies/person.owl#hasSibling>


ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#hasUncle>
    SubPropertyChain:
        <http://logic.stfx.ca/ontologies/person.owl#hasFather>
                    o <http://logic.stfx.ca/ontologies/person.owl#hasBrother>


ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#hasBrother>
    SubPropertyOf:
        <http://logic.stfx.ca/ontologies/person.owl#hasSibling>


ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#parentOf>
    InverseOf:
        <http://logic.stfx.ca/ontologies/person.owl#hasParent>


ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#hasParent>
    InverseOf:
        <http://logic.stfx.ca/ontologies/person.owl#parentOf>

ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#hasChild>
ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#hasSibling>
    Characteristics:
        Asymmetric


ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#marriesTo>
    Characteristics:
        Irreflexive

ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#ancestorOf>
    Characteristics:
```

```
        Transitive
    Domain:
        <http://logic.stfx.ca/ontologies/person.owl#Person>
    Range:
        <http://logic.stfx.ca/ontologies/person.owl#Person>

ObjectProperty: <http://logic.stfx.ca/ontologies/person.owl#P>

DataProperty: <http://logic.stfx.ca/ontologies/person.owl#hasAge>
    Characteristics:
        Functional

DataProperty: <http://logic.stfx.ca/ontologies/person.owl#hasName>
   Domain:
        <http://logic.stfx.ca/ontologies/person.owl#Person>
    Range:
        xsd:string
DataProperty: <http://logic.stfx.ca/ontologies/person.owl#hasLastName>
    SubPropertyOf:
        <http://logic.stfx.ca/ontologies/person.owl#hasName>
Class: <http://logic.stfx.ca/ontologies/person.owl#Mother>
    SubClassOf:
        <http://logic.stfx.ca/ontologies/person.owl#Person>,
        <http://logic.stfx.ca/ontologies/person.owl#Parent>
         and <http://logic.stfx.ca/ontologies/person.owl#Woman>
Class: <http://logic.stfx.ca/ontologies/person.owl#HappyPerson>
    SubClassOf:
        <http://logic.stfx.ca/ontologies/person.owl#Person>,
        <http://logic.stfx.ca/ontologies/person.owl#Happy>
         and <http://logic.stfx.ca/ontologies/person.owl#Person>,
        <http://logic.stfx.ca/ontologies/person.owl#hasChild>
                          only <http://logic.stfx.ca/ontologies/person.owl#Happy>

Class: <http://logic.stfx.ca/ontologies/person.owl#Parent>
     SubClassOf:
        <http://logic.stfx.ca/ontologies/person.owl#Person>

Class: <http://logic.stfx.ca/ontologies/person.owl#Woman>
     SubClassOf:
        <http://logic.stfx.ca/ontologies/person.owl#Person>

Class: <http://logic.stfx.ca/ontologies/person.owl#Man>
    SubClassOf:
        <http://logic.stfx.ca/ontologies/person.owl#Person>

Class: <http://logic.stfx.ca/ontologies/person.owl#Person>

Class: <http://logic.stfx.ca/ontologies/person.owl#Child>
    SubClassOf:
        <http://logic.stfx.ca/ontologies/person.owl#Person>

Class: <http://logic.stfx.ca/ontologies/person.owl#Dad>
```

```
   EquivalentTo:
       <http://logic.stfx.ca/ontologies/person.owl#Father>
    SubClassOf:
       <http://logic.stfx.ca/ontologies/person.owl#Person>

 Class: <http://logic.stfx.ca/ontologies/person.owl#Father>
    EquivalentTo:
       <http://logic.stfx.ca/ontologies/person.owl#Dad>
    SubClassOf:
       <http://logic.stfx.ca/ontologies/person.owl#Person>

 Class: <http://logic.stfx.ca/ontologies/person.owl#Happy>

Individual: <http://logic.stfx.ca/ontologies/person.owl#Bob>
    Types:
       <http://logic.stfx.ca/ontologies/person.owl#Child>

 Individual: <http://logic.stfx.ca/ontologies/person.owl#John>
     Types:
       <http://logic.stfx.ca/ontologies/person.owl#HappyPerson>
     Facts:
       <http://logic.stfx.ca/ontologies/person.owl#hasChild>
                          <http://logic.stfx.ca/ontologies/person.owl#Tom>

 Individual: <http://logic.stfx.ca/ontologies/person.owl#Marry>
    Types:
       <http://logic.stfx.ca/ontologies/person.owl#Woman>
    Facts:
       <http://logic.stfx.ca/ontologies/person.owl#hasChild>
                            <http://logic.stfx.ca/ontologies/person.owl#Bob>

 Individual: <http://logic.stfx.ca/ontologies/person.owl#Tom>
    Types:
       <http://logic.stfx.ca/ontologies/person.owl#Person>
```

Listing 8.1: An ontology about Person in Manchester Syntax

```
<Person>(<John>)
<Person>(<Bob>)
<HappyPerson>(<John>)
<Person>(<Marry>)
<hasChlid>(<John>,<Tom>)
<hasChild>(<Marry>, <Bob>)
<Person>(<Tom>
<Happy>(<John>)

SubClassOf: <Person>, <Woman>
                  <Person>(a) <- <Woman>(b)

ObjectInverseOf: <hasParent> , <parentOf>
            <hasParent>(a, b) <- <parentOf>(b,a)
```

```
           <parentOf>(b,a)<- <hasParent>(a,b)

TransitiveProperty: <ancestorOf>
        <ancestorOf>(a,c) <- <ancestorOf>(a,b) ^ <ancestorOf>(b,a)

SubObjectProperty: <hasSibling> ,<hasBrother>
         <hasSibling>(a,b) <- <hasBrother>(a,b)

SubObjectProperty: <parentOf> , <fatherOf>
      <parentOf>(a,b) <- <fatherOf>(a,b)

FunctionalObjectProperty(<hasMother>)

IrreflexiveObjectProperty(<marriesTo>)

EquivalentClassOf: <Dad> , <Father>
          <Dad>(a,b) <- <Father>(a,b)
          <Father>(a,b) <- <Dad>(a,b)

SubClassOf: <Person> ,<HappyPerson>
           <Person>(a) <- <HappyPerson> (a)

SubObjectProperty: <parentOf> , <motherOf>
           <parentOf>(a,b) <- <motherOf>(a,b)

PropertyRange: <Person> , <ancestorOf>
        <Person>(b) <- <ancestorOf>(a,b)

PropertyRange: <Woman> , <hasMother>
         <Woman>(b) <- <hasMother>(a,b)

FunctionalObjectProperty (<hasFather>)

PropertyDomain: <Person>, <ancestorOf>
           <Person>(b) <- <ancestorOf>(b,a)

PropertyDomain: <Person>, <hasMother>
        <Person>(b) <- <hasMother>(b,a)

InverseFunctionalObjectProperty(<fatherOf>)

InverseFunctionalObjectProperty(<motherOf>)

SymmetricObjectProperty(<friendOf>)
             <friendOf>(a,b) <-<friendOf>(b,a)

PropertyDomain: <Person>,<hasName>
            <Person>(b) <- <hasName>(b,a)

ObjectPropertyChain: <hasUncle>,<hasFather> ,<hasBrother>
       <hasUncle>(a,c) <- <hasFather>(a,b) ^ <hasBrother>(b,c)
```

```
AllValuesFrom: <Happy>, <HappyPerson>, <hasChild>(b,a)
     <Happy>(b) <- <HappyPerson>(a) ^ <hasChild>(b,a)

SubObjectProperty: <hasParent> ,<hasMother>
      <hasParent>(a,b) <- <hasMother>(a,b)

SubObjectProperty: <hasSibling>,<hasSister>
     <hasSibling>(a,b) <- <hasSister>(a,b)

UnionOf: <Parent> , <Father>,<Mother>
     <Parent>(a) <- <Father>(a)
     <Parent>(a) <- <Mother>(a)

IntersectionOf: <Mother>, <Parent>,<Woman>
        <Mother>(a) <- <Parent>(a) ^ <Woman>(a)

SubClassOf: <Person>,<Dad>
      <Person>(a) <- <Dad>(a)

SubClassOf: <Person>, <Child>
      <Person>(a) <- <Child>(a)

SubClassOf: <Person> , <Parent>
       <Person>(a) <- <Parent>(a)


SomeValuesFrom: <Parent> , <hasChild>, <Person>
           <Parent>(a) <- <hasChild>(a,b) ^ <Person>(b)

IntersectionOf: <HappyPerson>,<Happy>,<Person>
           <HappyPerson>(a) <- <Happy>(a) ^ <Person>(a)

AsymmetricObjectProperty(<hasSibling>)

SubClassOf: <Person>,<Father>
       <Person>(a) <- <Father>(a)

FunctionalProperty(<hasAge>)

SubObjectProperty: <hasParent>,<hasFather>
     <hasParent>(a,b) <- <hasFather>(a,b)

SubClassOf: <Person>,<Mother>
        <Person>(a) <- <Mother>(a)
```

Listing 8.2: Datalog rules for the Person ontology

Bibliography

[ACG+05] Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. Quonto: Querying ontologies. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 1670–1671. AAAI Press / The MIT Press, 2005.

[ACKZ09] Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. The DL-Lite Family and Relations. *Journal of Artificial Intelligence Research (JAIR)*, 36:1–69, 2009.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AJPS10] Lina Al-Jadir, Christine Parent, and Stefano Spaccapietra. Reasoning with large ontologies stored in relational databases: The ontomind approach. *Data & Knowledge Engineering*, 69(11):1158–1180, November 2010.

[BB93] Alex Borgida and Ronald J. Brachman. Loading data into description reasoners. *SIGMOD Rec.*, 22:217–226, June 1993.

[BN03] Franz Baader and Werner Nutt. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[Bro05] J. Broekstra. *Storage, Querying and Inferencing for Semantic Web Languages*. PhD thesis, VU Amsterdam, 2005.

[CGL+07] Diego Calvanese, Giuseppe Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite Family. *Journal of Automated Reasoning*, 39:385–429, October 2007.

[CGL+09] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, and Riccardo Rosati. Ontologies and Databases: The DL-Lite Approach. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web*, volume 5689 of *Lecture Notes in Computer Science*, pages 255–356. Springer, 2009.

[dMRGAM08] Maria del Mar Roldan-Garcia and Jose F. Aldana-Montes. DBOWL: Towards a Scalable and Persistent OWL Reasoner. In *Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, ICIW '08, pages 174–179, Washington, DC, USA, 2008. IEEE Computer Society.

[FvHH+01] D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, and P. F.

Patel-Schneider. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2):38–45, 2001.

[GFH+03] Jennifer Golbeck, Gilberto Fragoso, Frank W. Hartel, James A. Hendler, Jim Oberthaler, and Bijan Parsia. The national cancer institute's thésaurus and ontology. *Journal of Web Semantics*, 1(1):75–80, 2003.

[GHVD03] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description logic programs: combining logic programs with description logic. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 48–57, New York, NY, USA, 2003. ACM.

[GPSK06] Bernardo Cuenca Grau, Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Modularity and Web Ontologies. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *KR*, pages 198–209. AAAI Press, 2006.

[Gru93] Thomas Gruber. A translation approach to protable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[HB09] M. Horridge and S. Bechhofer. The OWL API: A java API for working with OWL 2 ontologies. In *6th OWL Experienced and Directions Workshop (OWLED)*, October 2009.

[HKS06] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible $\mathcal{SROIQ}$. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 57–67. AAAI Press, 2006.

[HLTB04] Ian Horrocks, Lei Li, Daniele Turi, and Sean Bechhofer. The instance store: DL reasoning with large numbers of individuals. In Volker Haarslev and Ralf Möller, editors, *Description Logics*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[HM00] James Hendler and Deborah L. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 6(15):67–73, 2000.

[HM01] Volker Haarslev and Ralf Moller. Description of the RACER system and its applications. In D. L. McGuinness et al, editor, *Proceedings of the 2001 International Workshop on Description Logics (DL-2001)*. CEUR Workshop Proceedings, 2001.

[Hor02] Ian Horrocks. DAML+OIL: a description logic for the semantic web. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1):4–9, March 2002.

[Hor10] Ian Horrocks. Scalable ontology-based information systems. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 2–2, New York, NY, USA, 2010. ACM.

[HP12] Steve Harris and Eric Prud'hommeaux. SPARQL 1.1 Query Language. W3C Working Draft. Available at http://www.w3.org/TR/sparql11-query/, January 2012.

[HPS10] Ian Horrocks and Peter F. Patel-Schneider. KR and Reasoning on the semantic web: OWL. In *Handbook of Semantic Web Technologies*, chapter 9. Springer, 2010.

[HPSvH03] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[HS04] Ian Horrocks and Ulrike Sattler. Decidability of $\mathcal{SHIQ}$ with complex role inclusion axioms. *Artificial Intelligence*, 160(1–2):79–104, December 2004.

[KMR10] Markus Krötzsch, Anees Mehdi, and Sebastian Rudolph. Orel : Database-driven reasoning for owl 2 profiles. In *23rd Int.Workshop on Description Logics (DL2010),*, pages 114–124, 2010.

[KS08] Petr Kremen and Evren Sirin. SPARQL-DL Implementation Experience. In *OWL: Experiences and Directions (OWLED)*, 2008.

[LMZ+07] Jing Lu, Li Ma, Lei Zhang, Jean-Sébastien Brunner, Chen Wang, Yue Pan, and Yong Yu. Sor: a practical system for ontology storage, reasoning and search. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1402–1405. VLDB Endowment, 2007.

[MGH+09] Boris Motik, Bernardo Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web Ontology Language: Profiles. W3C Recommendation, Available at http://www.w3.org/TR/owl2-profiles/, October 2009.

[MM04] Frank Manola and Eric Miller, editors. *RDF Primer*. W3C Recommendation. World Wide Web Consortium, February 2004.

[Mot08] Boris Motik. KAON2 - Scalable Reasoning over Ontologies with Large Data Sets. *ERCIM News*, 2008(72), 2008.

[MS06] Boris Motik and Ulrike Sattler. A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. In Miki Hermann and Andrei Voronkov, editors, *Proc. of the 13th Int. Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006)*, volume 4246 of *LNCS*, pages 227–241, Phnom Penh, Cambodia, November 13–17 2006. Springer.

[MSH09] Boris Motik, Rob Shearer, and Ian Horrocks. Hypertableau reasoning for description logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.

[MSS05] Boris Motik, Ulrike Sattler, and Rudi Studer. Query Answering for OWL-DL with Rules. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 3(1):41–60, 2005.

[MvH04] Deborah McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation Available at http://www.w3.org/TR/owl-features/, February 2004.

[PJC09] Jyotishman Pathak, Thomas M Johnson, and Christopher G Chute. Survey of modular ontology techniques and their applications in the biomedical domain. *Integrated Computer-Aided Engineering*, 16:225–242, 2009.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for

RDF. W3C Recommendation. Available at http://www.w3.org/TR/rdf-sparql-query/, 2008.

[RFM12] Abdur Rakib, Rokan Uddin Faruqui, and Wendy MacCaull. Verifying resource requirements for ontology-driven rule-based agents. In Thomas Lukasiewicz and Attila Sali, editors, *FoIKS*, volume 7153 of *Lecture Notes in Computer Science*, pages 312–331. Springer, 2012.

[RGJDGC+11] Alejandro Rodríguez-González, Enrique Jimenez-Domingo, Angel García-Crespo, Giner Alor-Hernandez, Juan Miguel Gomez-Berbis, and Ruben Posada-Gomez. Designing an ontology to support the creation of diagnostic decision support system. In *Proceedings of the 2nd ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, BCB '11, pages 612–616, New York, NY, USA, 2011.

[RJ03] Cornelius Rosse and José L. V. Mejino Jr. A reference ontology for biomedical informatics: the foundational model of anatomy. *Journal of Biomedical Informatics*, 36(6):478–500, 2003.

[RR06] Alan L. Rector and Jeremy Rogers. Ontological and practical issues in using a description logic to represent medical concept systems: Experience from galen. In Pedro Barahona, François Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors, *Reasoning Web*, volume 4126 of *Lecture Notes in Computer Science*, pages 197–231. Springer, 2006.

[RSSS93] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the coral deductive database system. *SIGMOD Rec.*, 22:167–176, June 1993.

[SML10] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, 2010.

[SP07] E. Sirin and B. Parsia. SPARQL-DL: Sparql query for OWL-DL. *3rd OWL Experiences and Directions Workshop (OWLED-2007)*, 2007.

[SPA11] SPARQL-DL API. http://www.derivo.de/en/resources/sparql-dl-api/, 2011.

[SPG+07] E Sirin, B Parsia, BC Grau, A Kalyanpur, and Y Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, June 2007.

[SPSW01] M. Q. Stearns, C. Price, K. A. Spackman, and A. Y. Wang. SNOMED clinical terms: overview of the development process and project status. In *AMIA Symposium*, pages 662–666, 2001.

[SS08] Markus Stocker and Michael Smith. Owlgres: A Scalable OWL Reasoner. In Catherine Dolbear, Alan Ruttenberg, and Ulrike Sattler, editors, *Proc of the 5th Int Workshop on OWL Experiences and Directions OWLED 2008*, volume 432. CEUR-WS.org, 2008.

[SSS91] Manfred Schmidt-Schau and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelleigence*, 48(1):126. 2.1.2, 1991.

[TBLL01] J. Hendler T. Berners-Lee and O. Lassila. The semantic web. In *Scientific American*, volume 284, pages 34–43, 2001.

[tH05] H. J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Journal of Web Semantics*, 3(2-3):79–115, 2005.

[TH06] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297, 2006.

[TPR10] Edward Thomas, Jeff Z. Pan, and Yuan Ren. TrOWL: Tractable OWL 2 Reasoning Infrastructure. In *the Proc. of the Extended Semantic Web Conference (ESWC2010)*, 2010.

[TZH07] Alexey Tsymbal, Sonja Zillner, and Martin Huber. Ontology - supported machine learning and decision support in biomedicine. In *Proceedings of the 4th international conference on Data integration in the life sciences*, DILS'07, pages 156–171, Berlin, Heidelberg, 2007. Springer-Verlag.

[VSM05] Raphael Volz, Steffen Staab, and Boris Motik. Incrementally Maintaining Materializations of Ontologies Stored in Logic Databases. *Journal of Data Semantics II*, 3360:1–34, 2005. LNCS, Springer.

[WLS03] Timo Weithöner, Thorsten Liebig, and Günther Specht. Storing and querying ontologies in logic databases. In Isabel F. Cruz, Vipul Kashyap, Stefan Decker, and Rainer Eckstein, editors, *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003*, September 2003.