Faculty of Computer Science, Free University of Bozen-Bolzano
Universidade Nova de Lisboa

Master of Science Thesis

Query Evaluation of Tractable Answering using Query Rewriting

by

Zakka Fauzan Muhammad

Supervisor: Dr. Mariano Rodriguez-Muro

Bolzano, 2012

To my beloved wife...

Contents

Abstract

Nowadays, the use of ontologies in many application domains has risen quickly. One of the most important aspects of using ontologies in practice is Ontology Based Data Access, i.e., querying large volumes of data while exploiting the semantics expressed in the ontology. There are two approaches for solving this problem, one possibility is to chase the data w.r.t. the ontology, however this process can be costly and it depending on the ontology language could not be applicable; a second possibility is to rewrite the query w.r.t. the ontology. The second approach has been extensively studied in the last years. Several ontology language families have been proposed that aim at providing efficient query answering in the presence of large amounts of data. One of these is the DL-Lite family that allows for the generation of simple query rewritings, i.e., Union of Conjunctive Queries. This family has been studied extensively and there is concrete evidence that good performance is achievable in practice. A second family of languages is the EL family. These languages allow to express recursion in the ontology and hence, query rewritings in this case are recursive Datalog programs. However, how to evaluate these programs in the presence of large amounts of data has never been studied, and of course, the performance of this kind of rewritings in practice is unknown.

The objective of this thesis is to extend knowledge of this topic. Some experiments with different ways of dealing with large volumes of data in ELHI ontologies will be done. We will report on the performance of rewriting techniques using different evaluation mechanisms, including traditional Datalog engines and the recursive features of SQL. In the end, we will provide a clear picture on the reach and limits of recursive query rewritings for Ontology Based Data Access. This data will be relevant for the implementation of OBDA query answering systems based on ELHI and any other any language that allows for recursively, e.g., different fragments of SWRL and other semantic web oriented languages.

Acknowledgements

I would like to thanks my supervisor, Dr. Mariano Rodriguez-Muro, who has been supporting me during the fulfillment of this thesis, also to go through several difficult situations I had before, like the changing of the topic, starting all over again, and problem with the visa to get back to Italy. Also I would like to say thank you to Prof. Diego Calvanese, who made a great input to this thesis, and also found out that the problem I would like to solve before is actually impossible to solve.

My gratitude also goes to the administrative staffs of Free University of Bozen-Bolzano, Ms. Veith Isolde and Ms. Federica Cumer, who have been so much patience answering my question regarding the defense. Without whom I might be lost in the execution of the thesis.

Also I would like to say my thanks to my house-mates while I was living in the apartment for the help and for being the same "crusaders" to get finishing the defense. Lilian, my house-mate for 6 months, also Ario and Rahmad, my two house-mates for quite long while I was in via Cappuccini 2, also my two best friends here. Thanks for all the input to the thesis, not forgetting all the laugh, the time to discuss together, whether it is important or not so important, and so on.

To all my friends who always asked "how is the progress of your thesis?", thank you very much. Marie, Mia, Hendro, Anton, Kosumo, Topan, Nabil, Evgeny, Josef, Ognjen, Héctor, and all my friend here that I have not stated. Without that 'annoying' question, I would not be encouraged to finish the thesis as soon as possible.

Of course this would apply to my family everywhere. My parents, who were being left by their kids and living in the home just the two of them. My brother Nanda, Habib, and Fahmi that are in different countries nowadays, also my grand-mas, my uncles, aunts, cousins, and my two little nephwes. Last but not least, to Dyah Saptanti Perwitasari, who was my friend for quite long enough when we were in bachelor degree, and now is my beloved wife, thank you so much for all your support and patience for my behaviour. I cannot thank you enough my family.

Alhamdulillah, thank you Allah for giving me a chance finishing this thesis.

Chapter 1

Introduction

Query rewriting has been an interesting topic on answering a query in Description Logics, since it can create a more effective answering. Several methods have been developed, such as the using of view as rewriting proposed by Beeri et al. (1997), the using of Relational Database System to rewrite ABox proposed by Lutz et al. (2009), *PerfectRef* proposed by Calvanese et al. (2007), a resolution-based rewriting proposed by Rosati (2007), and another rewriting technique based on resolution, $\mathcal{R}^{\mathcal{DL}}$, proposed by Pérez-Urbina et al. (2010), Pérez-Urbina et al. (2008a,b, 2009a).

However, there are only few of them that have tried to be evaluated using a Datalog engine or DBMS, to measure the performance of those query rewritings in the real world.

One of the query rewriting covering quite many languages and has been implemented is $\mathcal{R}^{\mathcal{DL}}$, since it has been proven to work on three different families of different logics, namely $\mathcal{EL}, \mathcal{ELH}, \mathcal{ELHI}$, *DL-Lite*$^+$, and *DL-Lite*$_\mathcal{R}$. On the first three families, we will be focusing only in the third one, since it is more expressive than the two formers.

Here, we would like to see how effective the implementation of this $\mathcal{R}^{\mathcal{DL}}$, which is REQUIEM, when they are evaluated on several Datalog engines or DBMS. We would like to cover the possibility of using several engines because one can be faster to the others for several cases, vice versa.

1.1 Structure of Thesis

Here, we will describe the structure of the document. In the Chapter 1, this chapter, we will give an introduction to explain what this document is about, what our motivation to create this document, and how we structure the document.

The Chapter 2 describes generally about the three families of description logics, which are *DL-Lite*$_{\mathcal{R}}$, $\mathcal{ELHI}$, and *DL-Lite*$^{+}$. It will not be explained everything about those three languages, since what we need here is just the brief knowledge about the three families, and the differences between the families, including the difference of answering on each of them and the limitation, based on the syntaxes accepted.

The Chapter 3 describes generally about the query rewriting we use, the reason why we use this query rewriting, the basis of this query rewriting, the system implementation this query, and the result of doing the query rewriting, respectively, for each language we explain in the Chapter 2.

The Chapter 4 describes generally about the engines we use to do the evaluation on query rewriting described on Chapter 3. In this chapter, we explain also the reason why we choose one engine over the another. There is another thing we explain in this chapter, which is how the execution is done for one kind of the engine, since the syntax of the produced result by the query rewriter is more similar to those in one of the kind of the engine, we will need to translate them to the other kind of engine to make the produced result executable. Beside, we will also explain how we will store the whole knowledge base inside the engine.

The Chapter 5 is the main chapter of the document. Here we will describe the way we evaluate the produced result, including how we obtain the ontology, how we create a "representative" query, and also generate the dummy data to be queried over. Also, we will show the result after we do the whole mechanism in this chapter, and we will do the comparison between the result of the evaluation on the different engines.

The last chapter, Chapter 6 is the conclusion of the document. We will put what we get from Chapter 5 and give our conclusion here. Also, we will put what can be the future work that is related to the result of this document.

Chapter 2

Description Logics

In this Chapter, we will present several families of description logics that cover several area in ontology. Moreover the query given on these three families, later, can be rewritten using the $\mathcal{R}^{\mathcal{DL}}$ that will be evaluated using several Datalog engines and RDBMSes.

2.1 DL-Lite$_{\mathcal{R}}$

This language is proposed by Calvanese et al. (2007). The *DL-Lite* family is a family of DLs specifically tailored to capture basic ontology languages, conceptual data models, e.g., Entity-Relationship, and object-oriented formalisms, e.g., basic UML class diagrams, while keeping the complexity of reasoning low. In particular, ontology satisfiability, instance checking, and answering conjuctive queries in these logics can all be done in LogSpace with respect to data complexity (Poggi et al., 2008). Here, we use one family of *DL-Lite*, which is *DL-Lite*$_{\mathcal{R}}$. In this thesis, most of the definition of *DL-Lite*$_{\mathcal{R}}$ taken from paper by Calvanese et al. (2009).

As mentioned in *Description Logics (DLs)* (Baader et al., 2003), *concepts*, which denote classes of objects, and *roles* (i.e., binary relationships), which denote binary relations between objects can model the domain of interest. Like any other logic, *DL-Lite*$_{\mathcal{R}}$ expressions are built over an alphabet, which comprises symbols for atomic concepts, atomic roles, and constants. Here, we use Γ to denote the alphabet of constants symbols for *objects*.

The complete notation of the language is as follow:

1. A denotes an *atomic concept*, B a *basic concept*, C a *general concept*. An atomic concept is a concept denoted by a name. The syntax of basic and general concepts will be described after.

2. P denotes an *atomic role*, Q a *basic role*, and R a *general role*. An atomic role is denoted by a name. The syntax of basic and general role will be described after.

The description of *DL-Lite*$_{\mathcal{R}}$ expressions is as follow:

1. Basic and general concepts have the syntax as follow:

$$\begin{array}{rcl} B & \rightarrow & A \mid \exists Q \\ C & \rightarrow & \top \mid B \mid \neg B \mid \exists Q.C \end{array}$$

   where $\neg B$ denotes the *negation* of a basic concept B. The concept $\exists Q$, also called *unqualified existential restriction*, denotes the *domain* of a role Q, which is the set of objects that Q relates to some object. The concept $\exists Q.C$, also called *qualified existential restriction*, denotes the *qualified domain* of Q w.r.t. C, i.e. the set of objects that Q relates to some instance of C.

2. Role expressions have the syntax as follow:

$$\begin{array}{rcl} Q & \rightarrow & P \mid P^- \\ R & \rightarrow & Q \mid \neg Q \end{array}$$

   where P^- denotes the *inverse* of an atomic role, and $\neg Q$ denotes the *negation* of a basic role.

DL-Lite$_{\mathcal{R}}$ *ontology*, or knowledge base(KB), like any other description logic families, is constituted by two components:

- a *TBox*, 'T' stands for *terminological*, which is a finite set of *intensional assertions*, and
- an *ABox*, 'A' stands for *assertional*, which is a finite set of *extensional assertions*.

In *DL-Lite*$_{\mathcal{R}}$, the TBox may contain intensional assertions as follows:

$$B \sqsubseteq C, \qquad\qquad Q \sqsubseteq R,$$

denoting respectively, inclusion between concepts and roles. It is intuitive to know that inclusion means, in every model of $\mathcal{T}$, each instance of the left-hand side expression is also an instance of the right-hand side expression.

An inclusion assertion that has '$\neg$' on the right side is called a *negative inclusion* (NI), while the one that does not have it is called a *positive inclusion* (PI).

Example 1. *The inclusion* Parent $\sqsubseteq$ $\exists$HAS-CHILD *defines that every parent must have a child, the inclusion* HAS-PARENT $\sqsubseteq$ HAS-ANCESTOR *defines that a parent of someone is also an ancestor of that person. The negative inclusion* Man $\sqsubseteq$ $\neg$Woman *defines that men and women are disjoint.*

Then, a *DL-Lite*$_\mathcal{R}$ TBox is a finite sets of intensional assertions of the two forms. On the other hand, a *DL-Lite*$_\mathcal{R}$ ABox consists of a set of *membership assertions*, where are used to state the instanece of concepts and roles. Such assertion have the form

$$A(a), \qquad\qquad P(a_1, a_2),$$

where A is an atomic concept, P is an atomic role, a, a_1, a_2 are constants in Γ.

Definition 1 (*DL-Lite*$_\mathcal{R}$ ontology)**.** *A DL-Lite*$_\mathcal{R}$ *ontology* $\mathcal{O}$ *is a pair* $\langle \mathcal{T}, \mathcal{A} \rangle$*, where* $\mathcal{T}$ *is a DL-Lite*$_\mathcal{R}$ *TBox and* $\mathcal{A}$ *is a DL-Lite*$_\mathcal{R}$ *ABox, all of whose atomic concepts and roles occur in* $\mathcal{T}$*.*

2.2 $\mathcal{ELHI}$

In this Section, we will briefly tell about $\mathcal{ELHI}$, which is another family of Description Logics. We would mostly take the material of this Section from papers explaining $\mathcal{EL}^{++}$ and $\mathcal{ELH}$ (Brandt, 2004, Baader et al., 2005, Pérez-Urbina et al., 2008a).

It has been shown that the complexity of query answering for the DL $\mathcal{ELHI}$ that subsumption and instance problem w.r.t. cyclic terminologies can be decided in polynomial time. It is ranging from LogSpace to PTime-complete w.r.t. the data complexity of answering query given.

Concept descriptions are inductively defined with the help of a set of concept constructors, starting with a set N_{con} of concept names and a set N_{role} of role names. The DL $\mathcal{ELHI}$ provides the concept constructors top-concept ($\top$), conjunction ($C \sqcap D$), and existential restrictions ($\exists R.C$).

Like *DL-Lite*$_\mathcal{R}$, $\mathcal{ELHI}$ consists of concepts and roles. The complete notation of the language is as follow:

1. A denotes an atomic concept, and B a basic concept. The syntax of atomic concept is exactly the same with one in *DL-Lite*$_\mathcal{R}$, while the syntax of basic concept will be described after.

2. P denotes an atomic role, and R a basic role. The syntax of atomic role is exactly the same with one in *DL-Lite*$_\mathcal{R}$, while the syntax of basic role will be described after.

Here, we use the normalized $\mathcal{ELHI}$ TBox as described by Brandt (2004), then we have the description of $\mathcal{ELHI}$ expressions as follow:

1. Basic concept has the syntax as follow:

$$B \to A \mid \exists R \mid \exists R.A \mid B_1 \sqcap B_2$$

   The concept $\exists R$ denotes the domain of a role R, the concept $\exists R.A$ denotes the domain of a role R of which the range is some instance of A. Then, the concept $B_1 \sqcap B_2$ denotes the object that is belong to both B_1 and B_2. In the end, $\exists R$ can be omitted since it is equal to $\exists R.\top$.

2. Basic role has the syntax as follow:

$$R \quad \to \quad P \mid P^-$$

   where P^- denotes the inverse of an atomic role.

Just like *DL-Lite*$_{\mathcal{R}}$ and any other description logic families, $\mathcal{ELHI}$ ontology is constituted by TBox and ABox. The $\mathcal{ELHI}$ ABox consists of a set of membership assertions, which are assertions of concepts or roles.

On the other hand, the $\mathcal{ELHI}$ TBox is slightly different. There are two possible subsumption on $\mathcal{ELHI}$, which are

$$B_1 \sqsubseteq B_2, \qquad\qquad R_1 \sqsubseteq R_2,$$

denoting respectively, inclusion between concepts and roles. One main difference with *DL-Lite*$_{\mathcal{R}}$ is, there is no difference between instances that can appear on the left-hand side of the expression with ones that can appear on the right-hand side.

From here, we can easily see that the qualified existential restriction can appear both in the left-hand side and right-hand side, which is one thing that is more expressive than *DL-Lite*$_{\mathcal{R}}$. On the other hand, there is no possibility of having negation on the subsumption (non-existential).

Examples of $\mathcal{ELHI}$ TBoxes that cannot be expressed with *DL-Lite*$_{\mathcal{R}}$ is shown in Example 2.

Example 2. *The inclusion* HAS-TEACHING.Course $\sqsubseteq$ Lecturer *means that everyone teaching a course is a lecturer. On the other hand,* Student $\sqcap$ Researcher $\sqsubseteq$ PhDStudent *tells us that every student that is also a researcher is a PhD student.*

2.3 DL-Lite$^+$

In this section, we will briefly tell about *DL-Lite*$^+$, which is one another family of Description Logics. We would mostly take the material of this Section from papers explaining DL-Lite (Calvanese et al., 2007).

DL-Lite$^+$ can be seen as the Description Logic family between *DL-Lite*$_\mathcal{R}$ and $\mathcal{ELHI}$. It is strictly less expressive than $\mathcal{ELHI}$, because there is no possibility of adding disjunctive in *DL-Lite*$^+$, it is uncomparable with *DL-Lite*$_\mathcal{R}$, since there is a feature added in *DL-Lite*$^+$ that is not exist in *DL-Lite*$_\mathcal{R}$, vice versa. However, it has been shown that *DL-Lite*$^+$ is more difficult than *DL-Lite*$_\mathcal{R}$, since the complexity of query answering in *DL-Lite*$^+$ is NLogSpace.

Just like *DL-Lite*$^+$ and $\mathcal{ELHI}$, the syntax of *DL-Lite*$^+$ also consists of two main elements, which are concepts and roles. The complete notation of *DL-Lite*$^+$ language is as follow:

1. A denotes an atomic concept, B a basis concept. The syntax of atomic concept is exactly the same with ones in *DL-Lite*$_\mathcal{R}$ and $\mathcal{ELHI}$, while the syntax of basic concept will be described after.

2. P denotes an atomic role.

The description of *DL-Lite*$^+$ expressions is as follow:

1. Basic concept has the syntax as follow

$$B \to A \mid \exists P \mid \exists P.A$$

   The concept $\exists P$ denotes the domain of a role P, the concept $\exists P.A$ denotes the domain of a role P of which the range is some instance of a concept A. In the end, we can easily omit $\exists P$ since it was a special case of $\exists P.A$ where A is equal to $\top$.

Then, same as before, *DL-Lite*$^+$ ontology is constituted by TBox and ABox. The *DL-Lite*$^+$ ABox consists of a set of membership assertions, which are also assertions of concepts or roles.

On the other hand, the *DL-Lite*$^+$ TBox can be differed into two possible subsumption, which are

$$B_1 \sqsubseteq B_2, \qquad\qquad P_1 \sqsubseteq P_2,$$

denoting respectively, inclusion between concepts and roles. Like $\mathcal{ELHI}$, there is no difference between instances that can appear on the left-hand side of the expression with ones that can appear on the right-hand side.

Note that, since the subsumption in $\mathcal{ELHI}$ is more general than one in *DL-Lite*$^+$, it is easy to see that $\mathcal{ELHI}$ is strictly more expressive than *DL-Lite*$^+$. The first TBox example in 2 can also be expressed by *DL-Lite*$^+$, while the second one is impossible since there is a disjunction on the left-hand side of the TBox.

Chapter 3

Query Rewriting

In this Chapter, we will explain more detail on query rewriting we are going to use for the evaluation, which is $\mathcal{R}^{\mathcal{DL}}$. The query rewriting itself has been implemented, named REQUIEM[*]. Given a query, this system will return the datalog query, union of conjunctive queries, or union of conjunctive queries and a linear datalog program, depends of in which family the Description Logics will be given, either in $\mathcal{ELHI}$, *DL-Lite*$_{\mathcal{R}}$, or *DL-Lite*$^{+}$.

Generally, there are two ways of answering a query. One, by trying every possible interpretation, and check whether the given interpretation is the model of the query w.r.t. the knowledge base and the database. This idea is not very efficient, since we may have a lot of interpretations. Hence, the second way to get the answer of a query, which is query rewriting over the knowledge base is commonly used to answer a query.

Query rewriting (or query reformulation) *rew* of a query Q is defined as follows. Given TBox $\mathcal{T}$, ABox $\mathcal{A}$. Defined *tr* as translation function from the knowledge base to the clause. Here, denoted $\mathcal{KB}$ as $\mathcal{T} \cup \mathcal{A}$. The query rewriting *rew* has to make $ans\,(Q, \mathcal{KB}) = ans\,(Q, rew\,(\mathcal{T}) \cup \mathcal{A})$. The idea of query reformulation is not to create the whole possible knowledge base when answering a query, since it is oftenly possible that knowledge base needed to answer a query is much smaller than the whole knowledge base.

3.1 Datalog

Datalog is the language used for deductive database system. It is similar to Prolog, where every clause must be a *horn clause*, i.e. conjunctive of literal with

[*]http://www.cs.ox.ac.uk/projects/requiem/home.html

at most one positive literal. The main reason why Datalog exists is because the need of combinating logic and database (Gallaire et al., 1984).

One difference between Datalog and Prolog is one cannot put a complex predicate, every predicate must consist of one or zero variable or constant, or in other words, no function term is allowed as an argument of the predicate in Datalog.

Datalog consists of head and rule, the syntax is as follow.

$$Body_1, \ldots, Body_n \rightarrow Head.$$

In case $n = 0$, it is called a fact, or *extensional database (EDB)*, otherwise it is a rule, or *intentional database (IDB)*. The definition of the syntax is, whenever $Body_1, \ldots, Body_n$ are all true, $Head$ will also be true. We may also say that fact is actually a subset of rule, where there is no body atom exists.

The EDB predicates value are given via an input database, while the IDB predicates value are computed by the program. In standard Datalog, EDB predicate symbols may only appear in Body (Calì et al., 2010). In other words, those predicates appearing in head of a non-zero predicate in the body are all IDBs.

We then define Datalog program as the set of rules having syntax as written above, note that the term rules here cover both rule as fact, where there is no body atom, and the "real" rule.

Regarding the complexity, the general Datalog program has a tractable complexity, w.r.t data complexity, which is PTime-complete (Dantsin et al., 2001, Gottlob and Papadimitriou, 2003), while its program and combined complexities are both ExpTime-complete.

3.1.1 Linear Datalog

One interesting fragment of datalog is linear Datalog. Linear Datalog programs are programs whose clauses have at most one intensional atom in their bodies (Calì et al., 2010, Afrati et al., 2003). This fragment of Datalog is interesting because it is a Datalog program of which the proof tree is basically a path.

Moreover, problem that is solvable by linear Datalog has quite interesting complexity, which is non-deterministic logarithmic space (NLogSpace-complete) w.r.t data complexity, and it is well-known that NLogSpace $\subseteq$ PTime, although it is still an open problem whether the left hand side is strictly contained or not on the other. On the other hand, the program and combined complexities are both PSpace-complete

3.1.2 Query

A query is an open formula of first order logic denotes as $\{\overrightarrow{x} \mid \phi(\overrightarrow{x})\}$ where $\phi(\overrightarrow{x})$ is a FOL-formula with free variables $\overrightarrow{x}$. Given an interpretation $\mathcal{I}$, $q^{\mathcal{I}}$ is the set of tuples of domains elements that, when assigned to the free variables, make the formula ϕ true in $\mathcal{I}$ (Calvanese et al., 2007).

A conjunctive query ($\mathcal{CQ}$) q is a query of the form $\{\overrightarrow{x} \mid \exists\overrightarrow{y}.conj(\overrightarrow{x}, \overrightarrow{y})\}$, where $conj(\overrightarrow{x}, \overrightarrow{y})$ is a conjunction of atoms, with free variables $\overrightarrow{x}$ and $\overrightarrow{y}$. Moreover, a *union of conjunctive queries* ($\mathcal{UCQ}$) q is a query of the form

$$\left\{\overrightarrow{x} \mid \bigvee_{i=1,\ldots,n} \exists\overrightarrow{y_i}.conj_i(\overrightarrow{x}, \overrightarrow{y_i})\right\}$$

.

We can also use the standard datalog notation to denote these queries. For example, a conjunctive query $q = \{\overrightarrow{x} \mid \exists\overrightarrow{y}.conj(\overrightarrow{x}, \overrightarrow{y})\}$ is denoted in datalog notation as $q(\overrightarrow{x}) \leftarrow conj(\overrightarrow{x}, \overrightarrow{y})$. Moreover, a union of conjunctive query with the form written above can be written as $q = \{q_1, \ldots, q_n\}$ (Calvanese et al., 2007).

Given a query q and a KB $\mathcal{K}$, the answer to q over $\mathcal{K}$ is the set $ans(q, \mathcal{K})$ of tuples $\overrightarrow{a}$ of constants appearing in $\mathcal{K}$ such that $\overrightarrow{a}^{\mathcal{M}} \in q^{\mathcal{M}}$, for every model $\mathcal{M}$ of $\mathcal{K}$ (Calvanese et al., 2007).

Later on, this is actually just the same as the Datalog, which is described in 3.1, we can denote the query as $H \leftarrow B_1, \ldots B_n$, this is done just to easily separate each atom in the body of the query.

3.2 Resolution-Based Query Rewriting

As mentioned above, our evaluation will be done on $\mathcal{R}^{\mathcal{DL}}$ which is based on resolution. Since there are several $\mathcal{R}^{\mathcal{DL}}$ used in our query rewriting, one for each family of description logics we use, we will have to have the translation DL-axioms for each family, respectively to the clause used in the resolution.

Here, we use $var(C)$ to denote the number of variable in clause C. Then, *depth* of a term is defined as follow: (i) $depth(t) = 0$ if t is either constant or variable, (ii) $depth(f(t_1, \ldots, t_n)) = 1 + max(depth(t_1), \ldots, depth(t_n))$. The *depth* of an atom is defined as $depth(P(t_1, \ldots, t_m)) = max(depth(t_1), \ldots, depth(t_m))$ while the *depth* of a Horn clause $C = B_1, \ldots, B_k \rightarrow H$ is defined as $depth(C) = max(depth(H), depth(B_1), \ldots, depth(B_k))$.

Table 3.1, 3.2, and 3.3 show the correspondency, respectively, between *DL-Lite*$^+$ axiom and *DL-Lite*$^+$ clause, as given in Pérez-Urbina et al. (2008b), between $\mathcal{ELHI}$ axiom and $\mathcal{ELHI}$ clause, as given in Pérez-Urbina et al. (2010) and Pérez-Urbina et al. (2008a), and between *DL-Lite*$_{\mathcal{R}}$ axiom and *DL-Lite*$_{\mathcal{R}}$ clause, as given in Pérez-Urbina et al. (2009a).

3.3 Query Rewriting on Description Logics

The query rewriting used by $\mathcal{R}^{\mathcal{DL}}$ is based on Resolution with Free Selection (RFS). Resolution with Free Selection is a well-known calculus that can be used

DL-Lite$^+$ Clause	*DL-Lite*$^+$ Axiom
$A(a)$	$A(a)$
$P(a,b)$	$P(a,b)$
$A_2(x) \leftarrow A_1(x)$	$A_1 \sqsubseteq A_2$
$P(x,f(x)) \leftarrow A_1(x)$	$A_1 \sqsubseteq \exists P.A_2$
$A_2(f(x)) \leftarrow A_1(x)$	
$A(x) \leftarrow P(x,y)$	$\exists P \sqsubseteq A$
$A_2(x) \leftarrow P(x,y), A_1(y)$	$\exists P.A_1 \sqsubseteq A_2$
$P_2(x,y) \leftarrow P_1(x,y)$	$P_1 \sqsubseteq P_2$
$A_2(b) \leftarrow A_1(a)$	
$A_3(x) \leftarrow A_2(f(x)), A_1(x)$	
$Q_P(\mathbf{t}) \leftarrow \mathbf{L}(\mathbf{t_i})$	

Table 3.1: Correspondency between *DL-Lite*$^+$ clause with *DL-Lite*$^+$ axiom

Note: A, A_1, A_2, and A_3 are atomic concepts, while P, P_1, and P_2 are atomic roles, Q_P is a query, and $\mathbf{L}$ is one or more role or concepts, or both. For the arguments, x and y are variables, a and b are constants, $\mathbf{t}$ can be zero to finitely many arguments, while $\mathbf{t_i}$ can be either one or two arguments, only. Each function in clause T2 $A_1 \sqsubseteq \exists P.A_2$ is unique. For each clause C of type Q1, (i) $var(C) \leq var(Q_C)$, (ii) $depth(C) \leq max(1, var(Q_C) - var(C))$, and (iii) if a variable x occurs in a functional term in C, x occurs in all functional terms in C.

to decide satisfiability on a set of Horn clauses. The calculus is parameterized by a selection function S that assigns to each Horn clause C a nonempty set of atoms such that either $S(C) = \{H\}$ or $S(C) \subseteq \{Bi\}$. The atoms in $S(C)$ are said to be selected in C. The basis of the resolution is as follow:

$$\frac{A \leftarrow B_1 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_n \qquad C \leftarrow D_1 \wedge \ldots \wedge D_m}{A\sigma \leftarrow B_1\sigma \wedge \ldots \wedge B_{i-1}\sigma \wedge B_{i+1}\sigma \wedge \ldots \wedge B_n\sigma \wedge D_1\sigma \wedge \ldots D_n\sigma}$$

The two clauses above are premises, and the clauses below is the result, or can be called a resolvent. WLOG, we can assume that the two premises do not have variable in common. Also we have that $\sigma = MGU(B_i, C)$. Set of Horn clauses LP is saturated by R if for every $P_1, P_2 \in LP$, we have resolvent PR of P_1 and P_2, the set LP contains a clause equivalent to PR up to variable renaming. Clause C is derivable from LP by R if $C \in LP_R$. Moreover, WLOG, we can say that the two premises do not share the same variable, and if there is, we can easily rename it on one of the premise.

The complete resolution defined on $\mathcal{R}^{\mathcal{DL}}$ can be found on the papers given in Pérez-Urbina et al. (2008b), Pérez-Urbina et al. (2008a), and Rosati (2007).

$\mathcal{ELHI}$ Clause	$\mathcal{ELHI}$ Axiom
$A(a)$	$A(a)$
$P(a,b)$	$P(a,b)$
$A_2(x) \leftarrow A_1(x)$	$A_1 \sqsubseteq A_2$
$A_3(x) \leftarrow A_2(x), A_1(x)$	$A_1 \sqcap A_2 \sqsubseteq A_3$
$P(x, f(x)) \leftarrow A_1(x)$	$A_1 \sqsubseteq \exists P.A_2$
$A_2(f(x)) \leftarrow A_1(x)$	
$P(f(x), x) \leftarrow A_1(x)$	$A_1 \sqsubseteq \exists P^-.A_2$
$A_2(f(x)) \leftarrow A_1(x)$	
$A_2(x) \leftarrow P(x,y), A_1(y)$	$\exists P.A_1 \sqsubseteq A_2$
$A_2(x) \leftarrow P(y,x), A_1(y)$	$\exists P^-.A_1 \sqsubseteq A_2$
$P_2(x,y) \leftarrow P_1(x,y)$	$P_1 \sqsubseteq P_2$ or $P_1^- \sqsubseteq P_2^-$
$P_2(x,y) \leftarrow P_1(y,x)$	$P_1 \sqsubseteq P_2^-$ or $P_1^- \sqsubseteq P_2$
$Q_P(\mathbf{t}) \leftarrow \mathbf{L}(\mathbf{t_i})$	

Table 3.2: Correspondency between $\mathcal{ELHI}$ clause with $\mathcal{ELHI}$ axiom

Note: A, A_1, and A_2 are atomic concepts, while P, P_1, and P_2 are atomic roles, Q_P is a query, and $\mathbf{L}$ is one or more role or concepts, or both. For the arguments, x and y are variables, a and b are constants, $\mathbf{t}$ can be zero to finitely many arguments, while $\mathbf{t_i}$ can be either one or two arguments, only. Each function in clause T2 $A_1 \sqsubseteq \exists P.A_2$ is unique. For each clause C of type Q1, (i) $\mathit{var}(C) \leq \mathit{var}(Q_C)$, (ii) $\mathit{depth}(C) \leq \mathit{max}(1, \mathit{var}(Q_C) - \mathit{var}(C))$, and (iii) if a variable x occurs in a functional term in C, x occurs in all functional terms in C.

3.4 REQUIEM

REQUIEM (REsolution-based QUery rewrIting for Expressive Models) is the implementation of query rewriting technique $\mathcal{R}^{\mathcal{DL}}$ developed by Urbina et al. at the Oxford University Computing Laboratory.

3.4.1 Query Produced by REQUIEM

Given $\mathcal{ELHI}$ TBoxes, the query rewriting done by REQUIEM will produce datalog queries, given *DL-Lite*$_{\mathcal{R}}$ TBoxes, the query rewriting done by REQUIEM will produce a union of conjunctive query, and given *DL-Lite*$^+$ TBoxes, the query rewriting done by REQUIEM will produce a union of conjunctive query and a linear datalog. In this Subsection, we will explain a bit why the queries produced are as they are.

For query rewriting applied for these three families of description logics, we may say that $\mathcal{ELHI}$ is the most difficult among them, since, although there is a possibility of adding the negation on the right-hand side of intensional assertion

DL-Lite$_{\mathcal{R}}$ Clause	*DL-Lite*$_{\mathcal{R}}$ Axiom
$A(a)$	$A(a)$
$P(a,b)$	$P(a,b)$
$A_2(x) \leftarrow A_1(x)$	$A_1 \sqsubseteq A_2$
$P(x,f(x)) \leftarrow A(x)$	$A \sqsubseteq \exists P$
$P(x,f(x)) \leftarrow A_1(x)$	$A \sqsubseteq \exists P.A_1$
$A_2(f(x)) \leftarrow A_1(x)$	
$P(f(x),x) \leftarrow A(x)$	$A \sqsubseteq \exists P^-$
$P(f(x),x) \leftarrow A_1(x)$	$A \sqsubseteq \exists P^-.A_1$
$A_2(f(x)) \leftarrow A_1(x)$	
$A(x) \leftarrow P(x,y)$	$\exists P \sqsubseteq A$
$A(x) \leftarrow P(y,x)$	$\exists P^- \sqsubseteq A$
$P_2(x,y) \leftarrow P_1(x,y)$	$P_1 \sqsubseteq P_2$ or $P_1^- \sqsubseteq P_2^-$
$P_2(x,y) \leftarrow P_1(y,x)$	$P_1 \sqsubseteq P_2^-$ or $P_1^- \sqsubseteq P_2$
$Q_P(\mathbf{t}) \leftarrow \mathbf{L}(\mathbf{t_i})$	

Table 3.3: Correspondency between *DL-Lite*$_{\mathcal{R}}$ clause with *DL-Lite*$_{\mathcal{R}}$ axiom

Note: A, A_1, and A_2 are atomic concepts, while P, P_1, and P_2 are atomic roles, Q_P is a query, and $\mathbf{L}$ is one or more role or concepts, or both. For the arguments, x and y are variables, a and b are constants, $\mathbf{t}$ can be zero to finitely many arguments, while $\mathbf{t_i}$ can be either one or two arguments, only.

on *DL-Lite*$_{\mathcal{R}}$ or express the disjointness, which is not in $\mathcal{ELHI}$, it is not used to rewrite the query. Moreover, this kind of intensional assertion will only be used for consistency checking in the end.

Thus, for rewrite the query, we may say that *DL-Lite*$_{\mathcal{R}}$ is an $\mathcal{ELHI}$ without the possibility of adding qualified existential restriction on the left-hand side and no disjunction on the left-hand side, while disjunction on the right-hand side is easily constructed by two intensional assertions.

Then, *DL-Lite*$^+$ is an $\mathcal{ELHI}$ without the possibility of adding inverse role and no disjunction on the left-hand side on intensional assertion. While, same with *DL-Lite*$_{\mathcal{R}}$, adding disjunctioness on the right-hand side is easily constructed by two intensional assertions.

Chapter 4

Datalog Engines and RDBMSes Used

In this Chapter, we will describe how we evaluate the query rewriting done using the $\mathcal{R}^{\mathcal{DL}}$ by several datalog engines and relational DBMSes. We will also justify the choice of these engines. Beside, since the result of query rewriting using REQUIEM will not be a SQL, we will provide the detailed translation from each possible results of rewritten query into SQL, so that the results produced can be easily transformed into SQL and later, executed by relational DBMSes used.

4.1 Datalog Engine

In this Section, we will provide several datalog engines, including their capabilities and features. Moreover, we will also provide the reasons why these engines are chosen for the evaluation of the query rewriting $\mathcal{R}^{\mathcal{DL}}$.

4.1.1 XSB

In this Subsection, we will give a brief explanation about what XSB is, and why we choose this Datalog engine over other engines available. Most of the material here is taken from Swift et al. (2011), Sagonas et al. (1994).

XSB[*] is an in-memory deductive database engine, of which the implementation derives from Warren Abstract Machine. XSB began from a Prolog foundation, and traditional Prolog systems are known to have serious deficiencies when used as database systems. XSB has a fundamental bottom-up extension, introduced through tabling (or memoing), which makes it appropriate as an underlying query engine for deductive database systems.

[*]http://xsb.sourceforge.net/

This tabling system will eliminate the redundant computation on the same query, moreover, the existence of tabling will make XSB able to compute all modularly stratified datalog programs in polynomial time, w.r.t. data complexity.

Also, XSB offers an alternative approach to creating a deductive database system. Rather than depending on rewriting techniques, it extends Prolog SLD resolution in two ways: (i) adding tabling to make evaluations finite and non-redundant on datalog, and (ii) adding a scheduling strategy and delay mechanisms to treat general negation efficiently.

Unlike XSB, most of the datalog engines used an extension of the magic set technique (Bancilhon et al., 1986), which was quite well-known for answering query, that was executed using bottom-up approach, to a top-down approach, for several query given. This is one reason why we would like to use XSB, because it is different to another similar system, moreover we want to know how effective the tabling is on answering query for ontology, which has incomplete database.

4.1.2 $\mathcal{LDL}^{++}$

In this Subsection, we will give a brief explanation about what $\mathcal{LDL}^{++}$ is, the feature it has and why we choose this Datalog engine. Most of the material here is taken from Arni et al. (2003, 1993).

$\mathcal{LDL}^{++}$ [†] is a deductive database system, of which the design is inspired from its predecessor, $\mathcal{LDL}$ (Chimenti et al., 1990). This system is focused on advanced development from the previous one.

One feature that is developed in $\mathcal{LDL}^{++}$ is the possibility to create a functional constraint on the rule of the program. With this functional constraint (by the use of `choice`), user can limit the result, they expect to output. Example 3 gives a nice application how this feature can be useful.

Example 3. *Given facts as follow:*

```
student('JimBlack',ee,senior).
professor(ohm,ee).
professor(bell,ee).
```

and rule as follow:

```
advisor(S,P) :- student(S,Majr,Yr), professor(P,Majr)
```

Query `advisor('JimBlack',P)?` *will return two answers, which we do not want, because none of the student may have more than an advisor*[‡]*. The possibility of adding a* `choice` *here will limit this.*

[†]http://www.cs.ucla.edu/ldl/

[‡]This is not 100% true in the real life, but we use this just for our case

```
advisor(S,P) :- student(S,Majr,Yr), professor(P,Majr),
                choice((S),(P))
```

This means the functional dependency `S` $\rightarrow$ `P`*, where one* `S` *may only have one* `P` *as the result from the rule.*

Another feature of $\mathcal{LDL}^{++}$ is the implementation of intelligent backtracking. Imagine we have the execution tree, if we are failed on one branch, with intelligent backtrack we can "jump" from one branch of execution to more than one level of the root, just to avoid the obviously failed execution on other branches. Example 4 explains how this actually works.

Example 4. *Given query*

```
q(x,y) :- b1(x),p(x,y),b2(x).
```

Let us assume that the data we have is as follow

```
b1(A1). b1(A2). b2(A2).
p(A1,B1). p(A1,B2). p(A1,B3). p(A2,B1). p(A2,B2).
```

Without intelligent backtracking, after assigning `x` *with* `A1`*, it will assign* `y` *with* `B1`*, and fails in* `b2`*, the problem is, it operates backtracking, where* `y` *is changed into* `B2`*, and fails again in* `b2`*, and yet another backtrack, changing* `y` *into* `B3`*, fails again, and in the end changing* `x` *and success in this time. We can see that several backtracks are useless since the value of variable changed is not the cause of failure on the query.*

With intelligent backtracking, the first time it fails on `b2`*, it will "jump" and backtrack to* `b1`*, which is the cause of failure. On the much bigger database, this will be really a huge efficiency of time consumption on answering a query.*

On the other hand, which is the difference between $\mathcal{LDL}^{++}$ and XSB, the system in $\mathcal{LDL}^{++}$ is still using magic set technique (Bancilhon et al., 1986). Moreover, although it still uses the magic set, it has already implemented the more specialized methods for left-linear and right-linear rules.

The possibility of doing the intelligent backtracking, the ability of doing the specialized method for left-linear and right-linear rules, yet the difference of having no tabling are the reasons why we choose this system. We will need to evaluate how affecting these three things on answering a query for an ontology given to the system.

4.1.3 DLV

DLV, or a disjunctive datalog system is a deductive database system based on disjunctive logic programming. It was created by Italian and Austrian research team from University of Calabria and Vienna University of Technology.

The DLV system includes several front-ends to deal with specific applications on knowledge representation and reasoning, which are:

1. Support for inheritance. This makes DLV system more and more like a semi object oriented system rather than only a datalog engine.
2. K planning system.
3. SQL frontend.

One of the speciality of DLV system is that it can accept a disjunctive on the head part of a rule. One of the example is

```
a ˆ b :- c.
```

means, if `c` true, either `a` or `b` (or both) must be true.

Because of the possibility of having disjunction on the head part of the rule, there are several types of reasoning can be follow from DLV system, which are:

1. Brave Reasoning. Brave (or credulous) reasoning is a reasoning such that, if there is a possibility to give a true value to two or more possibility predicates or arguments, it will give true value to all of them, unless it is impossible to give such a valuation. Because of this reasoning, given a non-negative query and a non-negative program, brave reasoning will have the most possible complete answer.
2. Cautious Reasoning. Cautious reasoning is a reasoning such that, if there is a possibility to give a true value to two or more possibility predicates or arguments, it will be more cautious. Unless the system sure that one of them is really true, it will give an all-false valuation to those possibilities. One drawback of this reasoning is, there is not yet a capability to conclude that there must be a true value on a group of predicates or arguments. Example 5 gives us a good example on how this reasoning system is not fully true yet.

   Example 5. *Given program as follow:*

   ```
   a ˆ b :- c.
   d :- a.
   d :- b.
   ```

   This cautious system will not conclude that `d` *must be true, because it does not know whether* `a` *or* `b` *is true, although it knows that one of them must be true. On the other hand, brave reasoning may conclude that* `d` *must be true since it concludes that both* `a` *and* `b` *are true.*

Although brave reasoning in the end gives the right result as we want, but we do not want the fact that both `a` and `b` are true, still there is no reasoning in this DLV system that contains the ignorance about predicates `a` and `b`, but is sure about the correctness of `d`. Moreover, knowing both `a` and `b` true will give very huge chance to get the wrong result, if given a different example other than one used in 5.

The reason why we use this datalog engine as one of the engines to evaluate the queries is, although does not use, we want to know how much the algorithm to evaluate disjunctiveness can affect the speed and the memory usage.

4.2 Relational Database Management System

In this Section, we will provide several Relational DBMSes (RDBMSes), including their capabilities and features. Moreover, we will also provide the reasons why these DBMSes are chosen for the evaluation of the query rewriting $\mathcal{R}^{\mathcal{DL}}$.

4.2.1 PostgreSQL

PostgreSQL is an open-source and RDBMS that is maintained by PostgreSQL Global Development Group. It was firstly released on June, 1989, and now the latest stable version of this RDBMS is 9.0.4.

Compared to any other open-source RDBMS, it has arguably the most feature. Although actually we will not need them, one of the reason we choose PostgreSQL is this, and most of the standard SQL-99 are accepted in PostgreSQL. Beside, it is quite popular RDBMS, so we decide to use this RDBMS to represent the open-source RDBMS.

4.2.2 DB2

DB2 is a proprietary RDBMS owned by IBM. Firstly released on 1983. Note that, although it is a proprietary RDBMS, it has a free version, which is called DB2 Express-C. Although this version is not as complete as the complete DB2, but for our purpose to evaluate the query rewriting, DB2 Express-C is more than enough. According to survey that was held by DC[§], which is about Worldwide Database Management Systems 20092013 Forecast and 2008 Vendor Shares, it was ranked two on the DBMS marketing share, where the first position was for Oracle database, and the third was Microsoft SQL Server.

One of the reason we choose DB2 instead of Oracle is the syntax of DB2 is more standard, the standard is SQL-99, while on the other hand, Oracle database has its own syntax, which is PL/SQL, that is quite different with the standard.

[§]http://www.idc.com/getdoc.jsp?containerId=219232

Moreover, with Oracle, we will need to install its own JRE, which is more consumption on the memory and harddisk space. For these reasons, we choose DB2 to represent the proprietary RDBMS, although the one we will use is not the proprietary one, but the free version.

4.2.3 Apache Derby

The last RDBMS we will try to use for the evaluation of the query rewriting is Apache Derby. It was maintained by Apache and first time released on 2004. Now it has already version 10.6.1.0.

Apache Derby, unlike PostgreSQL and DB2, is an in-memory database. One of the drawback of this is, of course, the number of data can be stored is very limited. On the other hand, the speed of access is believed to be faster than ones that are stored on disk. One other difference between this database with the two formers is the language it was built is Java, while the first two are built in C and C++.

One of the other reason why we choose this RDBMS is that because the syntax it uses is exactly as IBM DB2 SQL Syntax. With this, it is easier to compare between those two, and moreover, since IBM DB2 SQL Syntax is really similar to the one that PostgreSQL has, it is also easy to compare, later, the performance between Derby and PostgreSQL for our evaluation.

4.3 Translation to SQL

In this Section, we will provide the detailed translation from the result of query rewriting, which can be either union of conjunctive query, linear datalog and union of conjunctive query, or datalog program. With this, we can execute the result produced by query rewriting on RDBMS we will use on our evaluation technique.

4.3.1 Execution of Union of Conjunctive Queries in RDBMS

Having a union of conjunctive query, we can easily translate this into SQL. Before going into detail, we will give a brief explanation about rectified datalog program. One method of rectification of the rules can be seen on 6

Definition 2. *The rules for predicate P are rectified when their heads are:*

1. *Identical, there is no variable difference between to the other*
2. *No constant argument*
3. *All variable are distinct*

Example 6. *Given a datalog program as follow:*

$$
\begin{aligned}
P(x,y) &\leftarrow Q(x,y)\\
P(x,z) &\leftarrow Q(x,y), Q(y,z)\\
P(x,x) &\leftarrow R(x)\\
P(x, \text{‘}a\text{’}) &\leftarrow Q(x,x)
\end{aligned}
$$

We may change those rules to a rectified ones without changing the meaning and the result once given a query. One of the possible rectified rules from the above rules is as follow:

$$
\begin{aligned}
P(x,y) &\leftarrow Q(x,y)\\
P(x,y) &\leftarrow Q(x,z), Q(z,y)\\
P(x,y) &\leftarrow R(x), x = y\\
P(x,y) &\leftarrow Q(x,x), y = \text{‘}a\text{’}
\end{aligned}
$$

It is easy to see that any datalog rules can be changed into a rectified ones. Say, if we have a non-identical argument in one of the rule, we may easily rename that rule, if we have a constant in the argument, we may change it into a variable and putting the equivalence between the variable and the constant in the body of the rule, and if we have two same variables on the arguments of the rule, we may change one of them, and put the equivalence between those two variables.

To make our translation easier, we will have to make every rules produced by query rewriting rectified.

One of the speciality of union of conjunctive query is that we do not need to do any more reasoning for each query. In the other words, we may execute each (conjunctive) query by doing a lookup-search on the table(s) and do the `join` between the shared variable(s).

Given a query $Q(x_1,\ldots,x_n) \leftarrow P_1(x_{1_1},\ldots,x_{m_1}),\ldots,P_k(x_{k_1},\ldots x_{k_m})$, let B_{x_i} the first position of x_i, the i-th argument of Q, in the body, w.r.t. the name of the predicate of the body, the execution of the query can be seen on Figure 4.1.

Here, we denote `execute` as the function that execute the SQL query, also well-known as a result set. Also, we will need to define what “minimal” pairs of position is.

Definition 3. *Given $X = \{x_1,\ldots,x_n\}$, we define the minimal pairs as $X_{min} = \{(x_{1,1},x_{2,1}),\ldots,(x_{1,k},x_{2,k})\}$ where*

- *for each i,j, we have $x_{i,j} \in X$*
- *for each x_i, x_j, we have either:*
  - *$(x_i,x_j) \in X_{min}$, or*

```
function executeCQinSQL
input: a conjunctive query Q
output: result of execution in SQL

ans <- selection clause on SQL
for each argument x in Q do
  ans <- ans + B(x)
ans <- ans + condition clause
for "minimal" pair position n and m, where the variable in n
    equal to the variable in m do
  ans <- ans + condition position n equal position m
return execute(ans)
```

Figure 4.1: Execution a Conjunctive Query in SQL

  - *there is a chain* $x_{k_1}, \ldots, x_{k_p}$ *where* $(x_i, x_{k_1}) \in X_{min}$ *or* $(x_{k_1}, x_i) \in X_{min}$*,* $(x_{k_1}, x_{k_2}) \in X_{min}$ *or* $(x_{k_2}, x_{k_1}) \in X_{min}$*, ...,* $(x_{k_p}, x_j) \in X_{min}$ *or* $(x_j, x_{k_p}) \in X_{min}$*.*

- *there is no* $X_{min2} \subset X_{min}$*, in which* X_{min2} *is minimal pairs of* X *too.*

After that, given union of conjunctive query UCQ Σ, where $\Sigma = \{Q_1, \ldots, Q_n\}$ and Q_i, for each i, is a query, the execution of Σ in SQL is as shown on Figure 4.2

```
function executeUCQinSQL
input: union of conjunctive queries UQ
output: result of execution in SQL

res <- empty result
for each query Q in UQ
  res <- union between execute(ans) and translateCQToSQL(Q)
return res
```

Figure 4.2: Execute a Union of Conjuntive Queries in SQL

With that execution, we have already defined the complete definition of execution from union of cunjunctive query using SQL query, and the execution can be done in RDBMS.

4.3.2 Execution of Linear Datalog and Datalog Query in RDBMS

Unlike translation from Union of Conjunctive Query which is simpler, a linear datalog and a general datalog may contain a (linear) recursivity inside it. With this, we have to make sure that for every atom inside, we have already had the whole data for the atom, so here we will generate the fixpoint for each atom asked, especially if there is a recursivity for the atom.

The complete execution function is as shown in 4.3, where the function to generate the fix point is as shown in 4.4.

```
function executeDatalogInSQL
input: Datalog Queries Q and Datalog Program DP
output: result of execution in SQL

res <- empty result
for each query Q1 in Q do
  for each atom P in Q1 do
    query <- generateFixPoint(P, DP)
  ans <- union between ans and query
return ans
```

Figure 4.3: Execution of a Datalog Query in SQL

```
function generateFixPoint
input: Atom A, Datalog Program DP
output: result of execution in SQL

res <- empty result
do
  ans <- executeUCQinSQL(DP[A])
  resbefore <- res
  res <- union between res and ans
while res <> resbefore
return res
```

Figure 4.4: Generation of the Fixpoint for Atom

Here, we denote `DP[A]` as rules in `DP` where predicate `A` appears as its head.

4.4 Storing the Knowledge Base

In this Section, we will briefly describe about the method we store the knowledge base. Here, there will be difference between storing on datalog engine and on RDBMS, because the syntax of datalog engine is more similar to the syntax of the result produced by query rewriting.

4.4.1 Storing the Knowledge Base on Datalog Engine

Storing knowledge base, i.e., the facts and rules, into datalog engines can be said as effortless since the fact and the rule are part of the datalog and datalog engines can easily process them as the query given.

4.4.2 Storing the Knowledge Base on RDBMS

Unlike on a Datalog engine, where we can store the knowledge base as it is, on RDBMS there might be several possible ways on how we store the knowledge base, especially on the different schema for it. For example, given facts (EDBs) $A/1$ and $P/2$, where N/k defines a predicate N having k arguments, at least there are three ways on storing it, which are (i) The 'normalized' one, where we create two tables, A and P, table A has a column and table P has two columns, (ii) The 'semi-normalized', where we still have two tables, A, and P, but each only has a column, and the arguments will be put together, separated by a separator, usually a comma, or (iii) The 'unnormalized', where we will only have one table, named table EDB, and everything is stored in a column, as it is written on the Datalog, note that these are only several possibilities and there are some other ways storing the knowledge base. Here, we are more interested on how the performance comparison between those, since the 'normal' must not be faster that the others. In other words, we want to know how influential the schema of the database to the performance of the query evaluation.

Chapter 5

Comparison of Query Evaluation

In this Chapter, we will do the complete evaluation over rewritten query, and use it to compare the efficiency of execution on the query rewriting for several Datalog engines and RDBMSes. Moreover, this will be related to which kind of description logic family is used, what the type of the query is, the schema of how we store the data, which can be different for RDBMS, and also what kind of ontology is used.

5.1 Evaluation Mechanism

In this section, we will explain the complete procedure, step by step, of our evaluation over the ontologies given and the query rewritten.

The whole evaluation was done on a notebook BenQ Joybook S42, using Microsoft Windows® 7, 32-bit as the Operating System, with Intel® Core™ 2 Duo CPU @ 2.26 GHz as the processor and having 3.00 GB of RAM.

5.1.1 Obtain the Ontology

Since the rewriting technique we use is $\mathcal{R}^{\mathcal{DL}}$ that has been implemented in REQUIEM, we mostly use the ontology that has been evaluated, for the query rewriting, in REQUIEM. Moreover, we will still need to use several other queries, since ones used in REQUIEM mainly used there to compare their rewriting technique, $\mathcal{R}^{\mathcal{DL}}$, with another rewriting technique proposed by Calvanese et al, which is PerfectRef (Calvanese et al., 2007), which is published as the result on their paper in Pérez-Urbina et al. (2009b), the ontologies can be seen in the Appendix A.

As we can see on the ontology, there are several several new auxiliary predicates `AUX`, the explanation of these variables is as follows. Let us see one of several knowledge base containing the auxiliary predicates, as shown in Figure 5.1.

$$\begin{aligned} AUX5(X) &\leftarrow Organization(Y), worksFor(X,Y) \\ Employee(X) &\leftarrow AUX5(X), Person(X) \end{aligned}$$

Figure 5.1: Existence of Auxiliary Predicate

This auxiliary predicates come because of the existence of null predicates. We can see those two knowledge base as follow. “If X, known as a person, works in Y and Y is an organization, then X is an employee”. We see that this kind of rule is impossible to in one rule, since in the normal datalog program, which may contains more than two predicates as the body, these rules can be rewritten as $Employee(X) \leftarrow Organization(Y), worksFor(X,Y), Person(X)$. Also we may also see how these rules are actually written on XML-format ontology, as shown in the Figure 5.2.

```
<owl:Class rdf:ID="Employee">
  <rdfs:label>Employee</rdfs:label>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Person" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#worksFor" />
      <owl:someValuesFrom>
        <owl:Class rdf:about="#Organization" />
      </owl:someValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Figure 5.2: Representation of Null Predicates on XML-Format Ontology

We can see that to create this kind of ontology, it is not enough to create one rule, since it would be an intersection between one class containing only one element, or predicate, to another class containing restriction of value on an argument.

Also, there are several functions inside the argument of the predicate, which are `f0, f1, f2, ...`. The explanation of these arguments is as follows. Let us see one of several knowledge base containing the auxiliary predicates, as shown in Figure 5.3.

This function inside the predicate of an argument comes because there is an dependency between one variable on the right-hand side (body) of a rule to one variable on the left-hand side (head) of the same rule, on two different rules. In this case, we can see that if there is a graduate student, he must take a course,

$$\begin{aligned} GraduateCourse(f0(X)) &\leftarrow GraduateStudent(X) \\ takesCourse(X, f0(X)) &\leftarrow GraduateStudent(X) \end{aligned}$$

Figure 5.3: Existence of Function as Argument

and that course must be a graduate course. The representation of these two rules in XML-Format Ontology can be seen on Figure 5.4.

```
<owl:Class rdf:ID="GraduateStudent">
  <rdfs:label>graduate student</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Person" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#takesCourse" />
      <owl:someValuesFrom>
        <owl:Class rdf:about="#GraduateCourse" />
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Figure 5.4: Representation of Variable Dependency in XML-Format Ontology

We can see that to create this kind of ontology, we need a restriction of a class, in this case `GraduateStudent` has to be restricted into two classes, which are `takesCourse` and `GraduateCourse`.

5.1.2 Create the Representative Query

Because we all already have several well known ontologies, as stated in Subsection 5.1.1, we also use the queries that stated there. To create a representative query, we change several queries because it has to contain all of the possibility rewriting technique we want to use. Moreover, this little adjustment needed because we want to cover all the possible Description Logics as written in Chapter 2. All the queries we use can be seen in the Appendix B.

5.1.3 Rewrite the Query

This is when we really use REQUIEM for our evaluation. Given ontology having format RDF/OWL and the query, this system will create result, depend to what the family of Description Logics the input ontology is, as stated in Subsection 3.4.1.

5.1.4 Generate the Data

We will use the data generator provided by SyGENiA[*] (Synthetic GENerator of instance Axioms). This tool for generating the data is created after the published paper about idea of making the random, complete, and "representative" data by Stoilos et al. in Stoilos et al. (2010). We will not write the data here or on the ontology since everytime we create the data, it will always change from one time to another time. To not make the process becomes too long, but to give a possibility of randomness, we make around 250-400 data for each ontology. The number can be different for one generation of the data to the other.

5.1.5 Configuration of the Engines

Although a lot of engines is used to evaluate the queries given, there is no changing needed to all of them. We just need to install them appropriately, given the database name correspondingly to ones we want to use, and use them as it is. Problem with this engines are, while there is no problem for the database engines, there are several problems with datalog engines since most of them can only be used with one or two specific operating system, and we have to configure them all, if we want to use them, not in the engines themselves, but on the codes. In this case, since we just make sure this program works on our computer, we do not yet create the program to make sure that all the process can work on the other

For the other configuration, beside using the default configuration as it is given by the engines, we also use the most precise result, e.g., in some engines it is possible that the result given are more than one, we will just use ones that give result most precisely, i.e, contained in all the results that can be given. If there is no such result, we can be sure that there is no result for the given query on our program.

5.2 Result of The Evaluation

We will point several aspects in which the query can be different in one engine to the others. We will separate each measurement by the result of the query produced after apply the query rewriting. Here, beside comparing the result of these 7 engines we use, we also want to know how the result if there is no query rewriting.

[*]http://code.google.com/p/sygenia/

We execute each query 5 times to each ontology and each engine, the result written on the C are the averages of the middle three, where we avoid the fastest and slowest result, the most consuming and least consuming memory, and we also round up the usage of the memory to the nearest natural number. For the memory usage, the method used to find how much the memory usage is by finding its highest point of memory usage, not the average of the memory usage along the process of the execution of the program.

5.2.1 Correctness

Here, we will explain the result obtained from the evaluation, using the correctness as the point of view. It is really necessary that this part of evaluation be the main consideration when using an engine for our knowledge base and also incomplete database, because given a wrong result, there is no use having the other criteria of evaluation true.

After all the executions of the queries on the program, we know that there is big differences between using query rewriting and not using query rewriting. All the results from query rewriting giving the right result, while on the other hand, there is no execution in program without query rewriting can be finished, since all got an error. This error occurs because there is no possibility, both in datalog engine, or other engine we create combining relational database and java program that can handle functional argument inside the predicate. Because of this, no program can be solved, unless if the query asked by the user contains no predicate that is related to any of the functional argument, so there must be no linking between the predicates asked in the query to any predicates that has functional argument, by any rules or chain of rules.

5.2.2 Time Measurement

Here, we will explain the result obtained from the evaluation, using time measurement as the point of view. Note that we put the time on the first line of the code and we put the end of time counter at the end of the code, so this time measurement not only measured those time needed to query rewriting, if there is any, but all time needed for reading and parsing the content of the files, writing into the files, generate the data, etc., are also included in this time measurement we use.

After all the executions of the queries on the program, we have quite difference time measurements for the results given. We may see that there are huge differences between the time needed to execute a query in datalog engines, which are much faster, than the time needed to execute a query in database engines, this is because our algorithm to execute the query on database engine is still quite straight-forward, in a quite modest and brute force way, without any optimization.

Moreover, we may say that the query rewriting is less effective for the queries that are less complicated, while on the other hand, it gives quite advantages on behalf of time usage when the query is quite complicated. These results can be seen on Figure 5.5 or the complete data can also be seen on Appendix C.

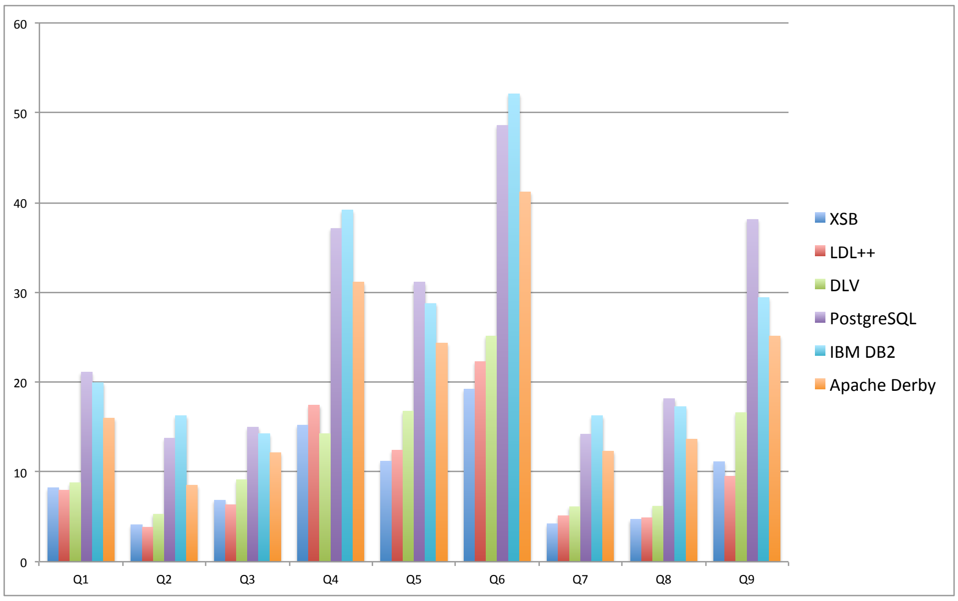

Figure 5.5: Evaluation Result of The Time Measurement

5.2.3 Memory Consumption

Here, we will explain the result obtained from the evaluation, using memory consumption as the point of view.

After all executions of the queries on the program, we are sure that there is no big difference on memory consumption among them all. To make sure that the result of memory consumption in a case was not just a coincidence result, we execute each queries several times on each engines and we make sure that there is no other software or process, except ones that have to be there, that is also running. As we can see on the Appendix C, the result was all about 10 to 50 MB. We can see from 5.6 and also from the appendix, since we cannot know how much of the process actually is used by each of the engine, we just measured the data on the consumption of memory by the Eclipse, which is the program we use to perform the evaluation to the queries given. Beside that, we may easily say that the memory usage in datalog engine is much less than one used by a database engine. This is probably because it is needed to run one or more software or service to execute a query using datalog engine, while on the other hand, the datalog engines all need less space and even they are really built for answering query that has form like deductive database.

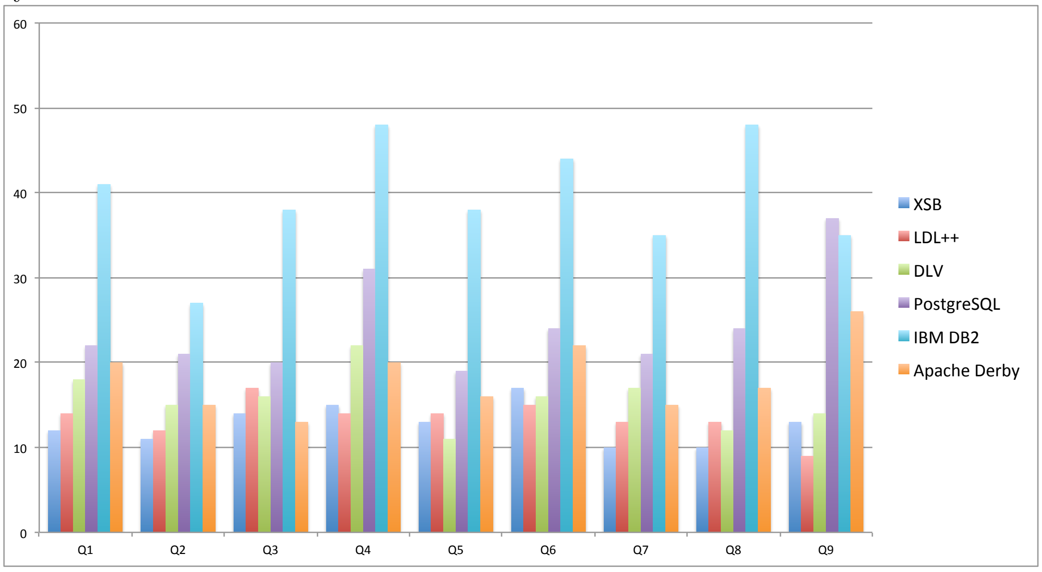

Figure 5.6: Evaluation Result of The Memory Consumption

Chapter 6

Conclusion and Future Work

In this section, we will write several conclusions regarding the system that have been built, also the results and how the result could be used later, and the future works that still related to this topic.

6.1 Conclusion

After seeing the program and the result of the evaluation, we may conclude several things, as follows:

1. Creating an engine for deductive database is not easy. We can see from the result, comparing the speed of result in datalog engines that have been there to other engines combining relational database and java program, the first engines work much faster, even more than two times faster than ones that we create.

2. Query rewriting makes us getting the answer. Without query rewriting, our program would not get the answer for the program. There is still no possible way to get answer to the query without using query rewriting.

3. Using small data, only 250-400, it is still faster to use in memory database, i.e, Apache Derby, than using real database relational, i.e, PostgreSQL or IBM DB2, this is probably because of the input output access time to memory database still needs much less time than doing the same thing to relational database.

4. If the data are big enough, using real relational database instead of memory database might be more useful, because the capability to manage all the data.

6.2 Future Work

Here, we will explain several future works that is related to this topic. Some future works that can be done regarding this thesis are as follows:

1. Creating a better, much more optimized query answering using relational database for storing the database. With this, it might possible that we get a faster result on answering using SQL and relational database than just use datalog engine, especially when we exploit all the capabilities of a relational database management system, which can handle large database, or using transactions to manage the possibility of changing or updating the data, or creating some views for some queries that are called a lot of time, so we can save some times.

2. If it is possible, answering the queries asked without using any query rewriting. This cannot be done if there is no other technique beside query rewriting. It is obvious that without doing any technique, i.e., no early processing on database and knowledge base, we will not get the result, but it might be possible to get the answer without adding the number of query we have to run on our program.

3. Comparing this query rewriting with another query rewriting. REQUIEM developers have claimed, and proven, that their algorithm is better than another query rewriting, i.e., Perfect Reformulation, in terms of number of the queries generated after doing all the process. However, we do not know whether the queries produced by this query rewriting are simpler than ones in Perfect Reformulation. It is a possibility that they have less number of query, while on the other hand, their queries are more complex than ones in other query rewriting technique.

Bibliography

Foto N. Afrati, Manolis Gergatsoulis, and Francesca Toni. Linearisability on Datalog Programs. *Theor. Comput. Sci.*, 308(1-3):199–226, 2003. [cited at p. 14]

Natraj Arni, KayLiang Ong, Shalom Tsur, and Carlo Zaniolo. LDL++: A Second Generation Deductive Databases Systems. Technical report, Technical report, MCC Corporation, 1993. [cited at p. 20]

Natraj Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. The Deductive Database System LDL++. *TPLP*, 3(1):61–94, 2003. [cited at p. 20]

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*, 2003. Cambridge University Press. ISBN 0-521-78176-0. [cited at p. 7]

Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the EL Envelope. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 364–369. Professional Book Center, 2005. ISBN 0938075934. [cited at p. 9]

François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15. ACM, 1986. ISBN 0-89791-179-2. [cited at p. 20, 21]

Catriel Beeri, Alon Y. Levy, and Marie-Christine Rousset. Rewriting Queries Using Views in Description Logics. In *PODS*, pages 99–108. ACM Press, 1997. ISBN 0-89791-910-6. [cited at p. 5]

Sebastian Brandt. On Subsumption and Instance Problem in ELH w.r.t. General TBoxes. In Volker Haarslev and Ralf Möller, editors, *Description Logics*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004. [cited at p. 9]

Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *LICS*, pages 228–242. IEEE Computer Society, 2010. ISBN 978-0-7695-4114-3. [cited at p. 14]

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable Reasoning and Efficient Query Answering in Description

Logics: The *DL-Lite* Family. *J. Autom. Reasoning*, 39(3):385–429, 2007. [cited at p. 5, 7, 10, 15, 29]

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. ISBN 978-3-642-03753-5. [cited at p. 7]

Danette Chimenti, Ruben Gamboa, Ravi Krishnamurthy, Shamim A. Naqvi, Shalom Tsur, and Carlo Zaniolo. The LDL System Prototype. *IEEE Trans. Knowl. Data Eng.*, 2(1):76–90, 1990. [cited at p. 20]

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Comput. Surv.*, 33(3):374–425, 2001. [cited at p. 14]

Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and Databases: A Deductive Approach. *ACM Comput. Surv.*, 16(2):153–185, 1984. [cited at p. 14]

Georg Gottlob and Christos H. Papadimitriou. On the Complexity of Single-rule Datalog Queries. *Inf. Comput.*, 183(1):104–122, 2003. [cited at p. 14]

Carsten Lutz, David Toman, and Frank Wolter. Conjunctive Query Answering in the Description Logic EL Using a Relational Database System. In Craig Boutilier, editor, *IJCAI*, pages 2070–2075, 2009. [cited at p. 5]

Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. Rewriting Conjunctive Queries under Description Logic Constraints. In Andrea Calì, Georg Gottlob, Laks V.S. Lakshmanan, and Davide Martinenghi, editors, *Proc. of the Int. Workshop on Logic in Databases (LID 2008)*, Rome, Italy, May 19–20 2008a. [cited at p. 5, 9, 15, 16]

Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. Rewriting Conjunctive Queries over Description Logic Knowledge Bases. In Klaus-Dieter Schewe and Bernhard Thalheim, editors, *SDKB*, volume 4925 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2008b. ISBN 978-3-540-88593-1. [cited at p. 5, 15, 16]

Héctor Pérez-Urbina, Ian Horrocks, and Boris Motik. Efficient Query Answering for OWL 2. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5823 of *Lecture Notes in Computer Science*, pages 489–504. Springer, 2009a. ISBN 978-3-642-04929-3. [cited at p. 5, 15]

Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. A Comparison of Query Rewriting Techniques for DL-Lite. In Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, and Ulrike Sattler, editors, *Description Logics*, volume 477 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009b. [cited at p. 29]

Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. Tractable Query Answering and Rewriting Under Description Logic Constraints. *J. Applied Logic*, 8(2):186–209, 2010. [cited at p. 5, 15]

Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking Data to Ontologies. *J. Data Semantics*, 10: 133–173, 2008. [cited at p. 7]

Riccardo Rosati. On Conjunctive Query Answering in EL. In Diego Calvanese, Enrico Franconi, Volker Haarslev, Domenico Lembo, Boris Motik, Anni-Yasmin Turhan, and Sergio Tessaris, editors, *Description Logics*, volume 250 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007. [cited at p. 5, 16]

Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as an Efficient Deductive Database Engine. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD Conference*, pages 442–453. ACM Press, 1994. [cited at p. 19]

Giorgos Stoilos, Bernardo Cuenca Grau, and Ian Horrocks. How Incomplete Is Your Semantic Web Reasoner? In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010. [cited at p. 32]

Terrance Swift, David S. Warren, Konstatinos Sagonas, Juliana Freire, Prasad Rao, Baoqiu Cui, Luis de Castro, Rui F. Marques, Diptikalyan Saha, Steve Dawson, and Michael Kifer. The XSB System Version 3.3: Programmer's Manual. *Web page: http://xsb.sourceforge.net*, 2011. [cited at p. 19]

Appendices

Appendix A

Knowledge Base Used

In this appendix, we give all the knowledge base we use. Note that there is no fixed database, since the database are all generated using the SyGENiA as stated on Subsection 5.1.4. We also do not use some data from the ontology given, if and only if those data are useless to our query answering program, e.g., data like `Professor is subclass of TopClass` can be ignored because everything is subclass of Topclass.

We use three ontologies, those are `university bench` ontology, `stock exchange`, and `path5`, each represents different complexity handled by the database. All the three ontologies can be found on the REQUIEM webpage.

The ontology for `university bench` is as follow:

$$\begin{aligned}
worksFor(X,Y) &\leftarrow headOf(X,Y)\\
degreeFrom(X,Y) &\leftarrow mastersDegreeFrom(X,Y)\\
memberOf(X,Y) &\leftarrow worksFor(X,Y)\\
degreeFrom(X,Y) &\leftarrow undergraduateDegreeFrom(X,Y)\\
member(Y,X) &\leftarrow memberOf(X,Y)\\
memberOf(Y,X) &\leftarrow member(X,Y)\\
degreeFrom(X,Y) &\leftarrow doctoralDegreeFrom(X,Y)\\
hasAlumnus(Y,X) &\leftarrow degreeFrom(X,Y)\\
degreeFrom(Y,X) &\leftarrow hasAlumnus(X,Y)\\
University(X) &\leftarrow mastersDegreeFrom(Y,X)\\
Organization(X) &\leftarrow affiliatedOrganizationOf(Y,X)\\
Work(X) &\leftarrow Research(X)\\
Article(X) &\leftarrow JournalArticle(X)\\
University(X) &\leftarrow undergraduateDegreeFrom(Y,X)
\end{aligned}$$

$$
\begin{aligned}
University(X) &\leftarrow degreeFrom(Y,X)\\
Course(X) &\leftarrow listedCourse(Y,X)\\
Person(X) &\leftarrow publicationAuthor(Y,X)\\
Research(X) &\leftarrow researchProject(Y,X)\\
Publication(X) &\leftarrow publicationResearch(X,Y)\\
Publication(X) &\leftarrow publicationDate(X,Y)\\
Organization(X) &\leftarrow University(X)\\
Schedule(X) &\leftarrow listedCourse(X,Y)\\
Person(X) &\leftarrow doctoralDegreeFrom(X,Y)\\
Person(X) &\leftarrow mastersDegreeFrom(X,Y)\\
Organization(X) &\leftarrow affiliateOf(X,Y)\\
AdministrativeStaff(X) &\leftarrow ClericalStaff(X)\\
Organization(X) &\leftarrow affiliatedOrganizationOf(X,Y)\\
GraduateCourse(f0(X)) &\leftarrow GraduateStudent(X)\\
takesCourse(X,f0(X)) &\leftarrow GraduateStudent(X)\\
TeachingAssistant(X) &\leftarrow teachingAssistantOf(X,Y)\\
Publication(X) &\leftarrow Specification(X)\\
Student(X) &\leftarrow ResearchAssistant(X)\\
Organization(X) &\leftarrow member(X,Y)\\
Research(X) &\leftarrow publicationResearch(Y,X)\\
AdministrativeStaff(X) &\leftarrow SystemsStaff(X)\\
Organization(X) &\leftarrow Department(X)\\
Person(X) &\leftarrow hasAlumnus(Y,X)\\
Software(X) &\leftarrow softwareVersion(X,Y)\\
ResearchGroup(X) &\leftarrow researchProject(X,Y)\\
Publication(X) &\leftarrow Manual(X)\\
Professor(X) &\leftarrow advisor(Y,X)\\
Faculty(X) &\leftarrow PostDoc(X)\\
Organization(X) &\leftarrow College(X)\\
Organization(X) &\leftarrow Institute(X)\\
Organization(X) &\leftarrow Program(X)\\
ResearchGroup(f1(X)) &\leftarrow ResearchAssistant(X)\\
worksFor(X,f1(X)) &\leftarrow ResearchAssistant(X)\\
AUX3(X) &\leftarrow College(Y), headOf(X,Y)\\
Dean(X) &\leftarrow AUX3(X), Person(X)
\end{aligned}
$$

$$
\begin{array}{rcl}
College(f2(X)) & \leftarrow & Dean(X) \\
headOf(X, f2(X)) & \leftarrow & Dean(X) \\
Person(X) & \leftarrow & Dean(X) \\
Faculty(X) & \leftarrow & Lecturer(X) \\
Person(X) & \leftarrow & GraduateStudent(X) \\
Course(X) & \leftarrow & teacherOf(Y, X) \\
Publication(X) & \leftarrow & UnofficialPublication(X) \\
Professor(X) & \leftarrow & tenured(X, Y) \\
Organization(X) & \leftarrow & orgPublication(X, Y) \\
Publication(X) & \leftarrow & publicationAuthor(X, Y) \\
Person(X) & \leftarrow & affiliateOf(Y, X) \\
Employee(X) & \leftarrow & AdministrativeStaff(X) \\
University(X) & \leftarrow & doctoralDegreeFrom(Y, X) \\
Software(X) & \leftarrow & softwareDocumentation(X, Y) \\
Person(X) & \leftarrow & advisor(X, Y) \\
Employee(X) & \leftarrow & Faculty(X) \\
Student(X) & \leftarrow & UndergraduateStudent(X) \\
Course(X) & \leftarrow & GraduateCourse(X) \\
AUX4(X) & \leftarrow & Course(Y), teachingAssistantOf(X, Y) \\
TeachingAssistant(X) & \leftarrow & AUX4(X), Person(X) \\
Course(f3(X)) & \leftarrow & TeachingAssistant(X) \\
teachingAssistantOf(X, f3(X)) & \leftarrow & TeachingAssistant(X) \\
Person(X) & \leftarrow & TeachingAssistant(X) \\
Work(X) & \leftarrow & Course(X) \\
Publication(X) & \leftarrow & Book(X) \\
Organization(X) & \leftarrow & ResearchGroup(X) \\
Professor(X) & \leftarrow & VisitingProfessor(X) \\
Faculty(X) & \leftarrow & teacherOf(X, Y) \\
Professor(X) & \leftarrow & Dean(X) \\
University(X) & \leftarrow & hasAlumnus(X, Y) \\
AUX5(X) & \leftarrow & Organization(Y), worksFor(X, Y) \\
Employee(X) & \leftarrow & AUX5(X), Person(X) \\
Organization(f4(X)) & \leftarrow & Employee(X) \\
worksFor(X, f4(X)) & \leftarrow & Employee(X) \\
Person(X) & \leftarrow & Employee(X)
\end{array}
$$

$$
\begin{array}{rcl}
AUX6(X) & \leftarrow & Course(Y), takesCourse(X,Y) \\
Student(X) & \leftarrow & AUX6(X), Person(X) \\
Course(f5(X)) & \leftarrow & Student(X) \\
takesCourse(X, f5(X)) & \leftarrow & Student(X) \\
Person(X) & \leftarrow & Student(X) \\
Person(X) & \leftarrow & degreeFrom(X,Y) \\
AUX7(X) & \leftarrow & Department(Y), headOf(X,Y) \\
Chair(X) & \leftarrow & AUX7(X), Person(X) \\
Department(f6(X)) & \leftarrow & Chair(X) \\
headOf(X, f6(X)) & \leftarrow & Chair(X) \\
Person(X) & \leftarrow & Chair(X) \\
Publication(X) & \leftarrow & softwareDocumentation(Y,X) \\
Faculty(X) & \leftarrow & Professor(X) \\
Person(X) & \leftarrow & undergraduateDegreeFrom(X,Y) \\
Course(X) & \leftarrow & teachingAssistantOf(Y,X) \\
Person(X) & \leftarrow & member(Y,X) \\
Professor(X) & \leftarrow & FullProfessor(X) \\
Article(X) & \leftarrow & TechnicalReport(X) \\
Publication(X) & \leftarrow & Article(X) \\
Professor(X) & \leftarrow & Chair(X) \\
Publication(X) & \leftarrow & Software(X) \\
Publication(X) & \leftarrow & orgPublication(Y,X) \\
Article(X) & \leftarrow & ConferencePaper(X) \\
Professor(X) & \leftarrow & AssistantProfessor(X) \\
Professor(X) & \leftarrow & AssociateProfessor(X) \\
AUX8(X) & \leftarrow & Program(Y), headOf(X,Y) \\
Director(X) & \leftarrow & AUX8(X), Person(X) \\
Program(f7(X)) & \leftarrow & Director(X) \\
headOf(X, f7(X)) & \leftarrow & Director(X) \\
Person(X) & \leftarrow & Director(X) \\
AUX0(X) & \leftarrow & 1(X) \\
Organization(X) & \leftarrow & AUX0(Y), subOrganizationOf(Y,X) \\
AUX1(X) & \leftarrow & 1(X) \\
1(X) & \leftarrow & AUX1(Y), subOrganizationOf(Y,X) \\
AUX2(X) & \leftarrow & 0(X)
\end{array}
$$

$$0(X) \leftarrow AUX2(Y), subOrganizationOf(Y, X)$$

The explanation of predicates `AUX` is already written on Subsection 5.1.1. Also, the explanation of new function such as `f0` is already written on the same Subsection. The other ontology we used is `stock exchange`, as follows:

$$\begin{aligned}
hasStock(Y, X) &\leftarrow belongsToCompany(X, Y)\\
belongsToCompany(Y, X) &\leftarrow hasStock(X, Y)\\
listsStock(Y, X) &\leftarrow isListedIn(X, Y)\\
isListedIn(Y, X) &\leftarrow listsStock(X, Y)\\
involvesInstrument(Y, X) &\leftarrow isTradedIn(X, Y)\\
isTradedIn(Y, X) &\leftarrow involvesInstrument(X, Y)\\
hasAddress(Y, X) &\leftarrow inverseofhasAddress(X, Y)\\
inverseofhasAddress(Y, X) &\leftarrow hasAddress(X, Y)\\
tradesOnBehalfOf(Y, X) &\leftarrow usesBroker(X, Y)\\
usesBroker(Y, X) &\leftarrow tradesOnBehalfOf(X, Y)\\
Transaction(X) &\leftarrow Acquisition(X)\\
isExecutedBy(X, f0(X)) &\leftarrow Transaction(X)\\
Address(X) &\leftarrow inverseofhasAddress(X, Y)\\
Person(X) &\leftarrow Investor(X)\\
tradesOnBehalfOf(X, f1(X)) &\leftarrow StockBroker(X)\\
Company(X) &\leftarrow hasStock(X, Y)\\
FinantialInstrument(X) &\leftarrow Stock(X)\\
Person(X) &\leftarrow PhysicalPerson(X)\\
Investor(X) &\leftarrow isExecutedFor(Y, X)\\
Transaction(X) &\leftarrow isExecutedBy(X, Y)\\
StockBroker(X) &\leftarrow StockTrader(X)\\
Stock(X) &\leftarrow isListedIn(X, Y)\\
Address(X) &\leftarrow hasAddress(Y, X)\\
Stock(X) &\leftarrow hasStock(Y, X)\\
LegalPerson(X) &\leftarrow Company(X)\\
belongsToCompany(X, f2(X)) &\leftarrow Stock(X)\\
StockBroker(X) &\leftarrow Trader(X)\\
isListedIn(X, f3(X)) &\leftarrow Stock(X)\\
Dealer(X) &\leftarrow Trader(X)
\end{aligned}$$

$$
\begin{aligned}
Trader(X) &\leftarrow Dealer(X)\\
Transaction(X) &\leftarrow involvesInstrument(X,Y)\\
Transaction(X) &\leftarrow isExecutedFor(X,Y)\\
Company(X) &\leftarrow belongsToCompany(Y,X)\\
involvesInstrument(X,f4(X)) &\leftarrow Transaction(X)\\
StockExchangeMember(X) &\leftarrow isExecutedBy(Y,X)\\
Transaction(X) &\leftarrow Offer(X)\\
Transaction(X) &\leftarrow isTradedIn(Y,X)\\
Stock(X) &\leftarrow listsStock(Y,X)\\
FinantialInstrument(X) &\leftarrow involvesInstrument(Y,X)\\
StockExchangeMember(X) &\leftarrow StockBroker(X)\\
hasAddress(X,f5(X)) &\leftarrow Person(X)\\
Person(X) &\leftarrow LegalPerson(X)\\
StockTrader(X) &\leftarrow Trader(X)\\
Trader(X) &\leftarrow StockTrader(X)\\
Person(X) &\leftarrow inverseofhasAddress(Y,X)\\
StockBroker(X) &\leftarrow Dealer(X)\\
inverseofhasAddress(X,f6(X)) &\leftarrow Address(X)\\
Person(X) &\leftarrow StockExchangeMember(X)\\
Stock(X) &\leftarrow belongsToCompany(X,Y)\\
Person(X) &\leftarrow hasAddress(X,Y)\\
isExecutedFor(X,f7(X)) &\leftarrow Transaction(X)\\
StockExchangeList(X) &\leftarrow isListedIn(Y,X)
\end{aligned}
$$

Same as the first ontology we have, the explanation of new functions inside the argument has been explained in Subsection 5.1.1. Last, we have the ontology for `path5` as follows:

$$
\begin{aligned}
Path4(f0(X)) &\leftarrow Path5(X)\\
edge(X,f0(X)) &\leftarrow Path5(X)\\
Path1(f1(X)) &\leftarrow Path2(X)\\
edge(X,f1(X)) &\leftarrow Path2(X)\\
Path2(f2(X)) &\leftarrow Path3(X)\\
edge(X,f2(X)) &\leftarrow Path3(X)\\
Path3(f3(X)) &\leftarrow Path4(X)
\end{aligned}
$$

$$\begin{aligned} edge(X, f3(X)) &\leftarrow Path4(X) \\ edge(X, f4(X)) &\leftarrow Path1(X) \end{aligned}$$

And we have also the explanation for the function inside the predicate covered in Subsection 5.1.1.

Appendix B

Query Used

In this appendix, we give all the query we used to evaluate the performance of the system we created. We try to create as representative query as possible for each ontology we used. The query we use are as follows.

1. For the ontology `university bench`, we use 3 queries, which are:

$$\begin{aligned}
Q(X,Y,Z) &\leftarrow Student(X), advisor(X,Y), FacultyStaff(Y),\\
&\quad takesCourse(X,Z), teacherOf(Y,Z), Course(Z).\\
Q(X) &\leftarrow Person(X), worksFor(X,Y), University(Y),\\
&\quad hasAlumnus(Y,X).\\
Q(X,Y,Z,W,V) &\leftarrow Professor(X), worksFor(X,V), name(X,Y),\\
&\quad emailAddress(X,Z), telephone(X,W).
\end{aligned}$$

2. For the ontology `stack exchange`, we use 3 queries, which are:

$$\begin{aligned}
Q(X,Y,Z) &\leftarrow FinantialInstrument(X), belongsToCompany(X,Y),\\
&\quad Company(Y), hasStock(Y,Z), Stock(Z).\\
Q(X,Y,Z) &\leftarrow Person(X), hasStock(X,Y), Stock(Y),\\
&\quad isListedIn(Y,Z), StockExchangeList(Z).\\
Q(X,Y,Z,W) &\leftarrow FinantialInstrument(X), belongsToCompany(X,Y),\\
&\quad Company(Y), hasStock(Y,Z), Stock(Z),\\
&\quad isListedIn(Y,W), StockExchangeList(W).
\end{aligned}$$

3. For the ontology `path5`, we use 3 queries also, which are:

$$\begin{aligned}
Q(X) &\leftarrow edge(X,Y), edge(Y,Z), edge(Z,W).\\
Q(X) &\leftarrow edge(X,Y), edge(Y,Z), edge(Z,W), edge(W,V).\\
Q(X) &\leftarrow edge(X,Y), edge(Y,Z), edge(Z,W), edge(W,V), edge(V,U).
\end{aligned}$$

Appendix C

Full Results of the Evaluation

In this appendix, we give all the results from the evaluation we make, this result contains the correctness, the time performance, and the memory usage for all 9 queries in 3 ontologies using all the 6 engines we use. The complete results are as follow:

1. Ontology `university bench`

   a) First query, number of data or facts created: 273.
   XSB, with query rewriting: Correct, time: 8.23 seconds, memory: 12 MB.
   XSB, without query rewriting: Error.
   $\mathcal{LDL}^{++}$, with query rewriting: Correct, time: 7.99 seconds, memory: 14 MB.
   $\mathcal{LDL}^{++}$, without query rewriting: Error.
   DLV, with query rewriting: Correct, time: 8.78 seconds, memory: 18 MB.
   DLV, without query rewriting: Error.
   PostgreSQL, with query rewriting: Correct, time: 21.13 seconds, memory: 22 MB.
   PostgreSQL, without query rewriting: Error.
   DB2, with query rewriting: Correct, time: 19.95 seconds, memory: 41 MB.
   DB2, without query rewriting: Error.
   Apache Derby, with query rewriting: Correct, time: 16.01 seconds, memory: 20 MB.
   Apache Derby, without query rewriting: Error.

   b) Second query, number of data or facts created: 341.
   XSB, with query rewriting: Correct, time: 4.13 seconds, memory: 11

MB.
XSB, without query rewriting: Error.
$\mathcal{LDL}^{++}$, with query rewriting: Correct, time: 3.86 seconds, memory: 12 MB.
$\mathcal{LDL}^{++}$, without query rewriting: Error.
DLV, with query rewriting: Correct, time: 5.32 seconds, memory: 15 MB.
DLV, without query rewriting: Error.
PostgreSQL, with query rewriting: Correct, time: 13.75 seconds, memory: 21 MB.
PostgreSQL, without query rewriting: Error.
DB2, with query rewriting: Correct, time: 16.28 seconds, memory: 27 MB.
DB2, without query rewriting: Error.
Apache Derby, with query rewriting: Correct, time: 8.51 seconds, memory: 15 MB.
Apache Derby, without query rewriting: Error.

c) Third query, number of data or facts created: 266.
XSB, with query rewriting: Correct, time: 6.88 seconds, memory: 14 MB.
XSB, without query rewriting: Error.
$\mathcal{LDL}^{++}$, with query rewriting: Correct, time: 6.37 seconds, memory: 17 MB.
$\mathcal{LDL}^{++}$, without query rewriting: Error.
DLV, with query rewriting: Correct, time: 9.12 seconds, memory: 16 MB.
DLV, without query rewriting: Error.
PostgreSQL, with query rewriting: Correct, time: 15.00 seconds, memory: 20 MB.
PostgreSQL, without query rewriting: Error.
DB2, with query rewriting: Correct, time: 14.25 seconds, memory: 38 MB.
DB2, without query rewriting: Error.
Apache Derby, with query rewriting: Correct, time: 12.15 seconds, memory: 13 MB.
Apache Derby, without query rewriting: Error.

2. Ontology `stock exchange`

a) First query, number of data or facts created: 391.
XSB, with query rewriting: Correct, time: 15.21 seconds, memory: 15 MB.

XSB, without query rewriting: Error.
$\mathcal{LDL}^{++}$, with query rewriting: Correct, time: 17.43 seconds, memory: 14 MB.
$\mathcal{LDL}^{++}$, without query rewriting: Error.
DLV, with query rewriting: Correct, time: 14.25 seconds, memory: 22 MB.
DLV, without query rewriting: Error.
PostgreSQL, with query rewriting: Correct, time: 37.14 seconds, memory: 31 MB.
PostgreSQL, without query rewriting: Error.
DB2, with query rewriting: Correct, time: 39.15 seconds, memory: 48 MB.
DB2, without query rewriting: Error.
Apache Derby, with query rewriting: Correct, time: 31.13 seconds, memory: 20 MB.
Apache Derby, without query rewriting: Error.

b) Second query, number of data or facts created: 301.
XSB, with query rewriting: Correct, time: 11.22 seconds, memory: 13 MB.
XSB, without query rewriting: Error.
$\mathcal{LDL}^{++}$, with query rewriting: Correct, time: 12.41 seconds, memory: 14 MB.
$\mathcal{LDL}^{++}$, without query rewriting: Error.
DLV, with query rewriting: Correct, time: 16.78 seconds, memory: 11 MB.
DLV, without query rewriting: Error.
PostgreSQL, with query rewriting: Correct, time: 31.13 seconds, memory: 19 MB.
PostgreSQL, without query rewriting: Error.
DB2, with query rewriting: Correct, time: 28.77 seconds, memory: 38 MB.
DB2, without query rewriting: Error.
Apache Derby, with query rewriting: Correct, time: 24.33 seconds, memory: 16 MB.
Apache Derby, without query rewriting: Error.

c) Third query, number of data or facts created: 277.
XSB, with query rewriting: Correct, time: 19.21 seconds, memory: 17 MB.
XSB, without query rewriting: Error.
$\mathcal{LDL}^{++}$, with query rewriting: Correct, time: 22.31 seconds, memory: 15 MB.

$\mathcal{LDL}^{++}$, without query rewriting: Error.
DLV, with query rewriting: Correct, time: 25.13 seconds, memory: 16 MB.
DLV, without query rewriting: Error.
PostgreSQL, with query rewriting: Correct, time: 48.61 seconds, memory: 24 MB.
PostgreSQL, without query rewriting: Error.
DB2, with query rewriting: Correct, time: 52.13 seconds, memory: 44 MB.
DB2, without query rewriting: Error.
Apache Derby, with query rewriting: Correct, time: 41.17 seconds, memory: 22 MB.
Apache Derby, without query rewriting: Error.

3. Ontology `path5`

   a) First query, number of data or facts created: 388.
   *** XSB, with query rewriting: Correct, time: 4.21 seconds, memory: 10 MB.
   XSB, without query rewriting: Error.
   $\mathcal{LDL}^{++}$, with query rewriting: Correct, time: 5.13 seconds, memory: 13 MB.
   $\mathcal{LDL}^{++}$, without query rewriting: Error.
   DLV, with query rewriting: Correct, time: 6.13 seconds, memory: 17 MB.
   DLV, without query rewriting: Error.
   PostgreSQL, with query rewriting: Correct, time: 14.21 seconds, memory: 21 MB.
   PostgreSQL, without query rewriting: Error.
   DB2, with query rewriting: Correct, time: 16.27 seconds, memory: 35 MB.
   DB2, without query rewriting: Error.
   Apache Derby, with query rewriting: Correct, time: 12.33 seconds, memory: 15 MB.
   Apache Derby, without query rewriting: Error.

   b) Second query, number of data or facts created: 305.
   XSB, with query rewriting: Correct, time: 4.71 seconds, memory: 10 MB.
   XSB, without query rewriting: Error.
   $\mathcal{LDL}^{++}$, with query rewriting: Correct, time: 4.88 seconds, memory: 13 MB.
   $\mathcal{LDL}^{++}$, without query rewriting: Error.

DLV, with query rewriting: Correct, time: 6.21 seconds, memory: 12 MB.
DLV, without query rewriting: Error.
PostgreSQL, with query rewriting: Correct, time: 18.15 seconds, memory: 24 MB.
PostgreSQL, without query rewriting: Error.
DB2, with query rewriting: Correct, time: 17.29 seconds, memory: 48 MB.
DB2, without query rewriting: Error.
Apache Derby, with query rewriting: Correct, time: 13.68 seconds, memory: 17 MB.
Apache Derby, without query rewriting: Error.

c) Third query, number of data or facts created: 331.
XSB, with query rewriting: Correct, time: 11.13 seconds, memory: 13 MB.
XSB, without query rewriting: Error.
$\mathcal{LDL}^{++}$, with query rewriting: Correct, time: 9.51 seconds, memory: 9 MB.
$\mathcal{LDL}^{++}$, without query rewriting: Error.
DLV, with query rewriting: Correct, time: 16.61 seconds, memory: 14 MB.
DLV, without query rewriting: Error.
PostgreSQL, with query rewriting: Correct, time: 38.13 seconds, memory: 37 MB.
PostgreSQL, without query rewriting: Error.
DB2, with query rewriting: Correct, time: 29.41 seconds, memory: 35 MB.
DB2, without query rewriting: Error.
Apache Derby, with query rewriting: Correct, time: 25.13 seconds, memory: 26 MB.
Apache Derby, without query rewriting: Error.

List of Figures

List of Tables