Politecnico di Milano

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

A Query Distribution Algorithm for the Context-ADDICT System

Relatore: Prof.ssa Letizia Tanca

Correlatore: Ing. Giorgio Orsi

Tesi di Laurea di:

Emanuele PANIGATI matr. 720917

Anno Accademico 2009-2010

To My Family
And Friends

Molti aspiranti inventori, quando falliscono nell'intento, si crucciano di essere nati in un epoca in cui tutto é giá stato compiuto e non c'é piú nulla di nuovo che si possa fare.

L'impressione sbagliata per cui, a mano a mano che si avanza, le possibilitá di invenzione si stiano esaurendo, non é affatto rara.

In realtá, é tutto il contrario.

Nikola Tesla,1916

Acknowledgements

Milano, 21 Ottobre 2010,

First of all I would like to thank my advisor, Professor Letizia Tanca (who really always supported me, and she also found me my first job!) and all the Context-ADDICT team's members for the technical support and revisions during this thesis work.

A special thanks goes to two of them: Ing. Giorgio Orsi, who really helped me a lot during this thesis work and revisions, and to Ing. Carlo A. Curino, who helped me a lot with the kick-off of my work.

I thank also Giorgio Inglese, Andrea Gabrielli, Luca Macagnino and Andrea Magni for their previous work on the CA system modules, so that I can continue their work.

A special thanks to my family, who always support me in my life and also in this particular moment that is my thesis work.

I would also like to thank a lot all my friends, especially my closest ones and the ones that work with me in my city theatre (of which I am the "*Stage Director*"), whose insistence obliged me to try and get my master degree as soon as possible.

A thank to all my colleagues engineers: Roberto Paulitti, Matteo, Paoli, Matteo Maggioni, Luca Mantovani, Marco Muffatti, Claudia Olzeri, Armando Natta, Stefania Penoni, Pietro Malossi, Alessandro Marin, Gabriele Maggioni, Andrea Paganini, and lots of other, whose support during my studies and

exams is a key factor in order to obtain the result of which I am proud.

Thanks also to everyone I forgot...

Table of Contents

List of Figures

List of Tables

List of Algorithms

Sommario

L'integrazione dei dati é un settore di ricerca presente sin dalla nascita dei primi sistemi per lo storage dei dati, agli albori della teoria delle basi di dati.

In questa branca di ricerca, tutt'ora aperta ed in evoluzione, una delle ultime innovazioni proposte é l'integrazione dei dati mediata da ontologie di dominio e di contesto, che rappresentano l'ambiente in cui é nata e si sviluppa la nostra ricerca.

Scopo di questo lavoro di tesi é stato quello di completare la piattaforma CONTEXT-ADDICT con un algoritmo in grado di distribuire le varie query prodotte dal modulo di riscrittura (giá implementato in un precedente lavoro) in modo da renderne possibile l'esecuzione sulle varie sorgenti di dati, cosa che al momento non era possibile fare.

Gli algoritmi che verranno presentati si occupano quindi in sostanza della distribuzione delle query sulle varie sorgenti e dell'integrazione dei risultati ricevuti da esse, in modo da fornire, nel piú ristretto tempo possibile, i migliori risultati possibili all'utente che aveva composto l'interrogazione iniziale al sistema, mantenendo durante tutto il processo la correttezza e la completezza dell'insieme di query che andranno distribuite sulle sorgenti.

Abstract

Data integration is a research branch that was born together with the database theory and the first data storage systems.

In this research branch, actually open and always in evolution, one of the most recently proposed data integration methods is the use of *domain and context ontologies* mediated integration methods mediated through domain - and context - ontologies, which represents our research field.

The ultimate purpose of this thesis work is to complete the CONTEXT-ADDICT platform with an algorithm able to distribute all the queries produced by the already existing *rewriting engine* in order to execute them over multiple data-sources. All the algorithms that we are going to expose in this thesis have the purpose of distributing queries to datasources and integrate the resultset that they give back to the system, in order to give, in the shortest time possible, the best achievable results to the final user, who issued the original query to the system. In the whole process the properties of *soundness* and *completeness* of the process must be guaranteed.

Prefazione

Il problema dell'integrazione dei dati é presente sin dagli inizi degli studi sulle basi di dati ed in particolare ai giorni nostri ́diviene piú accentuato a causa dell'eterogeneitá delle possibili sorgenti di informazioni che possono prendere parte al processo di integrazione (basi di dati *relazionali*, file *XML* o addiritura il *World Wide Web* stesso.

Il problema puó sorgere in diversi ambiti, quali l'integrazione di dati provenienti da diverse sorgenti interne ad una stessa compagnia oppure l'integrazione di dati di diverse compagnie (per esempio in caso di fusioni od acquisizioni).

Nel corso degli anni, gli approcci al problema sono stati di vario tipo; il primo esempio di architettura per l'integrazione dei dati é il cosiddetto *Data WareHouse*, che grazie alla procedura nota come *ETL* (**E**xtraction, **T**ransformation and **L**oading), consente di caricare dati provenienti da vari sistemi all'interno di un unico database centralizzato, aggregandoli rispetto ad alcune dimensioni (ad esempio tempo e/o spazio) in modo che risultino utili per eventuali analisi sui dati.

Il passo successivo svolto dalla ricerca ci porta verso i sistemi *con schema mediato*, ovvero sistemi in cui i dati rimangono fisicamente nelle varie sorgenti e le interrogazioni vengono svolte su di una sorta di schema globale e vengono poi tradotte nelle varie richieste da inoltrare alle varie sorgenti.

In questo caso, i possibili approcci sono essenzialmente due, *GAV* (**G**lobal-**A**s-**V**iew) e *LAV* (**L**ocal-**A**s-**V**iew); nel primo caso lo schema mediato é uno schema globale che in pratica una sorta di *view* sulle varie sorgenti (che contengono i dati), mentre nel secondo caso sono le sorgenti che sono delle *view* sullo schema globale (che contiene i dati). Inutile dire che il secondo caso é generalmente piú complesso, ed a volte contorto, da gestire.

l'approccio basato su schema globale si basa sull'omogeneitá delle sorgenti: infatti in questi sistemi di integrazione non ci si cura delle traduzioni tra

modelli e si suppone di partire da sorgenti basate sullo stesso modello dei dati. Estensioni verso sorgenti eterogenee vanno integrate con appositi traduttori, o *wrappers*.

Con l'avvento del *World Wide Web* (in particolare del *Semantic Web*) e di tutti i vari formati collaterali utili alle gestione delle informazioni nell'era di *Internet* (quali ad esempio i file *XML*, i file *RDF*, le ontologie, ...) la necessitá di integrare dati provenienti da diversi tipi di sorgenti si é fatta via via piú forte, rendendo necessario lo studio di tecniche di integrazione dei dati sempre piú avanzate e complesse.

La piattaforma CONTEXT-ADDICT all'interno della quale si va ad inserire il lavoro svolto in questa tesi, appartiene alla categoria di integrazione dei dati guidata dalle ontologie (e, cosa che peró non sará particolarmente tenuta in conto in questa tesi, in quanto nel lavoro svolto questa caratteristica é poco utilizzata, *context-aware*), una delle ultime tecniche su cui la ricerca sta investendo, in quanto consentirebbe una facile implementazione appunto data nel web semantico.

Questo tipo di soluzione é stato raggiunto dopo che, in un primo tempo, si era tentato di arricchire semanticamente gli schemi relazionali, cosa che ha avuto poco successo, dato l'avvento successivo di strutture fortemente semantiche quali le ontologie, che consentono una piú facile implementazione di un *livello semantico* dei dati.

Il lavoro svolto in questa tesi mira a completare il cuore del sistema CONTEXT-ADDICT, composto sostanzialmente dal motore di riscrittura delle query, con un motore di distribuzione delle stesse, in quanto i prodotti della riscrittura sono spesso multi-sorgente, cosa che li rende inutilizzabili per l'esecuzione sulle sorgenti di dati originali.

Il sistema di riscrittura genera un insieme di query *corrette* e *complete*, ognuna delle quali puó multi-sorgente; lo scopo del nostro lavoro sará di estendere questo sistema con la parte di distribuzione delle query, che genera un insieme di query, ognuna delle quali é mirata a una sola sorgente.

Per fare ció sará necessario l'uso di due meta-modelli: il primo é una sorta di semplificazione del concetto di ipergrafo, mentre il secondo é un meta-modello della query, che verrá chiamato *MetaQuery*. Mediante l'utilizzo di questi modelli ed applicando due euristiche (una di *similaritá* ed un'altra di *contenimento*) verrá compiuto il processo di distribuzione e di ottimizzazione,

che porterá tra le altre cose ad un'accentuata riduzione del numero di query da valutare da parte delle singole sorgenti.

Al termine di questo lavoro verrá inoltre eseguita una rapida valutazione *a priori* della complessitá degli algoritmi proposti ed una breve fase di esposizione dei risultati ottenuti da una prima, blanda, attivitá di testing del sistema.

Chapter 1

Introduction

Data Integration is the problem of integrating data which come from multiple, sometimes heterogeneous (like *Relational*, *XML* or *WWW* itself), different sources.

Actually the problem could arise in multiple different scenarios, such as integration of data inside the same company or integrating data among different companies (e.g.: after the merger of different insurance companies they may want to share data among the two information systems).

The first, well known, application of the techniques of *Data Integration*, is the one called "*Data Warehousing*", in which data coming from different information systems are collected and stored into a central database, using a procedure named "*ETL*" (**E**xtraction, **T**ransformation and **L**oading).

The main purpose of the *Data Warehousing* approach is to analyze the data w.r.t different perspectives (e.g. market analysts may be interested in analyzing sales data of a certain product, at certain time, in a certain place); This approach is usually called "*off - line*" since queries are submitted to the *Data Warehouse* system, that usually does not contain real-time data, but usually stores data aggregated under some *dimension* (such as time, place, or any different dimension which allows data aggregation).

An example of the design of the *Data Warehouse* architecture is shown in (Figure 1.1).

Recently researchers provide other approaches in order to obtain *Data Integration*; in particular, in case of homogeneous systems we reach the definition of two different way to provide integration the first way is called *GAV* (**G**lobal-**A**s-**V**iew) while the second one is called *LAV* (**L**ocal-**A**s-**V**iew). The first one

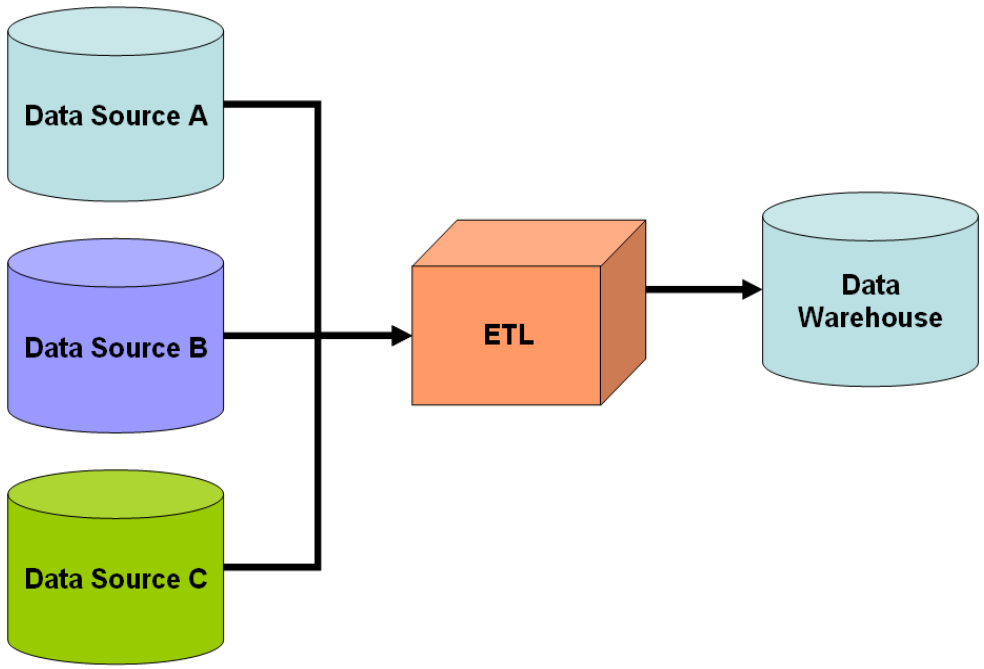

Figure 1.1: Data Warehouse Architecture

defines the global schema (the one which contains integrated data) as a set of view over the multiple original sources schemas and is suitable when the original sources schemas are not supposed to change; the second defines the sources as sets of views over the global schema and is suitable when the global schema is not supposed to change.

The mediated architecture is shown in Figure 1.2.

In the last years, partly due to the *Semantic Web* explosion, we have assisted to the birth of ontologies and to the "*renaissance*" of the so - called "*Knowledge Engineering*".

Using ontology in databases and informative systems made possible to "enrich" data with more semantic information about them compared with the tiny semantic information present in a standard database schema, making possible a more articulate querying of the sources (in the ontology world, under some assumption and with some limitation of the expressive power, it is possible to "reason" with concepts included in them) and also, by extracting semantic models from the original sources, it is possible to integrate multiple heterogeneous systems.

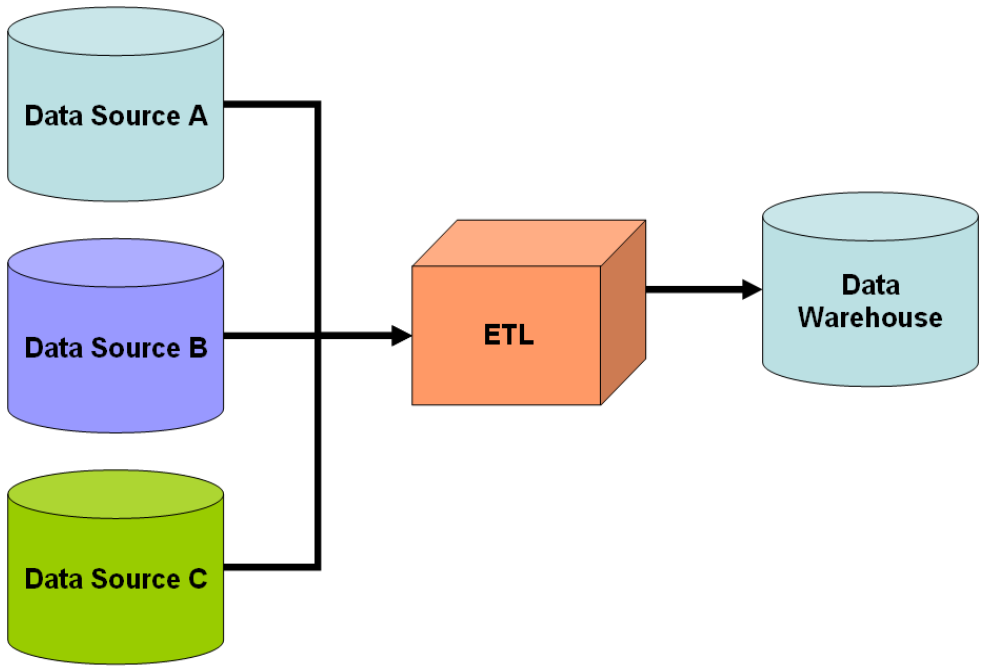

Figure 1.2: Mediated Schema Data Integration Architecture

This thesis work is part of a larger project, named "*Context - ADDICT*[1] [11]. The final target of this project is to produced an ontology - based tool which allows to integrate multiple distributed (and mobile), heterogenous data sources, which produce huge amounts of data which needs to be filtered and tailored on the needs of the final user interest and context.

The work described in this thesis concentrate on the issue of data integration. It aims at describing (and producing) an algorithm to rewrite and distribute a *SPARQL* query issued over a global domain ontology - describing the domain in which we are interested - to each original data source by means of the *data source semantic ontologies* (which describe the semantic of the sources). The results coming from the data sources are finally integrated in order to answer to the original query issued by the user.

This technique of data integration since them make it possible to interoperate very quickly and also can be easily used from interfaces of the kind used by the *Semantic Web*, which in future can lead to a more user-friendly use of the data integration systems (which are at the moment very specific and not

[1]Context Aware Data Design, Integration, Customization and Tailoring.

easily accessible by average users).

1.1 Outline of the Thesis

In Chapter 2 we recall some preliminary concepts about the *Data Integration* problem, ontologies and description logics, in order for the reader to better comprehend where is the starting point of this work.
In Chapter 3 we are going to see which is the architecture of the *Context - ADDICT* system and which are its modules. In Chapter 4 we are going to see the current solutions for query rewriting and dispatching over multiple heterogeneous sources. In Chapter 5 we will describe our semantic extraction procedure from generic relational schemata. Then, in Chapter 6 we will describe the algorithm used in order to process and distribute the query while in Chapter 7 we are going to see the detailed implementation of it in the related modules of the *Context - ADDICT* system. In Chapter 8 we are going to evaluate the resulting system and in the final Chapter 9 we report the final conclusion about the whole work.

Finally, in Appendix A we report some examples of the ontologies mentioned in our work.

Chapter 2

Preliminary Concepts

2.1 Introduction

In this chapter we will provide some preliminary concepts needed to understand better what is the scope of this thesis work. We will start with a quick introduction on Ontologies and Description Logics, giving some formal definiton useful to understand how the query processing and distribution algorithm works.

We will also give basic notions about about $CA - ML$, the ontology mapping language that is used in CONTEXT-ADDICT and `SPARQL`, the query language that is used in order to query our system.

Most of the concept that we discuss in this chapter has been borrowed from [39] and [27], those discuss previously implemented modules of the CONTEXT-ADDICT system.

2.2 Ontologies

Ontology is a branch of philosophy, it is the science of what is, of the kinds and structures of the objects, properties and relations in every area of reality. Ontology in this sense is often used in such a way as to be synonymous with metaphysics. In simple terms it seeks the classification of entities. Each scientific field will of course have its own preferred ontology, defined by the field vocabulary and by the canonical formulations of its theories.
Traditional (philosophical) ontologists have tended to model their own research

fields on these scientific ontologies, either by producing theories which are like scientific theories but radically more general than these, or by producing theories which represent a strong organization of scientific theories or a clarification of their foundations.
Philosophical ontologists have more recently begun to concern themselves not only with the world as this is studied by the sciences, but also with domains of practical activity such as law, medicine, engineering, commerce. They seek to apply the tools of philosophical ontology in order to solve problems which arise in these domains.

2.2.1 Ontologies in Information Systems

The concept of ontology was first borrowed from the realm of Philosophy by Artificial Intelligence researchers and has since become a matter of interest to computer and information scientists in general. In Computer Science literature, the term takes a new meaning, not entirely unrelated to its philosophical counterpart.
A definition of ontology is attributed to Tom Gruber[42] who defines it as *a shared and formal specification of a conceptualisation.* This definition, in its terms, borrows from the Artificial Intelligence literature on Declarative Knowledge, which is concerned with the formal symbolic representation of knowledge[34]. In this field, formal logical languages, such as first-order predicate calculus, are used to describe models of the real world. This is necessary because natural languages are too ambiguous for machine interpretation and formal reasoning.
In an ontology, and as a consequence in a knowledge base reasoning is usually based on the Open World Assumption (OWA), that assumes that knowledge of the world is incomplete: if something cannot be proved to be true, then it does not automatically become false. This is a key difference with classical database theory where a Closed World is assumed: if something is already present as an axiom then it is true and deducible, it is considered false otherwise.

Elements of an ontology

An ontology O can be formally defined as a 5-tuple; this 5-tuple contains (according to [36]):

- *a set C whose elements are called concepts.*
- *a set of relations* $R \subseteq C \times C$.
- *a set of axioms on O, called* A^O.

To cope with the lexical level, the notion of a lexicon is introduced.
For an ontology structure $O = \{C, R, A^O\}$ *a lexicon L is defined as*

$$L = \{L^C, L^R, F, G\}$$

where:

- L^C *is a set whose elements are called lexical entries for concepts.*
- L^R *is a set whose elements are called lexical entries for relations.*
- $F \subseteq L^C \times C$ *is a reference for concepts such that:*

$$F(l_c) = \{c \in C \mid (l_c, c) \in F\} \; \forall l_c \in L^C$$
$$F^{-1}(c) = \{l_c \in L^C \mid (l_c, c) \in F\} \; \forall c \in C$$

- $G \subseteq L^R \times R$ *is a reference for relations such that:*

$$G(l_r) = \{r \in R \mid (l_r, r) \in G\} \; \forall l_r \in L^R$$
$$G^{-1}(r) = \{l_r \in L^R \mid (l_r, r) \in G\} \; \forall r \in R$$

Individuals: Individuals (or instances) are the "Ground level" components of an ontology. The individuals in an ontology may include concrete objects such as people, animals, vehicles as well as abstract individuals such as numbers and words. An ontology need not include any individuals, but one of the general purposes of an ontology is to provide a means of classifying individuals, even if those individuals are not explicitly part of the ontology.

Classes: Classes (or concepts) are abstract groups, sets, or collections of objects. They may contain individuals, other classes, or a combination of both. Ontologies vary on whether classes can contain other classes, whether a class can belong to itself, whether there is a universal class (that is, a class containing everything), etc.
Sometimes restrictions along these lines are made in order to avoid the loss of

decidability of the inference system.

Roles: Roles are relations binding two concepts (object properties) or a concept to a datatype (datatype properties). Roles describe interactions between concepts or their properties.

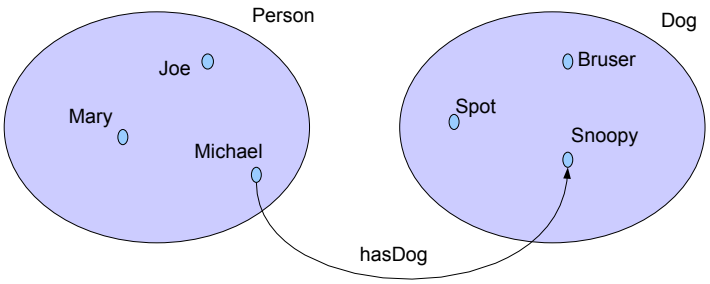

Figure 2.1: A conceptualization

Axioms: Axioms are used to impose constraints on the values of classes or instances. Axioms represents the starting point for the inference systems, they are the hypothesis of a formal logic theory and are always true if used inside the ontology.

2.2.2 The OWL language

OWL (Web Ontology Language[5]) is a language for defining and instantiating Web ontologies based on XML, (Extensible Markup Language)[4] and RDF (Resource Description Framework)[2]. OWL is designed for use by applications that need to process the meaning of an information instead of just presenting that information to the user.
OWL can be used to explicity represent the meaning of terms in vocabularies and the relationships between those terms; by this language it is possible to infer new knowledge from a conceptualization using a specific software called *reasoner*.

OWL provides three increasingly expressive sublanguages also called *OWL dialects*:

- **OWL Lite:** Provides classification hierarchies and simple constraints, it only permits to express relationships with maximum cardinality equal to O or 1.
- **OWL DL:** Supports those users who want a high expressiveness while retaining computational completeness and decidability. OWL DL includes all OWL language constructs, but they can only be used under certain restrictions (i.e. a class cannot be an instance of another class). OWL DL is so named due to its correspondence with Description Logics (see below).
- **OWL Full:** Provides the maximum expressiveness and the syntactic freedom of RDF with no guarantees on computational complexity. A key difference of this dialect from the former is that a deductive process within such a theory can be undecidable.

There are many other formalism to describe an ontology, one of the most used is the N3 form[1]. It has the same expressivity as OWL and it is particularly useful to machines because it is more compact than OWL with fever constraints on the representation. The N3 form consists of a set of triples (Subject - Predicate - Object) that describes the so called *statements*. The set of the statements represents the final ontology.
In Figure 2.2 it is shown how the same class (in this case a military vehicle)can be represented in different formalisms: OWL, N3 and a graph based representation. The last is equivalent to OWL and only used by designers to have a graphical representation of the ontology.

2.3 Description Logics - DL

Description Logics (DL)[9] are a family of decidable logics for formal knowledge representation. They can represent concepts and their relationships (roles) giving them also formal semantics. In particular, we will refer to a description logic named *SHOIN($\boldsymbol{D}_n$)* that is the underlying logic of the OWL language described in Section 2.2.2.

```
<rdf:Description rdf:about="&weapons;MineWarfareVessel">
  <rdfs:subClassOf rdf:resource="&weapons;SurfaceShip"/>
  <rdfs:subClassOf rdf:resource="&weapons;ModernNavalShip"/>
  <rdf:type rdf:resource="&owl;Class"/>
</rdf:Description>
```

```
a:MineWarfareVessel
    a owl:Class;
    rdfs:subClassOf a:ModernNavalShip , a:SurfaceShip .
```

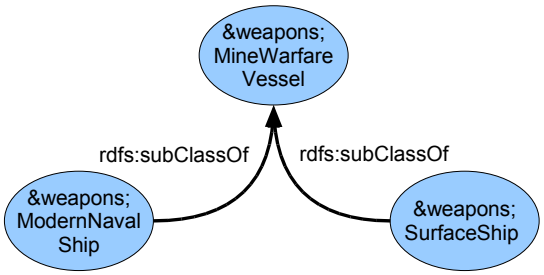

Figure 2.2: Three equivalent representation of the same class

The main descriptive tools of a DL are represented by its *concepts constructors*: by combining them in a suitable form it is possible to describe the concepts and their relations. One particular DL is defined on the basis of its constructors.

2.3.1 Concepts and Roles

In DL, like in ontologies in general, we can distinguish Concepts and Roles; a concept is a set of individuals of the application domain that have some common characteristics as it can be for *people* or *cars*. Roles are the logical representations of relationships between concepts, for example, the role *hasFather* or the role *madeOf*.

Concepts and roles are also divided into *atomic* and *complex*, where by atomic we mean that it is not decomposable in terms of other concepts or roles inside the same logic theory. A complex concept is made of other atomic or complex concepts and roles combined by means of the constructors of the given DL.

The process of assigning a meaning to an atomic concept is called *Symbol Grounding* and is related to the problem of interfacing the automatic system to the real world (e.g. using sensors and trasducers). In DL there are two special concepts named *Top* and *Bottom*, Top ($\top$) is a concept that contains

all the individuals of the domain, while Bottom ($\bot$) is the empty concept, which also represents the contraddiction.

2.3.2 Knowledge Bases and KRS

A knowledge Base (KB) is a particular and evolved form of information system, it can contain many different conceptualizations of different domains named ontologies and it is managed through a Knowledge Representation System (KRS) that gives the facilities for managing and querying the KB, defining new concepts and roles and inferring new knowledge. If a KRS can only query, the KB is called a Knowledge Inference System (KIS) or simply a *Reasoner*.
A KB is usually constituted by two elements: KB = $\langle$T*box*, A*box*$\rangle$

- **TBox:** The T*box* (Terminological box) contains all the concept and role definitions, and also contains all the axioms of our logical theory (e.g. *"A father is a Man with a Child"*). The axioms of a T*box* can be divided into *definitions* (C $\equiv$ D) and *subsumptions* (C $\sqsubseteq$ D), the former used to say that a concept C is equivalent to another concept D (atomic or complex), the latter used to say that a concept C is a subclass of the concept D. The reader can find more precise definitions of these operators in Section 2.4.

- **ABox:** The A*box* contains all the assertions (also known as facts) of the logic theory, an assertion is used to express a property of an individual of the domain (for example *"Tom is a father"* is represented as *Father(Tom)*). An assertion is also *R(a,b)* where *R* is a role (e.g. *hasFather (James, Tom)*).

2.3.3 Interpretations

An interpretation is the way to give a formal semantics to a KB. An interpretation of a KB is a triple:

$$\mathrm{I}=\langle \Delta^I, \xi^I, \psi^I \rangle$$

where:
Δ^I is called *interpretation domain* and contains all the individuals of the domain which we want to predicate on.

$\xi^I : \Theta \longrightarrow \wp(\Delta^I)$ with Θ the set of the concepts, is the evaluation function for the concepts and associates a concept C to a subset of Δ^I.
$\psi^I : \Pi \longrightarrow \wp(\Delta^I \times \Delta^I)$ with Π the set of roles, is the valuation function for the roles and associates a role R to a subset of $\Delta^I \times \Delta^I$.

2.3.4 Models

Given a T*box* T = $\{C_i \sqsubseteq C_j, i, j \in [\ 0..\text{n}\ ]\}$ with C_i and C_j generic concepts, and an interpretation I,
I is a *model* for T if and only if for all the subsumptions in T we have that $\text{C}_i^I \subseteq \text{C}_j^I$) where:

$$\forall i \in [0..n]\ \text{C}_i^I = \begin{cases} \xi^I(C_i) & if\ C_i\ is\ a\ concept. \\ \psi^I(C_i) & if\ C_i\ is\ a\ role. \end{cases}$$

Note that the T*box* is expressed only in terms of subsumptions; it obviously comprehends also the equivalence definitions ($C_i \equiv C_j$) that are decomposed in two subsumbtions ($C_i \sqsubseteq C_j$ and $C_j \sqsubseteq C_i$)

2.4 Reasoning services

Reasoning services[20] are the tasks of the KRS. We can distinguish them into services for the T*box* and services for the A*box*[26]. The services that involve only the T*box* are:

- **Subsumption:** This task verifies if a concept C is subsumed by another concept D (T*box* $\models$ C$\sqsubseteq$D); this is true if and only if for all the interpretations I we have that $\text{C}_i^I \subseteq D_i^I$.
- **Consistency:** This task verifies that there exists at least one interpretation I for a given T*box* (T*box* $\not\models \bot$).
- **Local Satisfiability:** This task verifies, for a given concept C that there exist at least one interpretation in which $\text{C}^I \neq \varnothing$.

The services for the A*box* are:

- **Consistency:** This task verifies that an A*box* is consistent with respect to a given T*box* (KB $\nvDash \bot$).
- **Instance Checking:** This tasks verifies if a given individual x belongs to a particular concept C (A*box* $\models$ C(x)).
- **Instance Retrieval:** This tasks returns the extension of a given concept C, that is, the set of individuals belonging to C.

2.5 Description Logics Families

In this section we will describe what are the constructors of the DLs and, as a consequence, what are the most common DL families. For every DL the concepts Top and Bottom are interpreted as:

$$\top \equiv \mathrm{A} \sqcup \neg \mathrm{A} \Longrightarrow \top^I = \Delta^I$$
$$\bot \equiv \mathrm{A} \sqcap \neg \mathrm{A} \Longrightarrow \bot^I = \varnothing$$

2.5.1 Constructors

Negation:
The negation constructor is indicated as ¬C and the presence of such a constructor in a specific DL is indicated with the character "C" in the logic's name (e.g. the logic AL with the addition of negation is named ALC); see Figure 2.3.¬C is interpreted as $(\neg \mathrm{C})^I = \Delta^I \backslash \mathrm{C}^I$. It is used to tell that a concept is disjoint from another concept (e.g. Man $\sqsubseteq$ ¬Woman). There are some logics that allow negation only for atomic concepts.

Concept Intersection:
The concept intersection constructor, denoted by C⊓D and interpreted as $(\mathrm{C} \sqcap \mathrm{D})^I = \mathrm{C}^I \cap \mathrm{D}^I$, is used to tell that a concept is defined by the intersection of the individuals of two concepts (i.e. Driver $\sqsubseteq$ Person ⊓ ¬Blind).

Concept Union:
The "union of concepts" constructor is denoted by C⊔D (also called U in the logic's name) and is interpreted as $(\mathrm{C} \sqcup \mathrm{D})^I = \mathrm{C}^I \cup \mathrm{D}^I$; it can be used to tell that a concept is defined by the union of the individuals of two concepts (i.e.

People $\sqsubseteq$ Male $\sqcup$ Female).

Universal Qualified Quantification:
The universal qualified quantification is denoted by $\forall$R.C and it is interpreted as $(\forall R.C)^I=\{a\in \Delta^I \mid \forall b, (a,b)\in R^I \Rightarrow b\in C^I\}$. It means that if an individual a is in relation with another individual b through R then b belongs to the concept C. This constructor can be also unqualified when $\top$ instead of C is used. This constructor is useful when we need to define the range of a given role R.

Existentially Qualified Quantification:
The existential qualified quantification is indicated as $\exists$R.C (also indicated as ϵ) and it is interpreted as $(\exists R.C)^I=\{a\in \Delta^I \mid \exists b, (a,b)\in R^I\}$. Also this constructor can be used in an unqualified form.

Transitive and Inverse Roles:
When a role has the transitive property it is indicated as R^+ and can be used in constructors like the existential and the universal quantifications. A logic that provides this constructor has the letter "S" in its name. This constructor is useful for instance, when we need to build a meronymy relation like "*partOf*". For example, if we have two axioms like:

$$\text{Engine} \sqsubseteq \exists \text{partOf.Clutch}$$
$$\text{Clutch} \equiv \exists \text{partOf.ClutchDisk}$$

we can infer

$$\text{Engine} \sqsubseteq \exists \text{partOf.ClutchDisk}$$

When a role represents the inverse relation of another concept S it is indicated as S^- and is interpretes as the inverse relation of the interpretation of S. This constructor is also indicated as "I". This constructor is used to define the domain of a given role R, for example, to express that a role R has a domain C and a range D we can write:

$$\top \sqsubseteq \forall R.D$$
$$\top \sqsubseteq \forall R^-.C$$

Role Subsumptions:
There exists a constructor that gives the opportunity to define an hierarchy of roles, this constructoris defined by R$\sqsubseteq$S $\Leftrightarrow$ [(a,b)$\in$R $\Rightarrow$ (a,b)$\in$S]. A description logic that provides this constructor has the letter "H" in its name.

Enumeration:
The constructor oneOf (also indicated by the letter "O") is used when we want to define a concept by enumeration of its individuals (e.g. RGBColors = {Red, Green, Blue }).

Existential Qualified Quantification:
We can make a role R a functional role using the constructor $\leq$1R (indicated by the letter "F") that is interpreted as $(\leq 1R)^I = \{s \in \Delta^I \mid count(t):(s,t) \in R^I \leq 1\}$. If we want to arbitrarly restrict the number of individuals that are involved in a relation, we can use the constructors N interpreted as $(\geqslant nR)^I = \{s \in \Delta^I \mid \sharp\, t:(s,t) \in R^I \geqslant n\}$ and Q interpreted as $(\geqslant nR.C)^I = \{s \in \Delta^I \mid count(t):(s,t) \in R^I \geqslant n \wedge y \in C^I\}$. The key difference between this constructor and the existential quantification is the possibility to restrict the number of individuals involved in a given role R.

2.5.2 Families

The logic we are interested in is named *SHOIN(D_n)*, and has the following constructors:

- Negation
- Intersection
- Union
- Universal Qualified Quantification
- Existential Qualified Quantification
- Transitive Roles (S)
- Roles Subsumptions (H)

- OneOf (O)
- Inverse Roles (I)
- Existential Quantificated but not Qualified Quantification (N)

and it carries a datatype system ($\mathbf{D}_n$).
There are more DL families; one of the most important is the logic *SHIQ* that is used instead of *SHOIN($\mathbf{D}_n$)* in many KRS that use ontology language like DAML-OIL instead of the OWL language.
The most common DLs and their constructors are shown in Figure 2.3.

2.6 Data integration systems related concepts

In this section we give a formalization of a data integration system (DIS) according to [31] and [40]. In a hierarchical DIS, elements of the global schema and of data sources are linked by means of mappings. It follows that a hierarchical DIS is mainly composed by the global schema, the data sources and the mapping between them. Data sources store the information of the system and their structure is described by a source schema. The global schema is a virtual integration of these source schemas, in order to provide an unified view to the user. It works like a mediator between the user and the sources, allowing users to query the mediator and receive results from data sources. Formally we can define:

Definition: A **data integration system I** is a triple (G,S,M), where:

- G is the global schema, expressed in a language L_G over an alphabet A_G. The alphabet comprises a symbol for each element of G (i.e. a relation if G is relational, a concept or role if G is a Description Logic, etc.).
- S is the source schema, expressed in a language L_S over an alphabet A_S. The alphabet A_S includes a symbol for each element of the sources.
- M is the mapping between G and S constituted by a set of assertions in the forms:

$$q_S \rightsquigarrow q_G$$
$$q_G \rightsquigarrow q_S$$

  where q_S and q_G are two queries of the same arity, respectively over the source schema S and over the global schema G. Queries q_S are expressed in a query language L_M, S over the alphabet A_S, and queries q_G are expressed in a query language L_G over the alphabet A_G,. Intuitively, an assertion $q_S \rightsquigarrow q_G$ specifies that instances resulting from the query q_S over the sources correspond to instances in the global schema represented by the query q_G (similarly for an assertion of type $q_G \rightsquigarrow q_S$).

Given a fixed infinite domain Δ of elements, a domain of the databases of the system, we can talk about the semantics of a DIS I. We start defining as

model of data present in the sources, the source database D over Δ, fixing the extension of the predicates of A_S. The extension of a predicate is the set of all x that verify the predicate $P(x)$ in the database. We also call global databases for a DIS I an instance of the global schema G over a domain Δ.

Definition: The **set of databases** for A_G (A_G is the alphabet of the global schema) that satisfy I relative to D, denoted $sem(I, D)$ is the set of databases B such that:

- B is a global database, and
- B satisfies the mapping M w.r.t. D (where D is the set of data sources).

The sentence B satisfies M w.r.t. D depends on the semantics of the mapping assertions.

2.6.1 An introduction to the problem of consistency in DIS

In other words, we call B a global database for a DIS I, if it satisfies all the constraints over G and it also satisfies the mapping between G and D. However, it may happen that a data integration system I be inconsistent w.r.t D , i.e. , $sem(I, D) = \emptyset$. To get an idea of what we mean for inconsistency, we can consider the global schema G as having only the concept Person and a role emphhasSSN having as domain Person and as range `xsd:string`[1]. If we express some form of key constraint over the role hasSSN such as: $\forall x Person(x)?\exists hasSSN.SSN$ saying also that hasSSN and its inverse are functional (i.e. , Persons have at most one SSN and every SSN has at most one Person). When data sources are mapped to the concept of Person we could have inconsistencies deriving from the fact that two (or more) different individuals (belonging to Person) have the same SSN (or that the same individual has two or more SSNs). This point requires particular attention because, as also stated by [40], the issue of consistency in data integration systems despite its importance, is a topic rarely discussed. Indeed, from a theoretical point of view, query answering over an inconsistent system (over a contradictory theory) is meaningless (ex falso quod libet principle). However, before concluding

[1] xsd:string in an XML Schema datatype.

this topic we note that in this thesis we are not concerned with the problem of consistency of the global query answering architecture and that consistency will be the subject of future work. Anyway, due to our dynamic and semi-automatic environment for data integration we expect inconsistencies to be rather frequent and, probably, we will have to address the problem of verifying the global systems consistency using a framework different from that of [40].

2.6.2 Mappings

In this section we describe the different approaches used to specify the relationship between the global schema and the source schemas. The mapping task is a crucial point of a DIS because it permits the translation of the query over the global schema into queries over the data sources. Moreover it is also important in the opposite way, when the answers coming from the different sources must be rewritten in terms of global schema elements. Before continuing, we give a particular classification of views common between GAV and LAV, that permits to characterize each source with respect to the global schema. According to [31] the specification *as(s)* determines how accurate is the source s with respect to the associated global view q_G. We have:

sound views: When a source s is *sound* ($as(s) = sound$), its extension provides any subset of the tuples satisfying the correspondent view q_G, in other words the tuples in source database are also present in global view, but the elements in global view could not be present in the data source. Formally speaking a database B satisfies the assertion $s \rightsquigarrow q_G$ with respect to D if

$$s^D \subseteq q_G^B$$

complete views: When a source s is *complete* ($as(s) = complete$) its extension provides any superset of the tuples satisfying the corresponding view. If a tuple is in s we can not say that it is also in global view, but if the tuple is not in s we can conclude that the tuple doest not satisfy the view. Formally, a database B satisfies the assertion $s \rightsquigarrow q_G$ with respect to D if

$$s^D \supseteq q_G^B$$

exact views: When a source s is *exact* ($as(s) = exact$) its extension is exactly the set of tuples satisfying the corresponding view. Formally, a database B satisfies the assertion $s \rightsquigarrow q_G$ with respect to D if

$$s^D = q_G^B$$

Mapping forms According to data integration literature,in a *DIS* we have the following types of mappings:

- GAV: (Global-as-view) where the mapping M associates to each element g in G a query q_S over S.
- LAV: (Local-as-view) where the mapping M associates to each element s of the source schema a query q_G over the global schema.
- GLAV: where there are no restrictions on the forms of mappings inside a *DIS* (a query over a data source can be related to a query over the global schema).

Rewritings Starting from the previously introduced mapping forms, we define three different strategies in order to rewrite queries:

GAV Global-As-View in a GAV mapping we have a global schema specified as a view over the sources, so usually[2] the process of query answering is resolved by *unfolding* the original query Q over the global schema G. This approach is good in stable environments, because as the global schema is defined in terms of the data sources when we add a new data source we have to re-define the global schema G taking into account the new data source.

LAV Local-As-View in a LAV mapping we have each data source DS_i specified as a view over the global schema G. The process of query answering requires the so called *view-based* query answering, because we have a query Q over the global schema G and we have to answer the query basically exploiting view definitions. This process can be a difficult task, but LAV approaches can be useful when data sources have to be frequently added or removed from the system, because in this case adding

[2]Considering the absence of integrity constraints over the global schema.

a new source simply requires to define the new data source in terms of the global schema G.

GLAV A mixed approach of the two above.

Maximally-contained rewriting In order to handle LAV mappings a particularly useful concept is the one of *maximally-contained rewriting*, which is defined as follows:

Let Q be a query, $V = V_1, ..., V_m$ be a set of view definitions, and L be a query language. The query Q' is a *maximally-contained rewriting* of Qusing V w.r.t. L if:

- Q' is a query in L that refers only to the views in V or comparison predicates
- Q' is contained in Q[3] and
- there is no rewriting $Q_1 \in L$ such that $Q \subseteq Q_1 \subseteq Q$ and Q_1 is not equivalent to Q'. When a rewriting is contained in Q but is not a maximally-contained rewriting we call it a *contained rewriting*.

In other words, the above definition states that given a set of data sources which represent a subset of the virtual tuples of the Global schema, we search for the rewriting Q' which returns the maximal set of tuples and we state that every query that returns a subset of those tuples is a contained rewriting. As noticed in [24], this problem depends also on the query language which is used to express the query.

Certain answer Another useful definition that we borrow from [24] is the definition of *Certain Answer* which is useful when we have some partial view[4], so when we retain valid the *Open World Assumption*.

[3]We say that a query Q' is contained in Q if it returns a subset of the tuples that would be returned by Q

[4]Informally, with the Closed-Word Assumption, we assume that the views are as- sumed to contain all the tuples that would result from applying the view definition to the database. Conversely, with the Open-World Assumption the extension of the views may be missing some tuples

Definition: Let Q be a query and $V = V_1, ..., V_m$ be a set of view definitions over a Global schema and let the sets of tuples $v_1, ..., v_m$ be extensions of the views $V_1, ..., V_m$ respectively. The tuple a is a *certain answer* to the query Q under the Open-World Assumption given $v_1, ..., v_m$ if $a \in Q(D)$ for all databases D such that $v_i \subseteq V_i(D)$ for every $i, 1 <= i <= m$.

Intuitively, the extensions of $V_1, ..., V_m$ do not define a unique extension of the database relations. Hence, given the extension of the views, we have only partial information about the real state of the database. A tuple is a certain answer of the query Q if it is an answer for any of the possible database extensions that are consistent with the given extension of the views [24].

2.7 Conjunctive queries

In data integration literature the interest is usually on conjunctive queries, that is, a query language that has at least the expressive power of `select project join` queries in SQL. Although formal definitions of conjunctive queries are given for example in [6], here we give a definition that is much more DL oriented and that suits well to our case:

Definition: A conjunctive query has the form:

$$Q(\bar{X}) := A_1(\bar{X}_1) \wedge ... \wedge A_n(\bar{X}_n)$$

where $\bar{x}$ are variables or individuals and each A_i is a binary or unary relation (respectively, a role membership assertion or a class membership assertion). More generally, a conjunctive query q over a knowledge base K is an expression of the form $q(x) \leftarrow \exists y.conj(x, y)$ where x are the so-called distinguished variables, y are existentially quantified variables called the non-distinguished variables and $conj(x, y)$ is a conjunction of atoms of the form $C(a), R(a, b)$ where C and R are respectively a concept and a role in K and a, b are constants of a fixed infinite domain or variables in x or y.

2.8 Context ADDICT Mapping Language

CA - ML is an OWL-DL language subset which allows us to express every kind of mapping between our *Domain Ontology* and our sources ontologies.

In order to define the language we need the following notation:

- A, which denotes a named concept (i.e. a concept that has been explicitly named and that is part of the asserted hierarchy);
- M, which denotes a named concept belonging to a mapping ontology;
- C, which denotes an anonymous concept (i.e. a concept that has not been explicitly named but exists and contains all the individuals that satisfy certain conditions);
- R, which denotes a named role;
- the subscript $_{DSO}$ which denotes *Data Source Ontology* and the subscript $_{DO}$ which denotes *Domain Ontology*.

Given this notation, it is possible to define an anonymous concept C, used to define mapping concepts as:

$C ::= A|C_1 \sqcap C_2|\exists R.B$

$B ::= A|T|B_1 \sqcap B_2$

With this concept, it is possible to define the following statements:

(*Definition of Mapping Concepts*)

$M_O \equiv C_O$

(*LAV and GAV sound Mappings*)

$A_{DSO} \subseteq M_{DO}$ (LAV)

$M_{DSO} \subseteq A_{DO}$ (GAV)

(*Full Implication Mappings*)

$R_{DSO} \subseteq P_{DO}$

$A_{DSO} \subseteq A_{DO}$

More details about $CA - ML$ can be found in [27].

2.9 SPARQL

SPARQL is the acronym for **S**PARQL **P** **a**nd **R**DF **Q**uery **L**anguage, which is the language, developed by the *Data Access Working Group* of the *w3c consortium*.

Actually it is a standard for RDF and ontology quering as recommended by *W3C* in an official recommendation of the *15th of January 2008* [3].

This query language has become the central point of the *Semantic Web*, allowing everyone to query and extract knowledge from RDF datafiles, using a syntax similar to that of other query languages such as, for instance, *SQL*.

The basic element of a SPARQL query is the so called "*Triple*", which is usually composed by:

- a ***subject***, which is the concept (or variable) the property stated in the *predicate* applies to
- a ***predicate***, the property which correlates *subject* and *object*
- an ***object***, which is a concept (or a variable, or a literal) the property in the *predicate* is applied to

We can have different triple patterns, for instance we can have type declarations (in which the predicate is the `rdf:type` property) or properties between two variables (usually those properties are `ObjectProperties`) or properties between a variable and a literal (usually those properties are `DataTypeProperties`; The query is composed by a mix of the previous elements, put together into the *WHERE* clause of the SPARQL query.

Moreover, a query can contain two more clauses, which do not properly respect the pattern of a triple.

These two clauses are:

- the `FILTER` clause, which allows the user to make tests against one or more of the variables involved in the query (e.g. testing if a variable is equal to a determinate string)
- the `ORDER BY` clause, which makes it possible to order the result of our query, by selecting the variables on which the ordering has to be computed.

It is also possible to use the `LIMIT` and `OFFSET` clauses in order to limit the number of results retrieved by the query and to move into subsets of the result space.

Also we can have different types of queries; beside the standard SELECT queries (that simply return each variable in the clause with direct bind to its values) we can have also CONSTRUCT, ASK and DESCRIBE queries.

First of all, the CONSTRUCT statement allows the user to retrieve the result into an RDF graph form (the graph structure has to be specified after the CONSTRUCT directive), instead of a list of variable - value bindings returned by the SELECT clause.

The ASK statement instead, only returns true if the query has solution, otherwise it will return false.

The DESCRIBE, returns an RDF graph containing RDF data about the results of the query.

Despite this great variety of possible queries, in this thesis work we are going to consider exclusively SELECT-statement-based queries, since they will be the most used ones for our purposes of data retrieval and integration.

Logic	Operators
AL	Negation of atomic concepts
	Intersection
	Universal Qualified Quantification
	Existential Unqualified Quantification
AL [U] [ε] [C]	All the constructors of AL
	Union
	Existential Qualified Quantification
	Negation of complex concepts
S	All the constructors of ALC
	Transitive roles
SI	All the constructors of S
	Inverse roles
SHI	All the constructors of SI
	Subsumption between roles
SHIF	All the constructors of SHI
	Functional roles
SHIN	All the constructors of SHI
	Existential Unqualified Quantificated Quantification
SHIQ	All the constructors of SHI
	Existential Qualified Quantificated Quantification
SHOIN	All the constructors of SHIN
	OneOf
SHOIN($\mathbf{D_n}$)	All the constructors of SHOIN
	A set of datatypes

Figure 2.3: Constructors of the most common DL Families

Chapter 3

Context - Addict Architecture

3.1 Introduction

In this chapter we briefly describe the Context-ADDICT (Context-Aware Data Design, Integration,Customization and Tailoring) project [13]. To this aim, we start by explaining its main objectives and then we give a general overview of the system discussing its main components.

3.2 Project objectives

The main goal of the Context-ADDICT system[1] is to create a *middleware* infrastructure to support the development and the design of context-aware and data-intensive applications. The focus is on mobile, *peer-to-peer* applications, where the notion of context can be exploited to provide the user with a filtered view over the data, retrieving only the information relevant to the user in his/her current context. The main issues addressed by the system are:

- Definition of a user and context model.
- Semantic extraction from data sources (the process of creating a uniform semantic representation of the data stored at a given data source).
- Semantic integration of different heterogeneous data sources.

[1]From this point on, we will eventually refer to the Context-ADDICT system with the abbreviation CA.

- *Data Tailoring* (the process of filtering data using a model of context).
- Support for query distribution (the process of querying multiple heterogeneous data sources).
- Synchronization of data.

3.3 Thesis objectives

In this thesis work we are going to focus on a particular aspect of the query processing in the CA system, which is the one that has to deal with query distribution over multiple sources. Indeed the currently available rewriting module of the system, works well only in the case of single source and has some problems when queries have to be distributed over several sources.

In particular, we are going to consider how query distribution works, starting from a SPARQL query issued towards the *Domain Ontology* of the CA system; that query will be *translated* by the *Rewriting Engine* in a way that is *Sound and Complete* but not usable by the SPARQL-query-to-SQL-query parser which is part of the CA system (which allows to translate only single target source SPARQL queries into SQL queries) since its components are declared onto different and multiple datasource ontologies.

Starting from these multi source rewritings, the algorithm presented in this thesis will obtain a *Complete set of Sound Queries* that the query execution module will be able to run on the SQL sources (since the SPARQL-query-to-SQL-query parser will be able to translate them in SQL).

After that, every single result coming from the query execution module goes back to the *Global Query execution module* which integrates all the results into the *Domain Ontology* and retrieves the set of integrated results.

3.4 Architecture

In this section we give an overview of the Context-ADDICT architecture, which is presented in Figure 3.1 . A detailed description of the entire system is presented in [13]. The exposition proceeds as follows: first we give a brief description of the principal data structures involved in the system, then we give

an overview of the architecture and finally we present the principal software components employed in CA.

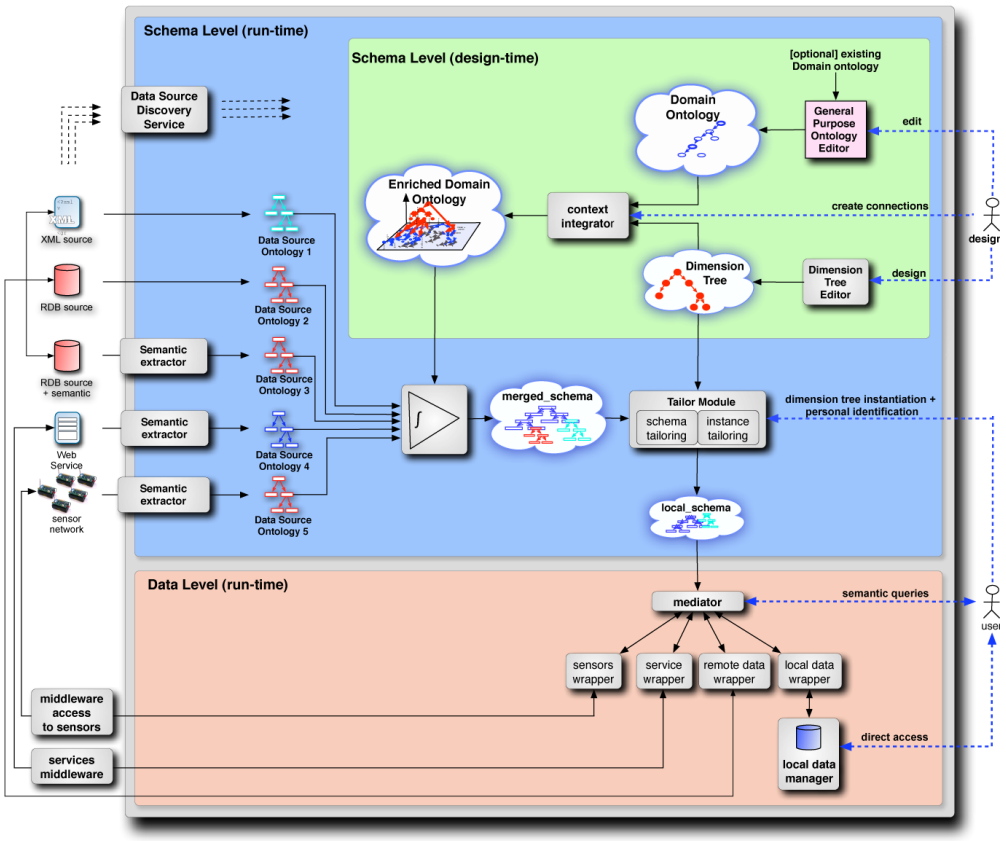

Figure 3.1: Context-ADDICT Architecture

3.4.1 Data structures

In this section we give a brief description of the principal data structures employed in CA. These notions will be used to give an overview of the entire system in the following sections.This exposition is given according to [35].

Domain ontology. The *Domain ontology (DO)* is a crucial component and its aim is to model the domain of the application. While developing the Domain ontology, the designer has to take into account all the concepts and relations which are relevant for the particular domain of interest. This step is a very important one because all the rest of the system, from the integration of data sources to the query answering process, depends heavily on the quality and on the formalism used during the design of the Domain ontology. The Domain ontology has to describe the application domain in all its relevant parts , while Data source ontologies, will usually describe only a part of the domain in a more specific way . An implicit assumption of CA is that, for the most of the cases, it will be possible to create mappings between Data source ontologies and the Domain ontology.

Dimension tree. The *Dimension tree* is a model aimed at describing the user and the context in which the user interacts with the system. This model is specified in [13] and it is described independently from the specific implementation (e.g. OWL, DTD). The Dimension tree aims at modeling the context in which a given user interacts with the system. The model represents a context as composed by some *dimensions* such as: space, time and holder (i.e. , the type of user interacting with the system). For each of those dimensions the model specifies also a sub-tree which represents the specific granularities of a particular dimension. From the Dimension tree we can derive different contexts (given by a specific assignment to each one of the dimensions) that the designer has to associate to appropriate parts of the Domain ontology (i.e. , a context configuration specifies a chunk of the Domain ontology that represents the portion of domain relevant to the context configuration itself).

Enriched Domain ontology. The *Enriched Domain ontology* is an ontology which contains classes and relations coming from the Domain ontology

and the Dimension tree. The Enriched Domain ontology also contains relations between the Dimension tree and the Domain ontology and the metadata that will be used to instruct the *Tailoring module* during the phase of *Data tailoring*.

Data source. In this work, we consider *Data sources* as sources of information which can be: RDBMS, XML, RDFS, sensor networks etc. In other words, we consider heterogeneous data sources.

Data source ontology. A *Data source ontology* DSO_i is a semantic description of a given data source (DS_i). This semantic description can be presented to CA by the data source itself (in this case we call the data source *cooperative*) or has to be automatically extracted from the data source (in this case we call the data source *non cooperative*). In the latter case we have also the problem of translating (at run-time) queries expressed on the semantic description of the data source to queries expressed in the native format of the data source itself. The problem of query translation is resolved using a *mapping* which explains how semantically described data (at the Data source ontology level) is actually stored inside the data source.

Mapping ontology. A *Mapping ontology* is a set of axioms used to link terms belonging to the Domain ontology with terms belonging to a Data source ontology. Formally, $\forall i$, where DSO_i is a Data source ontology, we have a Mapping ontology M_i which describes the relationships between DSO_i and the Domain ontology DO. In the following, we will eventually refer to Mapping ontologies with the notation M_i. The language was quickly presented in the Chapter 2.

Merged Schema. The *Merged schema* is the ontology resulting from the integration of Data source ontologies and the *Enriched Domain ontology*

Local Schema. The *Local schema* is the output of the Tailoring Module which takes as input the Merged Schema and returns a view, called the Local schema, defined considering informations provided by the Dimension tree instantiation. Moreover, for each *non-collaborative* data

source, the Local schema contains metadata which will be used to translate queries in the original data format.

3.4.2 Overview

Referring to Figure 3.1 we can split CAs architecture into two main sections which are the *Schema level* and the *Data level*. The Schema level is further divided in two distinct sections (corresponding to different phases of the systems life cycle) : *Design-time* and *Run-time*. Design-time Schema Level refers to the processes of Domain ontology development and Dimension tree instantiation, while Run-time Schema level refers to the processes of data source integration and *Data Tailoring*. Data level is the run-time part of the system which deals with the problems of synchronization of data sources and query answering over the integrated schema (the queries can be either local or remote).

3.4.3 Software modules

In CA, various software modules are employed in the different phases of the systems lifecycle.

General Purpose Ontology editor. At design-time, during the ontological engineering step (for example during the Domain ontology modeling step) it is employed a standard ontology editor, such as Protégé [21] or Swoop [28].

Context integrator. This tool assists the designer during the definition of the relations that hold between the Dimension tree and the Domain ontology and in the definition of the Domain ontology chunks for each Dimension tree configuration.

Semantic extractors. These modules are used to deal with non-cooperative data sources. Their main goal is to extract a semantic description of a particular data source (i.e. , to extract a Data source ontology for a given data source). A module for each type of *non-cooperative* data source has to be developed. For the case of relational data sources an automatic extraction module, named ROSEX, has been developed in [35]

Ontology mapping. This module deals with the problem of finding mappings between Data source ontologies and the Domain ontology in a (semi) automatic way. X-SOM (eXtensible Smart Ontology Mapper) is a software component that has been developed for this purpose [16]. X-SOM executes mainly two different steps:

Ontology matching: where two different ontologies are compared in order to find similarities between them. This process takes into account different modules to find similarities, such as syntactic-based, structural-based and Bayesian network-based modules whose results are subsequently weighted by some function (such as weighted average or a neural network).

Ontology mapping: where two different ontologies are mapped using informations from the previous step and OWL predicates [41] such as `owl:equivalentClass` and `owl:equivalentProperty` or `rdfs:subClassOf` and `rdfs:subPropertyOf`.

Tailoring Module This module uses information from the *Enhanced Domain ontology* to produce the Local schema (i.e., it uses informations about the users context to tailor the Merged schema).

Local Data Manager. Deals with the problem of storing data inside the users device . This component differs depending on devices computational power . For example, in case of workstations with high computational resources, *Local Data Manager* can be either an RDBMS or an XML processor, while on a small PDA the Local Data Manager has to be developed ad-hoc, for example according to [12].

Query Processing Subsystem. This module has to sync different data sources and addresses the following issues:

Query decomposition. [2] Considering the Local schema, which is the schema obtained by the integration of Data source ontologies and the Domain ontology subsequently tailored by the Tailoring module, we have a tailored model which consists of a part of the Domain ontology and parts of Data source ontologies.

[2] This module will be called *rewriting engine*, in the follow up of this document.

Query distribution. [3] To obtain the data relevant for the user we have to query this model distributing the query over the different Data source ontologies. As already said, this thesis focuses on the distribution part of the query processing subsystem trying to solve the issue of rewriting a query Q, expressed in terms of the Domain ontology , in terms of Data source ontologies.

Query translation. Once we have distributed the query over each Data source ontology, in the case of *non-cooperative* Data source , we have to translate the query Q into the native data sources query language, as explained above.

Data Integration Subsystem. After each query has been executed onto the original datasource, the system must collect all the result and must integrate them together, in order to return the final, complete answer to the final user.

3.5 Goals of this thesis

The final output of this thesis will be the implementation of the prototypes of two new modules, *DiSPARQL*[4], which is the completion of the *Query Processing Subsystem*, and the *data integration module* that allows the generation of the final answer to the original query.

The insertion point of the dispatching module is after the rewriting engine and before the query execution modules.

The integration module is placed as the final module of the whole chain of processing, after the execution module, and it will be also charged of sending the final results to the user.

In Chapter 6 we will describe all the theories and algorithms that are useful for the implementation of this two new modules, while in Chapter 7 we will take a closer look to the effective implementation of *DiSPARQL* and *Data integration module*.

[3] This module will be called *dispatching engine* or *distribution engine*, in the follow up of this document.

[4] **Di**spatching **SPARQL** which is our "*dispatching engine*".

3.6 Summary

In this chapter we have introduced the architecture of the Context-ADDICT system, giving an overview of its basic data structures and software modules. More- over, we have clarified what is the aim of this thesis inside the Context-ADDICT project.

Chapter 4

State of the Art

4.1 Introduction

Data integration system are actually one of the most open branches in which *DB* researches are moving into.

Moreover, as it has been already said in the introductory chapter of this thesis, nowadays researchers are very interested in mediated data integration techniques.

This is because, in addition to the relational schemas integration, sometimes could be also necessary to integrate other, heterogeneous, data sources, such as *XML* and the *WWW* itself.

In this chapter we are going to see which are actually the solution used in order to make data integration focusing into two main areas: *RDF/RDFS based data integration systems*, which represents the starting point of our work with the CONTEXT-ADDICT system and *ontology driven query distribution algorithms*, which is the main interest of this thesis work.

Since the aim of this thesis is to study a query distribution module which will be used to semantically distribute queries in a data integration system, we take into consideration also works such as Piazza [23] which, though considering XML data integration in a P2P environment, address the problem of query answering through query rewriting.

As proposed in [44] we can group various methods in categories which are different in the way ontologies are employed. In particular, here we unify

the definitions (given in [44]) of multiple-ontology approaches and hybrid approaches. Doing this distinction we have:

Single ontology approaches: where we have a single global ontology G and we have to specify mappings between each data source D_i and the global schema G. This approach is good when we have sources which contain data that provide nearly the same view of the domain. If we have a data source containing data at a different level of granularity w.r.t the global schema G then creating a mapping can be very difficult.

Multiple ontology approaches: In multiple ontology approaches we usually have a single Data source ontology DSO_i for each data source D_I, a global schema G and a mapping between the two. Recently, multiple ontology approaches can be composed also by Data source ontologies DSO_i which are mapped one to each other in a way similar to *P2P* environments. The advantage of multiple ontology approaches is that they permit to specify the mapping between the source and its own ontology without considering the rest of the system, so this method in well suited for information integration issues where there is frequently the need of adding and removing sources from the integration framework.

Moreover, we can divide ([40]) different approaches according to the type of the mapping we have between sources and the global schema G. Again, we have three different approaches (for further details, see Section 2.6.2 and [27]):

GAV, Global-As-View.

LAV, Local-As-View.

GLAV.

All these type of mappings have been already described in Chapter 2.

Given these definitions, we can consider some relevant works in ontology-based data integration systems.

4.2 Information Manifold

Information manifold is a DIS based on the CARIN Description Logic [32]. Information Manifold (IM) was the first system that addressed the problem

of data integration using LAV mappings. Indeed, the goal of the Information Manifold was to provide a uniform query interface to a multitude of data sources, thereby freeing the casual user from having to *locate* data sources, *interact* with each one in isolation and manually combine *results*. The problem of query answering inside Information Manifold was addressed through query rewriting and LAV mappings were handled by the bucket algorithm [25] which was the first bucket-based algorithm employed in order to answer queries using views. Moreover, the IM addressed the problem of specifying mappings using conjunctive queries (which was the same formalism used to query the system) and addressed also the problem of considering inclusion dependencies specified over the global schema. The query answering process consisted of two steps which were the query rewriting phase (which considered the problem of rewriting a query Q into a query Q' using map- pings) and the query evaluation phase which considered the problem of evaluating Q' over the sources.

4.3 RDF/RDFS based relational database integration

An interesting approach is an integration framework based on the RDF/RDFS formalism proposed in [15]. By the definitions we stated before, we can define their approach as a single ontology approach based on LAV mappings. In [15] the concept of RDF views is used to specify mappings between local sources and the global schema. RDFS constructs such as `rdfs:subClassOf` or `rdfs:subPropertyOf` are used to specify integrity constraints (role and class inclusions) over the global schema. Since the sources are expressed in terms of the global schema (LAV), the problem is that of answering queries using views [24]. To better understand what is intended by an RDF View [15] we consider the following definition:

Definition: A typical RDF view is in the form:R($\bar{X}$)- : G($\bar{X}$, $\bar{Y}$)

1. R($\bar{X}$) is the head of the view and R is a relational predicate.
2. G($\bar{X}$, $\bar{Y}$) is called the view body and G is a set of RDF triples with some node replaced by variables.

3. $\bar{X}$, $\bar{Y}$ contain variables or constants. Variables in $\bar{X}$ are called distinguished variables, and variables in $\bar{Y}$ (those in the bodys view but not in the heads view) are called existentially-quantified variables.

Given this definition it is possible to formalize mappings in the form of RDF views between every relational source and the target ontology (the global ontology is a FOAF ontology[1]). In the following, we see an example taken from [15] where a legacy employee database (from Zhejiang University) is mapped to the domain ontology (FOAF). Although in [15] they formally describe the semantics of the query answering as the result of evaluating the (original) query over a *Target RDF instance* constructed from RDF view definitions and RDFS constraints, they subsequently address the problem of query answering by query rewriting. Given a set of RDF view definitions which describe the sources in terms of the global ontology, the algorithm performs the following steps:

- Start from view definitions and extend those definitions with integrity constraints (this is done to enrich the query using semantic constraints in order to answer more types of query). For example, semantic constraints can state that both foaf:schoolHomepage and foaf:accountServiceHomepage are `rdfs:subPropertyOf` foaf:Homepage and both foaf:OnlineChatAccount and foaf:OnlineEcommerceAccount are `rdfs:subClassOf` foaf:OnlineAccount.

- Group triples in the view definitions by subject name, that is, parts of the body of a view definition which have a common subject are grouped together, creating the so called *class mapping rules*.

- Next the algorithm skolemizes ([6]) triples, substituting all existential variables with their associated skolem functions.

- Construct and optionally merge class mapping rules (merging rules means for example that if there are multiple rules for a given class,those rules are merged).

[1]The Friend of a Friend (FOAF) project: `http://www.foaf-project.org` .

After this step there are a set of rules which can be used to rewrite the original query $Q1$ (there is also a step of grouping and Skolemization on the query $Q1$ similar to the steps commented before).

Next the algorithm begins to look for rewritings for each triple group by trying to find applicable class mappings. If it finds one it rewrites the triple group by the head of the class mapping rule and generates a new partial rewriting. Finally, they show also soundness and completeness of the global query answering algorithm.

An interesting approach in the direction of semantic access to data sources is that of D2R [10], a mapping language which maps relational schemata to RDF triples. Through the D2R-Server it is possible to query or even browse with semantic or common browsers the tuples of a database as if they were RDF documents.

4.4 Description logic and ontology based data access

In [14], [7] and [29] the topic of interest is finding a formalism based on Description Logics that allows query answering within PTIME data complexity; to this aim several subsets of the OWL language have been developed to find the best trade-off between expressive power and computational complexity of sound and complete reasoning and query answering. In particular, DL-Lite [14] is a fragment of OWL2- DL which allows to delegate query evaluation to a relational engine; this language has been refined into other dialects such as $DL-Lite_A$ or $DL-Lite_F$, that extend the core DL-Lite language with different combinations of DL constructors while preserving FO-reducibility [30] in order to be able to efficiently perform tasks such as consistency checks and query answering. Such systems are especially useful to access relational data by querying the corresponding conceptual representation based on an ontology; this access pattern is usually denoted as ontology-based data access (OBDA). However, some of the approaches to OBDA [33] [19] are based on languages that exceed the expressiveness of first-order queries and, in order to ensure completeness of query answering, they require either deep access to the data sources (i.e., in update) or enough temporary memory to produce those infer-

ences that cannot be captured by first order queries (e.g., transitive closure). This may be a problem when accessing dynamic and transient data sources.

4.5 The Piazza PDMS

Classical mediator-based integration systems have problems in evolving environments where source schemas change over time. In [23] they point out that frequently in a data integration system, a mediated schema becomes a bottleneck for the entire system. In such an environment (a mediator-based data integration system) data sources cannot change significantly and during evolution they might violate the mappings to the mediated schema. In some sense the typical flexibility of the Web (where new pages can be authored,uploaded and quickly linked to existing pages) is missing, and as a result data integration systems pro- vide limited support for large-scale data sharing. The vision of a PDMS (peer data management system) is to blend the extensibility of the HTML Web with the semantics of data management applications. In such a PDMS there is no hierarchy of mediators but the framework supports any arbitrary relationship between peers; an important point is that Piazza exploits relationships (between peers) in a transitive manner. In [23] the logical model of a PDMS is given in terms of peer relations and stored relations. The difference between the two types of relations is similar to that of TBoxes and ABoxes in knowledge bases. *Peer relations* are relations of the peer schema which represent only the structure of a peer (like virtual schemas) while *stored relations*, like data sources in a data integration system, also contribute data to the system. In [23] there are two types of mappings: mappings between *peer schemas* and *stored schemas* (called storage descriptions) and mappings between peer schemas and peer schemas (called peer descriptions). Peer descriptions define the correspondences between the views of the world at different peers. Storage descriptions on the other hand, map the data stored at a peer into the peers view of the world (indeed these mappings have the same function of mappings in a DIS). Stored relations are analogous to data sources in a data integration system. Summarizing, in contrast to a hierarchical data integration environment, which has a tree-based hierarchy with data sources schemas at the leaf nodes and one or more mediated schemas as intermediate nodes, a PDMS can support an arbitrary graph of interconnected schemas. In

particular, we have introduced Piazza because we are interested in its query reformulation algorithm [23]. The query reformulation algorithm takes as input a query Q and a set of peer mappings and storage descriptions. The output is a query expression Q' that refers only to stored relations. An important assumption made in [23] is that all the peer mappings are available at a single location, and hence the reformulation is done in a single place. The algorithm presented in [23] is sound and complete which means that the evaluation Q' will always only produce certain answers to Q (certain answers are found in polynomial time). Q' is called the maximally-contained rewriting of Q: that is a query over the sources that produces all the answers to Q that are possible from any query Q. In the Piazza PDMS the query reformulation algorithm is seen as the construction of a rule-goal tree: goal nodes are labeled with peer relations, while rule nodes are labeled with peer mappings. Mappings can be both GAV and LAV mappings so the first challenge of the algorithm is to combine and interleave the two types of reformulation techniques. The first type of reformulation (GAV) replaces a subgoal with a set of sub-goals, while the other (LAV) replaces a set of sub-goals with a single sub-goal. In the case of LAV mappings an algorithm of answering queries using views [25] is employed. The algorithm will produce the query rewriting by building a rule-goal tree, while it simultaneously marks certain nodes as covering not only their parent node, but also their uncle nodes.

4.6 Highly Dynamic DIS

A step toward more dynamic DISs is represented by [22], an infrastructure for query answering over distributed ontologies. The approach virtually integrates distributed and autonomous ontologies using ontology meta- data contained in distributed registries. Query answering is performed through rewriting into disjunctive Datalog programs with mappings expressed using DL-Safe rules [38] that are much more expressive than those used in [14]. However, they only refer to ontological data sources and they only consider manually-designed mappings. A further contribution toward highly dynamic DISs is the Active Ontology (*ActOn*) project [45] which is an ontology-based information integration system for highly dynamic distributed sources. *ActOn* is based on a set of domain ontologies, that describe the meta-data information model,

and a single ontology describing all the available information sources which are accessed through both dynamic and static wrappers. Differently from our approach, the wrappers are not context-aware and the mappings are restricted to manually-designed GAV mappings only.

Chapter 5

Semantic Extraction of Relational Data Sources

Since our distribution algorithm has been studied in order to deal with *distributed*, *non cooperative* sources (*cooperative* sources, such as ontologies and RDF files, could afford with multiple sources queries), we are going to introduce some details about the semantic extraction from a relational data source, an operation which represent the beginning of the whole integration process.

Concepts exposed in this chapter are borrowed from [35] and [43].

5.1 Ontological Extensions for Relational Databases

In this section, we discuss the automatic generation of an ontological description of a relational schema with key constraints and inclusion dependencies [6]. This process provides the relational data source with an enriched, semantic description of its content that can be used both for documentation purposes through annotation (e.g., during schema evolution [17]) and to achieve interoperability in an open-world scenario (e.g., in data integration [13]). This enriched description provides an infrastructure to access and query the content of the relational data source by means of a suitable query language for ontologies such as SPARQL.

The access infrastructure to the relational data source consists of three ontologies which are used to describe different aspects of its structure:

Table 5.1: The Relational.OWL ontology

Relational.OWL Classes		
rdf:ID	**rdfs:subClassOf**	**rdfs:comment**
dbs:Database	rdf:Bag	Represents the relational schemas
dbs:Table	rdf:Seq	Represents the database tables
dbs:Column	rdfs:Resource	Represents the columns of a table
dbs:PrimaryKey	rdf:Bag	Represents the primary key of a table
dbs:ForeignKey	*rdf:Bag*	*Represents the foreign key of a table*

Relational.OWL Properties			
rdf:ID	**rdfs:domain**	**rdfs:range**	**rdfs:comment**
dbs:has	owl:Thing	owl:Thing	General composition relationship
dbs:hasTable	dbs:Database	dbs:Table	Relates a database to a set of tables
dbs:hasColumn	dbs:Table *dbs:PrimaryKey* *dbs:ForeignKey*	dbs:Column	Relates a tables, primary and foreign keys to a set of columns
dbs:isIdentifiedBy	dbs:Table	dbs:PrimaryKey	Relates a table to its primary key
dbs:hasForeignKey	*dbs:Table*	*dbs:ForeignKey*	*Relates a table to its foreign keys*
dbs:references	dbs:Column	dbs:Column	Represents a foreign-key relationship between two columns
dbs:length	dbs:Column	xsd:nonNegativeInteger	maximum length for the domain of a column
dbs:scale	dbs:Column	xsd:nonNegativeInteger	scale ratio for the domain of a column

- *Data Model Ontology (DMO)*: represents the structure of the data model in use. This ontology does not change as the data source schema changes, since it strictly represents the features of the data model such as the logical organization of entities and attributes. For the relational model, we adopt the Relational.OWL ontology [18] whose structure is shown in Table 5.1. Since the current version of Relational.OWL does not distinguish between *composite* (foreign keys that references more than an attribute at the same time) and multiple foreign keys, we extended the Relational.OWL with explicit foreign keys. Our extensions to the Relational.OWL ontology are italicized in Table 5.1[1].

- *Data Source Ontology (DSO)*: represents the intensional knowledge described by the data source schema. This ontology capture the conceptual schema (ER-like) from which the relational schema under analysis is derived. The DSO does not contain individual names (instances), which are stored in the DB and accessed on-demand.

- *Schema Design Ontology (SDO)*: this ontology maps the DSO to the DMO and describes how concepts and roles of the DSO are rendered

[1] multiple domains are considered in union.

in the particular data model represented through the DMO. This ontology enables the separation of the schema's metadata (by means of the SDO) and schema's semantics (described by the DSO). We remark that, in general, the SDO can be extremely useful during schema evolution, because it describes how the changes in the relational schema are going to affect the semantics of the schema itself by detecting changes in the conceptual model.

Note that, despite in this paper we focus on relational data sources, it is easy to see that the same infrastructure can be straightforwardly replicated in different settings, in order to access data stored under different data models, i.e., by means of a DMO designed for that data model. To this, the appropriate SDO must be associated at design-time, dictating how the elements of the data source schema are rendered in the ontology.

The extraction procedure first generates the DSO by applying a set of rules whose preconditions and effects are shown in Table 5.2. Concepts and roles of the DSO are then connected through mappings of the SDO to the corresponding concepts of the DMO.

5.1.1 DSO Extraction

An ontology can be defined as a 5-tuple $\langle N_C, N_R, N_T, N_I, A\rangle$ where N_C is a set of concept names, N_R is a set of role names, N_T is a set of attribute names (i.e., roles whose range is a concrete domain), N_I is a set of names for individuals (i.e., constants) and A is the set of axioms of the theory. In this work we assume ontologies whose semantics is given in terms of Description Logic formulae [8].

We now describe in more detail the extraction process of the DSO starting from a given relational schema extended with key constraints. Note that, without loss of generality, we assume that attribute names be unique within the database. The relational schema is represented as a 7-tuple $\mathcal{R}$=$\langle \mathcal{R}, \mathcal{A}, \mathcal{D}, att, dom, pkey, fkey\rangle$ where:

1. $\mathcal{R}$ is a finite set of n-ary relation schemata;

2. $\mathcal{A}$ is a finite set of attribute names;

3. $\mathcal{D}$ is a finite set of concrete domains (i.e., datatypes);
4. *att* is a function which associates to a relation schema $r \in \mathcal{R}$ its set of attributes $\{a_1, a_2, \ldots, a_n\}$;
5. *dom* is a function associating each attribute to its concrete domain;
6. *pkey* is a function associating to a relational schema $r \in \mathcal{R}$ the set of attributes of its primary key (with $pkey(r) \subseteq att(r)$ and $pkey(r) \neq \varnothing$);
7. $fkey$ is a function associating to a relational schema $r \in \mathcal{R}$ the set of attributes which are (part of) foreign keys in r (with $fkey(r) \subseteq att(r)$).

In the following, we use the notation C_r to refer to a concept of the DSO obtained from the translation of the relation schema $r \in \mathcal{R}$, R_a to refer to a role obtained from the translation of an attribute $a \in att(r)$, while we denote the domain and the range of a role R by $Dom(R)$ and $Ran(R)$ respectively. Moreover, we use the notation $r(a)$ to denote the relational projection ($\pi_a r$) of an attribute a of a relation r. We now discuss the translation rules in Table 5.2 in order to provide their rationale.

Some of the rules (namely `R1` to `R4`), address the translation of relational tables by taking into account their primary keys and their relationships with other tables through foreign keys. `R1` generates new concepts in the ontology for each table with at least one proper primary key (i.e., which is not also a foreign key) which correspond to strong and weak entities in an Entity-Relationship schema. `R2` takes as input a relational table with arity $n > 2$, where all the attributes composing the primary key are also foreign keys. Such tables correspond to ER relationships which cannot be directly translated into a (binary) role and are *reified* by means of a new concept representing the association. In addition, a new role is generated for each attribute in the table.

Each of these roles has as domain the concept obtained by reifying the association, and as range the concept obtained from the translation of the referenced table. It is worth noting that the application of the reification does not guarantee the uniqueness of the individuals belonging to the reified concept (which correspond to the tuples of the relational table). However, this is not a problem if we do not allow updates, since the constraint is already enforced by underlying relational engine. `R3` takes care of two-column

Table 5.2: Relational to ontology translation rules

Rule	Preconditions	Effects
R1	$\exists r \in \mathcal{R}$ such that: $\mid pkey(r) \mid = 1$ or $\mid pkey(r) \mid \geq 1$ and $\exists a \in pkey(r) \mid a \notin fkey(r)$	a concept C_r a role R_a $\forall a \in pkey(r)$ *and* $a \notin fkey(r)$ an axiom $C_r \equiv \exists R_a.dom(a)$ $\forall a \in pkey(r)$ *and* $a \notin fkey(r)$
R2	$\exists r_i \in \mathcal{R}$ such that: $\mid att(r_i) \mid > 2$, $\forall a \in pkey(r_i), a \in fkey(r_i)$, $\forall a \in fkey(r_i) \exists r_j \mid r_i(a) \subseteq r_j(b) \wedge b \in pkey(r_b)$ for some $b \in att(r_j)$	a concept C_{r_i} a role $R_a \forall a \in pkey(r_i)$ an axiom $C_{r_i} \equiv \exists R_a.C_{r_j}$ $\forall a \in pkey(r)$ an axiom $Dom(R_a) \sqsubseteq C_{r_i}$ $\forall a \in pkey(r)$ an axiom $Ran(R_a) \sqsubseteq C_{r_j}$ $\forall a \in pkey(r)$
R3	$\exists r_i \in \mathcal{R}$ such that: $\mid att(r_i) \mid = 2$, $att(r_i) = pkey(r_i)$, $fkey(r_i) = pkey(r_i)$, $\exists r_j, r_k \in \mathcal{R}, \exists a_1, a_2 \in att(r_i) \mid r_i(a_1) \subseteq r_j(b) \wedge r_i(a_2) \subseteq r_k(c)$ for some $b \in att(r_j)$ and $c \in att(r_k)$	a role R_{r_i} an axiom $Dom(R_{r_i}) \sqsubseteq C_{r_j}$ an axiom $Ran(R_{r_i}) \sqsubseteq C_{r_k}$
R4	$\exists r_i \in \mathcal{R}$ such that: $\mid pkey(r_i) \mid \geq 1$ and $\exists a \in att(r_i) \mid a \in pkey(r_i) \wedge a \in fkey(r_i)$, $\exists r_j \in \mathcal{R} \mid r_i(a) \subseteq r_j(b)$ for some $b \in att(r_j)$	a role R_a an axiom $C_r \sqsubseteq \exists R_a.C_{r_j}$ $\forall a$ declared as not null an axiom $Dom(R_a) \sqsubseteq C_{r_i}$ an axiom $Ran(R_a) \sqsubseteq C_{r_j}$
R5	$\exists r_i \in \mathcal{R}$ such that: $\exists a \in att(r_i) \mid a \notin pkey(r_i) \wedge a \in fkey(r_i)$, $\exists r_j \in \mathcal{R} \mid r_i(a) \subseteq r_j(b)$ for some $b \in pkey(r_j)$	a role R_a such that $Dom(R_a) \sqsubseteq C_{r_i}$ and $Ran(R_a) \sqsubseteq C_{r_j}$, an axiom $C_r \sqsubseteq \exists R_a.C_{r_j}$ $\forall a$ declared as not null
R6	$\exists r \in \mathcal{R}$ such that: $\exists a \in att(r) \wedge a \notin fkey(r)$	an attribute T_a with $Dom(R_a) \sqsubseteq C_r$ and $Ran(R_a) \sqsubseteq dom(a)$, an axiom $C_r \sqsubseteq \exists R_a.C_{r_j}$ $\forall a$ declared as not null

tables where primary and foreign key attributes coincide. Differently from R2, this association is translated into a new binary role, whose domain and range are the concepts obtained from the translation of the referenced tables. R4 completes the translation of tables where the primary key consists of foreign key attributes and non-foreign key attributes. These tables are the result of the translation of *weak entities* and are already rendered as concepts by R1. However, the resulting concept must be related to the concept corresponding to the strong entity providing the key for the weak entity. This connection is rendered as a new role whose domain is the concept corresponding to the weak entity, and whose range is the concept corresponding to the strong entity.

The two remaining rules (i.e, R5 and R6) address the translation of all the table columns which are not covered by the above rules. In particular, R5 renders all the foreign key attributes of a table - which are not part of the primary key - as new roles whose domain and range are the source and the referenced table respectively. R6 takes care of all the remaining table columns which are not part of a foreign key. The effect of R6 is the creation of a new attribute for each column that matches the rule's preconditions. The domain of this attribute is the concept resulting from the translation of the table and the range is the corresponding (or a compatible) concrete domain.

5.2 Summary

In this chapter we have seen how the semantic extraction step is performed when we deal with relational data-sources and which are the rules involved in the translation.

The process will also produce as output all the ontologies necessary in order to perform the backward translation of the query (from SPARQL to SQL) that will be performed by the query execution module.

Chapter 6

Query Processing in Context-ADDICT

In this chapter we are going to review how the query distribution algorithm works, presenting all the features which it needs in order to do correctly its work. We are going to present two heuristics which allow us to reduce sensibly the number of queries which we are going to distribute over each source.

In the next Chapter (Chapter 7) we are going to see how this algorithm is implemented in our system.

In both this two Chapters we are going to refer to a basic case in which all our data sources are *Relational* and *non-cooperative* sources.

6.1 Overview of the process

As a preliminary step, the system needs that each datasources make available an own *semantic description*, written in an ontology form (the so-called *Data Source Ontology*). Before beginning the query rewriting process, it is also necessary to draw (automatically with *X-SOM* in case of simple mapping's types or manually in case of complex mapping's types, like GAV or LAV) the mappings between the enriched domain ontology and every data source involved in the process; the mapping ontologies must be defined using the $CA - DL$ logic.

Then, starting from the query issued towards the *Domain ontology*[1], the

[1]From now on this query will be called *original query*.

system starts the query rewriting and distribution process by *Reasoning* and then mapping the concept contained into the domain ontology to their related concepts of the DSO.

After this has been done automatically by the rewriting engine, the system generates the rewritings of the original query by combining each possible element that can be mapped on the DSO in order to create queries that can be answered by the DSO.

However, the product of the rewriting engine is a set of *Sound and Complete* queries; each single query could be referred to multiple DSO; while this is good, for instance, for queries issued over ontologies, this is not good for queries issued over relational sources; in this final case we need that a query correspond exactly to one and only one source.

So for our final purposes, we can say that we need a *Complete* set of *Sound* queries; this means that every single query has to be sound (and also complete towards the single source, not towards the original query), with respect to the original query, and the whole set of queries has to be complete, which means that running the whole set of queries will provide the same resultset as running the original query.

After having this query set (the one which is possible to dispatch over each single source), system runs it over each source (for relation data sources there is an apposite module, called *SPARQLExplorer*, which allows to translate SPARQL queries into SQL queries and to retrieve data coming from a relational source in an "*ontological*" format.

Then the system simply runs the original query over the domain ontology, which meanwhile has been enriched with the result that comes from each DSO, and gives to the user the final, integrated, results.

The whole process is represented in the following Figure 6.1.

Looking at Figure 6.1 we can understand how the system works in order to respond to a query issued towards the system by a user, and shows which are the modules involved in the whole process.

In particular we can see from the figure that we have three different queries which flows through our system:

1. A query Q which is the original query, issued by the user towards the enriched domain ontology of our system.

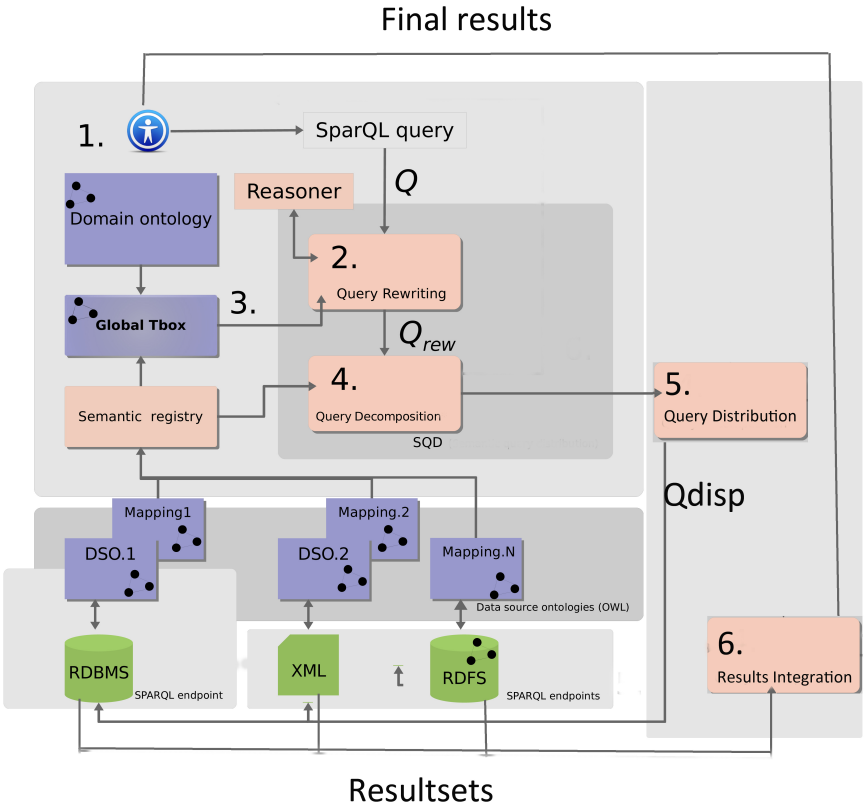

Figure 6.1: Module Architecture of CONTEXT-ADDICT platform

2. A set of queries Q_{rew} which is the output of the rewriting engine and, after the tailoring made by the query decomposition module, goes to the query distribution engine.

3. A set of queries Q_{disp} which is the final set of single-source queries that can be sent to the query execution engines.

The *query rewriting* and *query decomposition* modules are subparts of a more general system that is called *Semantic Query Decomposition* system (SQD)[2], which has been deployed in [27].

[2]Formerly, in [27], the acronym stays for *Semantic Query Distribution* but, since the system lacks of the distribution feature, it has been renamed *Semantic Query Decomposition*, not to confuse it with our newly implemented *Query Distribution Engine*.

In the figure the query execution modules are not represented since it is obvious that, once we need to process a query of the set Q_{disp} on a specific data source we need to use the proper execution engine.

The output of this thesis are the two modules in the rightmost part of the figure, the *query distribution* and *results integration* modules, which work is crucial in order to allow the system to work properly exploiting all of its functionalities.

6.2 Useful structures

In order to perform the query distribution task we need to define two useful data structure, which we will use during the whole process: the *hyper-graph* structure and the *meta-query* structure.

6.2.1 Hyper-graph structure

The hyper-graph structure we will use for the query distribution has the following characteristics:

- it will contain a set of vertexes, which we need in order to keep in mind which are the result variables of the query;
- it will contains a set of hyper-edges, which represents the query pattern.

More details about the implementation of this structure can be found in Section 7.2.1.

6.2.2 Meta-query structure

The *meta-query* model is a simple structure which allows our system to proceed in the real distribution of the query.

The main components of this structure are:

- a field which indicate the DSO to which the model refers to
- the set of the variables used in the query
- the set of the result variables (which is a subset of the previous one)

- a structure which make easily possible to understand which variables makes join together (in the follow-up we will call that structure *Local Joins*)
- the set of elements (triples and filter clauses) included in the query

In this way, distributing queries become more easily.

Like Hyper-graph structure, also in this case more details about the implementation of this structure can be found in Section 7.2.2.

6.3 Heuristics

In order to reduce the numbers of queries produced by the system, we define two heuristics, *Similarity*, which allows us to understand if a query is similar to another (which means that probably the two queries will retrieve the same resultset of data), and *Containment*, which allows us to understand if a query is probably included in another one.

6.3.1 Similarity

This heuristics allows us to say that if a query is "*similar*" to another one, which means that all of the following clauses are satisfied:

- the two queries are issued against the same DSO
- the two queries have the same set of result variables
- the two queries involves exactly the same predicates in the *WHERE* clause

In this case we can drop one of the query, keeping the other in our set.

6.3.2 Containment

This Heuristic allows us to drop one of the two compared query if we can assert that all the following statements are true:

- the two queries are issued against the same DSO

- one of the two queries contains the same set of result variables of the other one plus, at least, another variable
- one of the two queries contains the same set of result predicates in the *WHERE* clause of the other one plus, at least, another predicate

In this case we allow our algorithm to drop the queries which is contained, keeping the one that contains it.

Notice that it would not necessarily means that the query which is contained in the other one is less general than the other. Probably it would not. Therefore, if we are interested in a smallest set of results than the one which this would retrieve, it is not useful to retrieve the whole result set, which only delays the time at which the final user get the answer to the original query.

The algorithm used in order to apply the containment heuristic to our set of queries is the one described into *Algorithm*1 and *Algorithm*2 (we need to split since it is too long to be reported into a single page).

It is possible to notice that the algorithm works with meta-queries, instead of queries, so we can re-define the heuristic as:

- the DSO are the same in both queries
- all the sets which compose the first meta-query are subset of the sets which compose the second one

In this case we can obviously drop the contained query.

6.4 Feasibility of a query

During the rewriting process, the only concern of the rewriting engine is to make *sound* and *complete* queries. This means both that the product of this process could be a multi-source query, but also could be a query in which two variables, on which we declare a property *prop1* that has domain x and range y, could be declared of type t and z, because maybe on some sources *prop1* could be rewritten as a property that has domain x and range y and on some other source *prop1* could be rewritten as a property which has domain t and range z.

Algorithm 1 isContainedInto Algorithm - Part 1.

Require: a *meta-query mq* that will be compared to the one on which the method has been called

Ensure: `True` if the query on which the method has been called is contained into *mq*; otherwise `false`

```
    if not this.sourceName is equals to mq.getSourceName then
      return false
 3: end if
    if not this.triples size ≤ mq.triples size and not this.triples is empty then
      for all elements in this.triple do
 6:     extract an element from the vector (lets call it current element)
        if not mp.triples contains the current element then
          return false
 9:     end if
      end for
    else
12:   return false
    end if
    if not this.used_variables size ≤ mp.used_variables size and not
    this.used_variables is empty then
15:   for all elements in this.used_variables do
        extract an element from the vector (lets call it current element)
        if not mp.used_variables contains the current element then
18:       return false
        end if
      end for
21: else
      return false
    end if
```

In order to understand better, lets consider the following example: suppose we have a general property *drives* that has domain *person* and range *vehicle*. Suppose that class vehicle has two subclasses *car* and *moto*, and also *drives*

Algorithm 2 isContainedInto Algorithm - Part 2.

```
    if not this.result_variables size ≤ mp.result_variables size and not
    this.result_variables is empty then
      for all elements in this.result_variables do
3:      extract an element from the vector (lets call it current element)
        if not mp.result_variables contains the current element then
          return false
6:      end if
      end for
    else
9:    return false
    end if
    if not this.joins_variables size ≤ mp.joins_variables size and not
    this.joins_variables is empty then
12:   for all elements in this.joins_variables do
        extract an element from the vector (lets call it current element)
        if not mp.joins_variables contains the current element then
15:       return false
        end if
      end for
18: else
      return false
    end if
21: return true
```

could be specialized into *drivesCar* and *drivesMoto* (obviously the domain remain the same as *drives*, but the range changes respectively into *car* and *moto*).

Suppose we have two datasources: one in which we can find data about persons which drives car and the other one that contains data on person which drives motorcycles.

Lets suppose we send to the system the following query to be rewritten: variable x is of type person, variable y is of type vehicle and x and y are

"*joined*" by the *drives* property.

We expect to obtain as the output of the *rewriting engine* two queries in which once y is of type *car* and the other time is of type *moto* and in the first case we have the *drivesCar* property while in the second we have the *drivesMoto* one.

However, it is possible for the rewriting engine, in order to have a complete query (which at a quick analysis it is not sound), to create a query on which x is of type *person*, y is of type *car* and the property is *drivesMoto* (and vice versa).

This would lead to an inconsistent query, which the system must drop, in order not to fake the result of our query process.

In order to verify if a query is feasible or not we apply the following algorithm (*Algorithm*3):

Algorithm 3 Feasibility Check Algorithm

Require: a meta-query MQ, the list of all properties contained in all the DSO (with their domain and range) P and a list of all the subclasses of each class contained in all the DSO C

Ensure: return `true` if the query is feasible, otherwise `false`

```
for all elements e in MQ.elements do
  if e is a Triple then
    if e is an OntProperty then
      for all properties p in P do
        if p.name = e.predicate.path then
          if e.subject is a variable then
            if not e.subject.type = p.domain and not e.subject.type =
            "NONE" and not e.subject.type is a subclass of p.domain
            then
              return false
            end if
          end if
          if e.object is a variable then
            if not e.object.type = p.range and not e.object.type =
            "NONE" AND not e.subject.type is a subclass of p.range
            then
              return false
            end if
          end if
        end if
      end for
    end if
  end if
end for
return true
```

6.5 Filter clause and Order By clause

As a preliminary decision is useful to point out some choices made about the handling of the Filter and Order By clauses.

Filter clause It is clearly understandable that it would be useful to push the *Filter* clause on every source (on which it is possible to use it) of our DSO set since this operation allows us to retrieve a smaller (because it is filtered) resultset from each DSO, while not pushing it on the DSO can bring, in some case, many useless result (since after the application of the filter clause they would be destroyed).

So the final decision is to push every filter clause on every DSO it can be pushed to.

Group By clause Otherwise, pushing the *Group By* clause to every source it is useless, since it is not useful to order each single resultset coming from the DSOs and then re-order the whole integrated resultset[3].

So the choice would be to process the order by clause only at the end of the integration process, when the result integration module have already put together every result and runs the original query.

6.6 Query distribution and result retrieval process

After that the rewriting module has produced the rewritten query set, it is time to distribute the query over every single DSO.

This job is done by the dispatching algorithm in three steps, starting from a single rewritten query of the set produced by the rewriting module[4]:

1. starting from the rewritten query, the dispatching engine produces a *hyper-graph* model of the query, which is the beginning point of the distribution

2. than, processing this hyper-graph, the engine produce a set of dispatched queries, using another meta-model (the *meta-query* data structure); each query produced by the engine is directed to each single data-source

[3] we can not suppose anything about the ordering between sources!

[4] In the following of this chapter we are going to refer to an element of this query set as *rewritten query*.

3. after having all the queries that could be dispatched over the sources, the engine makes some optimizations, using the previously introduced heuristics

Once we have the output of the dispatching engine, it is possible to send each query to the matching DSO, and by means of the right execution module (it should be one for every different type of source), the system execute the query.

Each execution module should be able to insert the result of the query execution into the DSO on which the query is stated, so it is possible to include each execution result back in the domain ontology. If the execution module it is not able to make this, the module is useless, since it probably can retrieve the final result but it cannot give them back in an understandable format to the *Result Integration* module.

The *Result Integration* module simply get back from the execution modules a set of DSO enriched with the results of the query; then, it pushes these results into domain ontology enriching it. The final step is running the original query onto this domain ontology enriched with results, and return the final result of the integration process to the final user.

6.6.1 Query distribution

In this section we are going to see how the dispatching engine works[5]

First step: Construction of the hyper-graph

The input of this step is a single rewritten query (one included in the set of the ones that were generated by the rewriting engine).

The output of this step will be a hyper-graph representing the structure of the queries.

The *hyper-graph* model Since our needs regarding the hyper-graph model are particular and also "basics" (we mean that we do not need any strange or particular complex operation to be made on the graph), we decide to create our own model of hyper-graph which has been already exposed in Section 6.2.1.

[5]For details about the implementation of the engine, see Chapter 7.

In addition to the previously defined hyper-graph model we need to introduce 3 standard operations:

- it should be possible to add or remove a vertex
- it should be possible to add or remove a *hyper-edge*
- it should be possible to retrieve the sets of vertexes and hyper-edges

Each vertex has a label (usually the variable of interest), while each hyper-edge contains, in addition to the label, the set of vertexes of which the relation is composed.

At this point we can refine the *Similarity* heuristic: we can say that two queries are similar if their hyper-graphs are similar.

This means that, defining an appropriate method on hyper-graphs like, for instance an *hashcode* which takes into account our specification of *similarity*, we can simply drop similar queries using a *hashmap - like* data structure, without any further and complex reasoning.

The *hyper-graph* generation Starting from the rewritten queries, the hyper-graph generation algorithm should return a valid hyper-graph of it, by visiting the query pattern.

The steps that the algorithm have to do are the following:

1. insert the result variables of the query into the set of vertexes of the graph
2. visit the pattern of the query unfolding each element contained in it and, once the visit reaches a triple, go to the next step
3. once the element visited is a triple, the algorithm has to add it as a hyper-edge to the hyper-graph, providing also to associate to each edge the correspondent vertexes representing the variables involved in the definition of the edge.
4. until there is something to visit, restart form 2

In ontologies world, the - *arity* of an hyper-edge should be one of the following:

- *1 - arity* if the edge represents a triple of the kind: `?var rdf:type dso:class`
- *2 - arity* if the edge represents a triple which contains a property (*DataType* or *Object*)
- *n - arity* (with $n \in (1, \infty)$) if the edge represents a filter clause

The algorithm used for the hyper-graph construction is the following one (Algorithm4).

Second step: from the hyper-graph to the meta-query

In order to distribute a query over the several *DSOs*, it is not enough to produce the hyper-graph, but some more processing is needed. So we need to move from a hyper-graph to another model, which is the *meta-query*[6].

First optimization: dropping similar hyper-graphs Before moving to the next step the engine will parse all the set of rewritten queries and it will generate the matching set of hyper-graphs.

Using the similarity heuristic, is now possible to drop similar queries in such a way to reduce the hyper-graph set which we are going to use as starting point for the next steps, and reduce response times of the whole system.

Distribution algorithm: first step - splitting The first part of the distribution algorithm receives as input an hyper-graph and gives as output a primary version of the meta-query which later will need some more processing, in order to be sure that we are generating sound queries.

As it starts, the procedure generates a set of meta-query which only have the DSO field set, for each DSO involved in the distribution process; later the procedure will fill the respective meta-query with every element present in the graph which belongs to it.

The algorithm visits randomly an edge of the ones included in the hyper-graph under examination[7].

[6] Basic definition of this structure has been already exposed in Section 6.2.2.

[7] Remember that the set of vertexes included in the hyper-graph has the purpose to store the result variables.

Algorithm 4 Query Grapher algorithm.

Require: a *rewritten* query Q
Ensure: a set of graph GS
create an empty set of hyper-graphs SG
create an auxiliary stack structure AS
create a new, empty hyper-graph G
for all result variables in Q **do**
add the result variable to the vertexes of G
end for
unfold the query pattern of Q, filling AS
for all elements in AS **do**
if element is a Triple **then**
if the predicate of the triple is equal to `rdf:type` **then**
add an unary hyperedge to G
else
add a binary hyperedge to G
end if
else
if element is a filter clause **then**
add an n-ary hyperedge to G
else
if element is a String **and** element is equal to `NextGraph` **then**
add G to SG and create a new, empty graph G
end if
end if
end if
end for
return SG

Before starting to split the query, the dispatching module visits the rewritten query and build into its own memory a structure, which later will allows it to find incoherencies in the generated meta-queries. In the follow-up of this thesis we are going to refer to this structure calling it *Global Joins* structure.

Once the algorithm has "*pulled out*" an edge from the hyper-graph, it has to follow the following steps, to produce the right set of meta-query:

1. looking at the label of the edge, it has to choose the correspondent meta-query model from the previous generated set (understanding which is the DSO involved by comparing the DSO field of the meta-query with the path of the DSO included into the label of the edge)

2. once the first step is complete, and the triples corresponding to the edge has been inserted into the right meta-query, the algorithm should process the vertexes included in the edge, adding them to the meta-query used variables, and if they match any of the result variables, they should also be added to the meta-query result variables

3. in order to make possible further optimizations, if the visited edge represent a triple which include an OntProperty, the variables involved (that, in standard DB language are said to make *join*) should also be added to the meta-query *Join Variables* structure

4. instead of the previous two steps, if the edge represents a filter clause, at this step of the algorithm the only feasible operation is to add it to every meta-query of the set

After this procedure has been executed we have a set of meta-queries that maybe are not *sound*.

In order to perform a check on the soundness of the meta-queries, we must perform another step, which actually is a "*cleaning step*", which makes possible to fix all the unsoundness in our queries.

The details about the algorithm involved into this first step are described into *Algorithm*5 and *Algorithm*6 (since it is to long to be reported in a single page).

To properly understand how it is possible to retrieve the correspondent element from the hyper-edge of the hyper-graph, we need to anticipate a detail about the hyper-graph implementation[8]: in our internal representation of the hyper-graph, the hyper-edge label is an `Element` of the query pattern (a Triple or a Filter clause).

[8]All the details about all the implementations are going to be discussed in the following chapter, Chapter 7.

Algorithm 5 Basic Split Generation Algorithm - Part 1

Require: a hyper-graph H
Ensure: a set of meta-query MQ'
create an empty set of meta-queries MQ (one for each datasource), setting only the respective datasource name field
create an internal structure to keep joins over variables in the original query JV
create a new, empty hypergraph G
for all hyper-edge in the hyper-graph hyper-edge set **do**
if hyper-edge represents a Triple **then**
if hyper-edge represent a type declaration triple (it should include `rdf:type` as the predicate URI) **then**
set *isTypeDecl* to `true`
path = the URI of the object of the triple represented by the hyper-edge
else
path = the URI of the predicate of the triple represented by the hyper-edge
end if
{In order to see what happens here, please check *Algorithm*6}
else
add the filter clause to the m.elements `ElementGroup` {if we are in this branch of the if, hyper-edge represents a Filter clause}
end if
end for
MQ = generateFinalSplits(JV)
for all m in MQ **do**
if isFeasible(m) **then**
MQ'.add(m)
end if
end for
return MQ'

Algorithm 6 Basic Split Generation Algorithm - Part 2 (inner for)

```
    for all m in MQ do
      if path contains the m.dataSourceName then
3:      add the triple represented by the hyper-edge to the m.elements
        ElementGroup
        if isTypeDecl then
          add the subject of the triple represented by the hyper-edge to
          m.used_variables list, setting also its type, provided by the path
          variable
6:      else
          add the subject of the triple represented by the hyper-edge to
          m.used_variables list, without setting its type
        end if
9:    end if
      if hyper-edge cardinality = 2 then
        add both the object and subject variables to both JV and
        m.joins_variables list
12:   end if
    end for
```

Distribution algorithm: second step - cleaning The cleaning step is fundamental in order to obtain sound queries, since during the previous step (also if we begin our process from a sound and complete query) the "*splitting*" of the query over multiple DSO may have introduced some incoherence in the single sources queries (such as, for instance, the loss of a join over two different variables).

Moreover, we may also have filter clauses where they are useless, so they are going to be removed from that meta-query.

The procedure start by visiting the set of triples included in the meta-query structure.

The algorithm can follow two different procedures, if the element we are considering is a Filter clause or if it is a Triple.

In the first case, the engine needs to retrieve the set of the variables used

into the filter clause. If this set is included into the set of meta-query used variables, the choice would be to keep the filter clause into the set of elements of the meta-query (in this case it is feasible, and useful, to keep it).

Otherwise the engine must purge the filter clause, since keeping it would lead to an unfeasible and not runnable query.

Instead, if the algorithm is evaluating a triple, we must do a simple step: looking at the global joins structure (built before starting the splitting phase), and comparing it with the used variable meta-query set, we have three possibility:

1. the meta-query used variables set contains zero or only one of the variables which originally makes join: the meta-query is consistent (for this join)

2. the meta-query used variables contains both the variables which originally makes join and the meta-query local joins structure also contain them (i.e. the global joins which we are evaluating is contained into the local join structure): the meta-query is consistent (for this join)

3. the meta-query used variables contains both the variables which originally makes join but the meta-query local joins structure does not contain them (i.e. the global joins which we are evaluating is not contained into the local join structure): the meta-query is not consistent (for this join) and need some fixes

Obviously this reasoning has to be made for each element included into the global joins structure.

As stated in the previous list, while case 1 and case 2 do not need any further processing, case 3 needs another step of processing.

In this last case indeed we need to make another split of the query, in order to obtain two distinct queries containing distinctly the two variables which originally make join[9].

The others tuple may go indistinctly in one of the new queries, unless there is some joins on any query's other variable (the algorithm keeps trace of it)

[9]Not performing this step usually leads to produce a query in which the original join has been replaced by a cartesian product.

which suggest to which of the two generated queries the other tuples has to be moved.

The algorithm we use in the "*cleaning*" step is the one described into emphAlgorithm7 and *Algorithm*8 and it represent the *generateFinalSplits* method called at line 17 of emphAlgorithm5.

Algorithm 7 Final Split Generation Algorithm - Part 1

Require: a set of meta-query MQ and a set of original query joins variables set JV
Ensure: a set of sound meta-query MQ'
create an auxiliary meta-query AUX
for all meta-queries mq in MQ **do**
if not mq.isEmpty **then**
for all j in JV **do**
if mq.used_variables contains both the two variables of j **and** mq.joins_variables contains j **then**
Add to AUX.joins_variables j
for all variables v contained into j **do**
add v to AUX.used_variables
end for
see *Algorithm*8 to understand what happens here
else
if not AUX.isEmpty **then**
add AUX to MQ'
end if
splits mq into mq1 and mq2, both instanced on the same data-source, performing also the Filter clause check as in the previous branch.
end if
end for
end if
add AUX to MQ'
end for
return MQ'

Algorithm 8 Final Split Generation Algorithm - Part 2 (inner for)

```
for all elements e in mq.elements do
  if e is a Triple then
3:  if e is a variable type declaration involving one of the variable included
    in j or e is a property which involves both the variables in j or e is a
    property which involves one of the variables in j and a constant or e
    involves only constants then
      add e to AUX.elements
    end if
6:  else
    if e.MentionedVars is contained into mq.used_variables then
      add e to AUX.elements {We are considering a Filter clause}
9:  end if
  end if
end for
```

At line 15 of *Algorithm*7 the algorithm simply separate the two problematic variables type declaration[10] putting them into two separate distributed queries, fixing this type of problem. Other elements that are included into the query are put into the second generated one, although for any other reason they should be included in the first ones (for instance because there is a join or because we are going to filter the variable included into the first query).

Distribution algorithm: third step - optimization At this point, we have to use the second of the heuristics that we have previously defined and refined.

After having the whole set of meta-queries it is possible to make some more reasoning on it and drop, using the containment heuristic, queries that might lead to large and useless resultsets, reducing the system response times (fewer queries to be run and fewer result leads directly to response time reduction).

[10]If we are at this line of the algorithm it means that the distributed query contains both the type declaration of the two involved variables, while it misses the property that at the origin concatenates them.

Distribution algorithm: fourth (and final) step - query generation After having split, cleaned, and optimized (and also dropped empty queries contained into) our meta-queries set, it is now possible, using the information contained into each one of them, to generate the correspondent *SPARQL* queries set, which is the ones of which the elements, passed to the executions modules, are going to be parsed, translated and executed onto the original DSO.

6.6.2 Result retrieval and integration

The *Result Retrieval and Integration* module has a double job:

- it is the *end-point* for each execution module, which means that, every execution module submits its own result (integrated into the DSO of the source that it has queried) to it
- after receiving the results, the module has to "*reverse*" the mappings that usually are used to map concept from domain ontology to DSO and use them to map concepts (and individuals) from DSO to domain ontology, in order to obtain an integrated knowledge base to query, to get the final, integrated results.

The first job is quite simple, since it has not any particular purpose. It is a pure data collection task.

The second step works with the mapping already defined and used by rewriting engine (using them in reverse mode), so also in this case there are not peculiar or strange features; it simply performs the following steps:

1. grab an individual from a DSO enriched with results (the one given back by an execution module)
2. understand which is the DSO class of this individual
3. map this DSO class with the corresponding one into the domain ontology
4. put the individuals into the domain ontology, associating it to the corresponding class found at previous step
5. repeat this process for every individual and every results' enriched DSO returned by the execution modules

After the integration step it is possible to execute the original query[11] (the one stated onto the domain ontology), including also any eventual order by clause and obtain the final result that now has only to be returned to the final user.

6.7 Summary

In this Chapter we have provided a high-level view of two of the new modules that need to be implemented into the *Context-ADDICT* system, which are the *Query Distribution* and the *Result Retrieval and Integration* modules.

These two modules are going to become two of the core modules of the system, and their implementation will be presented into the next chapter (Chapter 7).

[11] While performing this step, we have already made some little modification to the original query, dropping every filter clause because we have already filtered each single data source result.

Chapter 7

Design and Implementation

In this chapter we are going to present how the algorithms and the features described in the previous chapter are implemented into the CONTEXT-ADDICT system and which are the interfaces involved in the communication between the already existent modules of the system and the new ones.

All the algorithms mentioned into this chapter have been already exposed into the previous chapter (Chapter 6) and we will refer to them only by reporting the respective algorithm number.

7.1 Introduction

The CONTEXT-ADDICT system architecture presented in Chapter 3 is composed essentially by 4 modules:

ROSEX the Relational to Ontology Semantic EXtractor is the module charged of obtaining a semantic description of the relational data sources[1]

X-SOM eXtensible Smart Ontology Mapper[2] is the module which allows finding automatically simple mappings between DSO_i and *Domain Ontology*

RewSPARQL [3] the module which rewrites queries issued towards the *Domain ontology* in a format understandable by DSO_i

[1] The process has been quickly explained in Chapter 5.

[2] Described in [39]

[3] The so-called *rewriting engine*.

SPARQLExplorer this module allows to translate a SPARQL query into a SQL query (it is the query execution module for the relational datasources)

We are going to add two new modules to our system, which are:

DiSPARQL "*DIspatching SPARQL*" is the module which distributes the queries over the various DSO_i; its position into the system is between *RewSPARQL* and the various query execution modules

Result Retrieval and Integration it is the module which integrates all the results, obviously placed after the various query execution modules

All this modules are written in *Java* language. In the following sections of this chapter however we are not going to give the Java encoding of our algorithm, but we will present the algorithm implementation in a more detailed way and we will present the key features of each algorithm.

Three sub-modules compose the *DiSPARQL* module:

- The *Main* sub-module, the one that coordinates all the operations of the other two
- The *Graph Generator* sub-module, which generates the hyper-graph of the rewritten query under examination by *DiSPARQL*
- The *Query Distribution* sub-module, the one which generates the final set of queries that we are going to distribute over our datasources.

Before giving the details about the modules implementation, we are going to see some helpful structures implemented into the new modules which we will help us to present some features of the algorithms presented.

7.2 Common and useful structures and algorithms

In this section we are going to give a quick look on some useful structures that come together with the new modules; this structures have a more general

purpose definition than the one strictly used in the new modules (they can also be helpful in already implemented ones, for optimization purposes), so we have decided to present them separately, before presenting their usage into the new modules.
Every structure is coded as a class into the package:
`it.polimi.elet.contextaddict.util`.

7.2.1 hyper-graph implementation

For our particular case of interest, since *Java* lacks of a good hyper-graph implementation and since we are going to deal essentially with SPARQL queries, we have decided to implement an ad-hoc solution for the hyper-graph model.

The choice we made is to implement a *Vertex* class which has only an attribute of type `Node` that is the LABEL of the vertex; then we have the *hyper-edge* class that includes the two attributes LABEL, of type `Element` and a vector of vertexes called ELEMENTS, which represents the set of variables included into the hyper-edge.

Both Vertex and hyper-edge implement the standard (and ad-hoc) constructors, getters and setters and override the default *toString*, *hashCode* and *equals* method; hyper-edge also implements a method to add to an edge the vertexes included one-by-one and a method to retrieve the cardinality of an edge.

The *hyper-graph* class contains two vectors:

- one containing the vertexes of the graph (that we will use in our algorithm in order to keep trace of the result variables of the query)
- one containing the edges the graph, which represents the real structure of the graph.

Besides all the default (and ad-hoc) constructors, getters and setters, the override of the default *toString*, *hashCode* and *equals* method, the `HyperGraph` class implements some ad-hoc methods to add vertexes, edges without vertexes, edges with vertexes, retrieve an `Iterator` for the edges vector and finally a method to understand if the graph does not contain any edges.

All the methods which allow to add something (vertexes or edges) to the hyper-graph can throw a specific exception (named

`AlreadyContainsElementException`) in the case that the object we are adding is already present into the graph.

This is the final implementation of *hyper-graph* that has been used into the *DiSPARQL* module.

7.2.2 Meta-query implementation

In order to define the meta-query implementation, we need two other auxiliary structures, which are the *JoinVar* and the *UsedVar* class:

JoinVar is the class that allows handling variables that make join. The attributes of the class are two strings, which contain a variable name. Besides the standard (and ad-hoc) constructors, getters and setters and the override of the default *toString*, *hashCode* and *equals* method, the class implements a method that, receiving two variables name as input, return `true` if the variables represent a join

UsedVar is the structure used by the distribution module[4] in order to remember which are the variables used into the query and of which type they are[5]

The ***meta-query*** implementation is probably one of the key of the functioning of the distribution algorithm; it has three vectors attributes (containing respectively all the *used variables*, the *result variables* and the *joins variables*, a string attribute which identify the source on which the query must be executed and an `ElementGroup` attribute, that will contain all the triples (and filters clauses) used into the query.

In this class we can find: the standard (and ad-hoc) constructors, getters and setters and the override of the default *toString*, *hashCode* and *equals* methods; Moreover there are three more methods, which will become very useful in our work:

[4] *Distribution module* is a synonym of *dispathcing engine*.

[5] Notice that, if the query does not contain any `rdf:type` declaration for a variable, it would have as type a defined constant, which is *NONE*; this means that we have no information about this variable type, and it is an useful information for the next steps of our algorithm.

- the **isEmpty** method, that returns `true` if the query result variables vector is "*empty*" (this means that, also if we run the query it will return an empty resultset)
- the **toQuery** method, which allows us to produce the `Query`[6] from our meta-query representation (this operation is necessary since, once we have distributed the queries, passing by the hyper-graph and meta-query models, we need to return back to a valid SPARQL Query)
- the **isContainedInto** method which is the implementation of the *containment* heuristic. Since this method is very peculiar for our optimization purposes, we are going to discuss it in a little bit deeper way into the next section (Section 7.2.2).

the isContainedInto method

This method has to manage the choice to drop a certain query in behalf of another one if the second one contains the set of variables and clauses with some more ones respect to the ones contained into the first one. This usually means that the first query will return a *super-set* of the resultset that the second one will give back (and usually the resultset returned by the first query would contains many useless data). Both the query has, obviously, to be stated onto the same DSO.

The algorithm used in order to make this choice is the one represented into *Algorithm*1 and *Algorithm*2.

7.2.3 PropertiesTable and SubClassesList

PropertiesTable is the structure which target is to make possible to create a list of all the properties, with their own *domain* and *range*, that would be helpful in the cleaning phase of the distribution algorithm.

SubClassesList is the structure, used in order to check domain and range properties consistency in the query distribution's algorithm cleaning phase, which simply maps all the eventually subclasses of a certain ontological class.

[6]The SPARQL version of the query.

It contains the list of all available subclasses for each class included in the *DSO*.

7.3 Query Distribution algorithm

The whole process of query distribution, as it was explained into the previous chapter, is made of two main steps: the creation of the hyper-graph of the *rewritten query* and the distribution of the query onto the various datasources.

This two steps involves algorithms implemented into two classes: the hyper-graph generation is handled by the *createGraph* method of the `QueryGrapher` class, while the query distribution and cleaning tasks are performed by *generateAllBasicSplits*, *generateFinalSplits* and *isFeasible* of the class `QueryDispatcher`.

The query containment heuristic is used in the main DiSPARQL module, after all the methods above has been used, to obtain the final set of query that we are going to distribute over our datasources.

As we have already said in Section 7.1, besides the main module of DiSPARQL we have the graph generator and the query distributor module; in the follow-up of this section we are going to explain which are the algorithms (described in Chapter 6) implemented in each of them, in order to allow them to work properly and give us the result that we expect.

We are going to present the main module only after we have seen how the other two modules works, since the main module has only to coordinate the execution of the other two modules and to use the two heuristics defined in 6.3 to optimize our result.

7.3.1 The query grapher sub-module

The query grapher module task is essentially to take in input a rewritten query from the set of rewritten queries produced by the rewriting engine and output a hyper-graph whose pattern reflect the original pattern of the query.

This operation is executed into the *createGraph* method that has to:

1. add all the result variables of the query to the set of vertexes of the hyper-graph

2. visit all the pattern of the rewritten query and, once it find a *triple element* or *filter element*, has to create the corresponding edge and add it to list of edges of the hyper-graph

The method `createGraphs` is the implementation of the algorithm described in *Algorithm*4, Section 6.6.1.

If the rewritten query contains any *UNION* clause, the algorithm must produce a graph for each query included into the union.

This will not generate any inconsistency in query processing, because we are working with queries that are of the type "*disjunction of conjunctive queries*", so we can split union queries when we send them on the datasources; we have only to remember that we must perform a union when we will perform the integration step.

In order to help us with the graph generation work, we are going to reuse some useful methods which allows us to easily visit the rewritten query implemented in the rewriting engine; this methods are the two *unfolds* methods and more details about them could be found in [27].

7.3.2 The query distribution sub-module

The query distribution module task is composed by two sub-tasks:

1. firstly it must distribute the query over each source mentioned in the query;

2. then it has to make some consistency check and optimizations on it, in order to be sure that we are going to dispatch a *sound* query and that we are not going to retrieve wrong or useless result from the datasources.

As we have already seen in Chapter 6, unfortunately, we cannot make the two operation into a single step, but we need multiples passing through the hyper-graph or through the meta-query.

So, the first step is to take the hyper-graph and put each element of the query pattern into the meta-query structure that is going to be sent to the DSO that is mentioned somewhere in the path of the element.

The module had previously generated a meta-query structure for each possible involved datasource.

While doing this, if the examined element is a binary edge (which means that we have a join between two variables in the *original* query), the algorithm must keep updated an internal structure, which traces the original joins (we need this structure in order to make the consistency check in the next step).

If the element examined is a filter clause, at this step of the processing we are not able to understand if it is necessary or not if pushed to a determinate datasources. Our choice at this point of the processing is to push it onto all the datasources, then when the algorithm generates the final version of our query will decide to push or not it (at this moment the algorithm could clearly understand it).

The method `generateAllBasicSplits`, contained in this module, is the implementation of the algorithm described into *Algorithm*5 and *Algorithm*6 in Section 6.6.1.

it is possible to notice, looking at lines 17 and 19 of *Algorithm*5, that is the basic split algorithm which is charged to call the other two methods (*generateFinalSplits* and *isFeasible*) and than it returns back to its caller (which is the *Main* sub-module of the distribution module) the final (and *sound*) set of meta-query.

The further steps of the process are necessary since, after this preliminary step of distribution, our distributed queries could still be unfeasible or unsound.

The first check we are going to perform is a *soundness* check. We are going to verify if we have preserved the joins of the original query into each distributed ones[7]; if we have lose any join and if the variables that make join are both included in the same distributed query, it means that we have changed a join request in a cartesian product request.

This will lead to wrong answers to the original query, since we are going to give back results that we are not interested into.

This does not mean that every single query of the distributed query set must contain every join present in the original query. This only means that, if one of the distributed queries contains two variables which make join in the original query, these two variables must make join also in the distributed query which contains them.

[7]From now on, we are going to refer to this set as *distributed queries set*; therefore we are going to call a single element of this set *distributed query*.

Moreover, at this point of the computation, it is possible to understand if a filter clause can be usefully inserted into a distributed query, since we have information about variables used in each distributed query.

We can leave a filter clause into a distributed queries if and only if the whole set of variables mentioned into the filter clause is contained (or equals) into the set of the distributed query used variables, otherwise it must be purged out of the final distributed query version.

The *generateFinalSplits* method, which is the other method included into this module, is the implementation of the algorithm described into *Algorithm*7 and *Algorithm*8 in Section 6.6.1.

The next check we need to perform is the feasibility check; it signify that we need to check if there are not differences between the variables declared type and any domain/range type of any eventual property which involves the variables themselves.

The only admitted difference is that the variable declared class is a sub-class of the domain/range allowed class.

In order to do this, before the module starts its own work, it needs an initialization phase in which (using two of the useful structure discussed in Section 7.2, `PropertiesTable` and `SubClassesList`), by visiting all the DSO, builds the two structures in which the algorithm will find any useful information for the feasibility check.

If the algorithm finds an unfeasible query, the algorithm simply drops it, since it cannot be executed on the datasources or may lead to useless, ambiguous results.

The method we are going to use to perform this is the implementation of the algorithm described into *Algorithm*3 in Section 6.4.

After this process has been performed, we obtain our final set of deliverable queries.

One objection to the order in which we process and drop unfeasible query can be that it is possible to see if a query is unfeasible since the beginning of the distribution process, looking at the original rewritten query.

This might not be true since, after the distribution process, elements of the query that may result in an "*original*" unfeasibility of the query may be pushed onto different DSOs, obtaining at the end a set of distributed **feasible** queries.

Also maybe only the part of the query distributed on a certain data source might be unfeasible, while the others are not unfeasible.

In the next section we are going to explain how the main sub-module of *DiSPARQL* works in order to coordinate the operation of the other two and which final optimization it is going to perform on our queries.

7.3.3 The main sub-module

The main sub-module is the one that receives in input the rewritten queries array from the rewriting engine.

The first operation that it will perform is to pass the elements contained in this array to the *query grapher* sub-module, which will return a set of hyper-graphs to be used in the follow up of the execution.

While receiving back hyper-graphs, by means of a *hashmap-like* data structure, the main sub-module apply the similarity heuristic and drops duplicates of the hyper-graphs it receives.

After that, processing a hyper-graph at once, it forward the execution to the *query distribution* sub modules, getting back results from it and storing always in another *hashmap-like* data structure (so if the splits of the queries had generated some similar query, the system will drop every duplicate of it).

After the previous step is completed, the algorithm applies to final resultant array the containment heuristic, dropping useless queries.

At this point we have our final, distributable set of queries.

7.4 Extensions to the SPARQLExplorer module

Since the *SPARQLExplorer*[8] module has been produced before the other main modules of the system had been specified, which needs some extensions on its interfaces to properly do its work.

In particular, at the beginning of our thesis work the module cannot return the results it gets from the relational datasource in an ontology format

[8]Some detailed note on SPARQLExplorer implementation and usage can be found in [37] and [43].

comprehensible by the integration module (i.e. the *SPARQLExplorer* module will not "insert" its result into the *DSO*).

In order to follow the integration module specification, *SPARQLExplorer* has been extended by adding a method that, once called by passing a SPARQL query and a configuration file, returns its result by enriching the *DSO* whose name is contained in the configuration file by adding all the individuals which satisfy our query (an individual is produced starting from the SQL retrieved results).

This way it is possible to integrate the answers to each of our distributed query.

7.5 Data retrieval and Integration

As it was said into the previous sections, the data retrieval process is performed by each query execution module (e.g. *SPARQLExplorer* for relational datasources, direct interrogation of OWL and RDF datasources by means of *ARQ*), which will give back its result to the data integration module.

Once a resultset comes from a query execution module, the *integration* module starts the integration process:

1. First of all, by means of the mappings between *Domain ontology* and *DSOs*, the integration module translate a concept that is part of the DSO into the related concept of the domain ontology, adding the related *individuals*[9] to the domain ontology.

2. Once all the result has been collected into the domain ontology, the module (using *ARQ* runs the original query, issued onto the domain ontology, on the domain ontology enriched by results, obtaining the final, integrated answer to our original interrogation.

At this point the system can give the answer back to the user, at the conclusion of the whole integration process.

[9] An *individual* in an ontology is an instance of a determinate concept.

7.6 Examples of query distribution

In this section we are going to show some examples of how, starting from a query issued onto the domain ontology, we get our final distributed queries array.

We are going to show only the examples about the two particular cases of query dropping and query re-splitting, since we think that an example of the standard case would not interest much for the understanding of this thesis.

Domain ontology, mappings ontology and everything can be useful to understand these examples can be found in Appendix A

The original query (the domain query), we are going to consider is the following:

```
PREFIX rdf: ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩
PREFIX rdfs: ⟨http://www.w3.org/2000/01/rdf-schema#⟩
PREFIX do: ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#⟩
SELECT ?x ?y ?z
FROM ¡⟨ile:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl⟩
WHERE { {
    ?x do:drives ?y.
    ?y do:hasBrand ?z.
    ?x rdf:type do:Person.
    ?y rdf:type do:Vehicle.
    ?z rdf:type do:Manufacturer.
    ?x do:hasName ?name.
    FILTER regex(?name, "cust1")
    }
}
ORDER BY ?x
```

7.6.1 Example One: unfeasible query dropping

The first example starts from this rewritten query, coming as output of the rewriting engine:

```
SELECT ?x ?y ?z
WHERE
{
    ?x ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#drives_moto⟩
?y .
    ?y ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#
auto.manufacturer⟩ ?z .
    ?x ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-
RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente⟩ .
    ?y ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-
RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#auto⟩ .
    ?z ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-
```

RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand⟩ .
?x ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente.name⟩ ?name .
FILTER regex(?name, "cust1")
}

The distributed queries will be:

SOURCE: `rosex1SemanticONTO.owl`

SELECT ?y
WHERE
{
?y ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#auto.manufacturer⟩ ?z .
}

SOURCE: `rosex4SemanticONTO.owl`

SELECT ?z
WHERE
{
?z ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand⟩ .
}

SOURCE: `rosex2SemanticONTO.owl`

SELECT ?x ?y
WHERE
{
?x ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#drives_moto⟩ ?y .
?x ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente⟩ .
?y ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#auto⟩ .
?x ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente.name⟩ ?name .
FILTER regex(?name, "cust1")
}

This final query will be dropped since the domain of *drives_moto* could not be *auto*, so the query is unfeasible.

7.6.2 Example Two: query splitting

The second example takes into consideration the following rewritten query.

SELECT ?x ?y ?z
WHERE
{
?x ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#drives_moto⟩ ?y .
?y ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#moto.manufacturer⟩ ?z .
?x ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-

RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente⟩ .
?y ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#moto⟩ .
?z ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand⟩ .
?x ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente.name⟩ ?name .
FILTER regex(?name, "cust1")
}

The distributed queries will be:

SOURCE: `rosex1SemanticONTO.owl`

SELECT ?y
WHERE
{
?y ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#moto.manufacturer⟩ ?z .
}

SOURCE: `rosex2SemanticONTO.owl`

SELECT ?x ?y
WHERE
{
?x ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#drives_moto⟩ ?y .
?x ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente⟩ .
?x ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente.name⟩ ?name .
FILTER regex(?name, "cust1")
}

SOURCE: `rosex4SemanticONTO.owl`

SELECT ?y ?z
WHERE
{
?y ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#moto⟩ .
?z ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand⟩ .
}

As it is possible to see, during the distribution process, the properties which correlates the two variables x and y is stated in `rosex1SemanticONTO.owl` while the two type declarations are stated in `rosex4SemanticONTO.owl`.

In this case we need a further split of the query issued to `rosex4SemanticONTO.owl`: so we obtain the two queries:

SELECT ?y ?z
WHERE
{
?y ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-

```
RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#moto⟩ .
}
```

and

```
SELECT ?y ?z
WHERE
{
    ?z ⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#type⟩ ⟨file:///Users/Lele/Documents/workspace/TIS-
RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand⟩ .
}
```

In this way we do not have the join, but also we avoid retrieving the results of a cartesian product, thing that will fake our answers.

7.7 Summary

In this chapter we have seen the implementation of the algorithms described in the previous chapter, presenting their features and showing their peculiarity.

We ended the chapter presenting some examples of how the algorithms work, focusing on two particular cases.

Chapter 8

Evaluation

In this chapter we are going to present some evaluation made on the new *DiSPARQL* module and on the integration module.

This evaluation were made essentially by understanding which advantages and problems present the new implemented algorithms and also by some testing in some simple use case, providing it is not such specific that we may find peculiar results (for better or for worse).

8.1 Introduction

First of all we need to specify that, without using the new implemented algorithms, the CONTEXT-ADDICT system was not able at all to retrieve results from sources other than *OWL* or *RDF* files.

With the introduction of the new algorithms the system becomes able to perform queries onto datasources.

On this sources it is not usually possible, or maybe convenient (think of the case of making joins between two relational datasources, at the opposite sides of the world: maybe it is better, if not necessary, to get results separately from the two datasources and joining them at a higher level), to process query on multiple sources, so the distribution step becomes essential.

In our work we focus on distribution among distributed *Relational* datasources, which is the case that involves *SPARQLExplorer*, *ROSEX* and the other modules presented in this thesis.

8.2 Complexity

In this section we are going to point out the asymptotic complexity of the new implemented algorithms, trying to figure out if making some optimization on them can reduce it.

In our evaluation we must consider essentially two metrics: we need to evaluate time complexity, since the system has to keep a good level of interactivity (it must not require too much time to give the results of our request) and, since the amount of data that may be retrieved could be very huge, also spatial complexity must be kept under control.

8.2.1 Complexity analysis

First of all we are going to consider the hyper-graph generation algorithm.

Considering the time complexity of the algorithm we can notice that, while the query unfold operation used for the kick-off of the process is a recursive algorithm (one iteration for each element of the query pattern), the graph generation algorithm only iterates over the elements of the unfolded query pattern. So if we call n the elements of the query pattern the time complexity of the whole algorithm would be $2n$; this means that we have an asymptotic complexity of the order of $O(n)$.

Considering spatial complexity, we see that the algorithm uses some *stack* structures in order to keep the results of the unfold process, and an array of hyper-graph which contains the generation results.

Moreover, at a deeper look, it is possible to understand that only one of the *stack* it is full at the same time, and considering also that *stack* structures are stored into the heap (so they are composed by dynamic fields), we can consider that the algorithm has an overall asymptotic spatial complexity of $O(n)$.

Let now consider the first of the two algorithms used for query distribution, *generateBasicSplits*.

Let us refer to n as the number of hyper-edges which are part of the hyper-graph and to m as the number of datasources on which we are going to distribute our query.

Since we have that, for each hyper-edge we must check to which of the m

datasource belongs, we have an average time complexity (due to the nested cycles) of $O(nm)$.

With regard to the spatial complexity of the algorithm, we only need to store information about the distributed queries (each one composed by some of the n elements), we can approximate the asymptotic complexity with $O(mn)$.

Probably the more complex algorithm is the one that generate the final results, *generateFinalSplits*.

Considering the temporal complexity of this algorithm we can see that the asymptotic complexity of it is $O(mno)$ in the best case (no need of further splitting) while it becomes $O(m^2no)$, where m is the number of datasources, n is the number of elements included in the query and o is the number of joins in the query.

Instead, evaluating spatial complexity, we can see that the algorithm only needs to store data about the produced query, so it asymptotic complexity would be $O(mn)$.

The only other interesting complexity element that we are going to evaluate is the point where we apply the containment heuristic. Since, for every query contained in our distributable query set we need to check if it is contained in any of the other query of the set, we obtain that the algorithm has a temporal complexity of $O(n^2)$, while it has a spatial complexity of $O(n)$ in the worst case, where n is the number of queries in the distributable set.

Evaluating the complexity of the integration process we can find that the integration algorithm has a temporal asymptotic complexity $O(mno)$ where m is the number of individuals included in DSO enriched with results, n is the number of mapping concepts included in the mapping and o is the number of possible distinct rewrites of each mapping concept. Spatial complexity is not important since the algorithm use structures that were already used during previous step of the processing.

8.2.2 Consideration about the complexity

Since our intent is to produce not a real-time system but, at least, an interactive system, we need to ensure that we are going to answer to query issued by the final user in reasonable time, which means that query processing must not take hours, days or maybe months, but that the whole process has to be brought

over and completed in the vicinity of second or minutes, depending of the complexity of the interrogation.

As it is possible to understand by the previous evaluation, the first crucial point in our processing is the *generateFinalSplit* algorithm, which is the most complex from the time point of view.

However, since the operations that this algorithm makes are necessary for the whole process, and since there are not other ways to do it, we cannot rescind from using it.

Moreover, after some little testing, the algorithm does not behave so bad as it is possible to imagine (w.r.t. the information given in the previous section), giving answers in a quite little time[1].

The other crucial point is the integration algorithm that, like the previous one, has been already optimized for the work it has to do. However, also this algorithm does not behave so bad has we will expect, making its job in reasonable time, allowing our system to work in a good way.

At the end of this evaluation, we have understood that the system behave according to the requirements we have expressed on it allowing us to retain, after all the analysis and consideration made in this chapter, that it would on average behave as we desire.

8.3 Analysis on the test results

Some simple test has been done on the system, in order to understand if the system can be considered an interactive system and in order to check that the system works properly and retrieves the right results from the datasources.

This simple test were made by using different queries and different datasources schemas (the datasources type is always the relational one)

The results of test were always comply with the expected ones; from this we can deduce that we are moving in the right direction.

When the system was tested using a query that includes three datatypes declaration, two joins and a filter clause, providing the system maps it to three

[1]The set of queries used in the test are the ones that were produced by rewriting the original query presented in Section 7.6.

different *asymmetric*[2] datasources, we have seen that, starting from an array of 96 rewritten queries that are the output of the engine, our distribution module obtain a set of 6 queries, which were verified to be the only 6 useful ones.

Otherwise, the same query issued onto three different *symmetric* datasources reduce the set of queries to be used from over a hundred of queries to 3 (obviously the only three needed by the system).

This means that also the containment heuristic works well, since the obtained queries are the ones which are the ones that respect them, providing us to retrieve only useful resultsets from the datasources.

At the end of this small test phase, we can say that the new implemented algorithms seems to work as we expect and they should bring correct results with every possible query that may be sent to the system.

Measurements made during this test highlights that the greatest percentage of system's computation time still remain the one involved in the rewriting engine reasoning time (which always states around 40-45 % of the total execution time), while operations made by our new algorithms seems not to be as complex as it was possible to suppose with our preliminary complexity study.

The only other "huge" operation that our system makes is the data integration operation that, after several tests, seems to absorb around the 20-25 % of the total execution time, while (in the simple tests that we have carried out) the serial execution of the two algorithms (*generateBasicSplits* and *generateFinalSplits*) only require around 2-5 % of the total computation time, allowing us to assert that with algorithm we are moving in the right direction, for our integration purposes.

Obviously, all the data shown in the last paragraphs of this section were given as percentage on the total computation time; this thing will mean that, more complex becomes the query more time will be elapsed by the whole system in order to process the original request[3].

Also as it is easily possible to understand that the integration time depends directly from the number of the result that we have to integrate together; it

[2]Here with *asymmetric* we intend to say that the tree datasources schemas contains different part of the information used by CONTEXT-ADDICT to retrieve the results, while with *symmetric* we intend that the datasources schemas are the same.

[3]At the moment the integration module has not already been implemented in a parallel architecture, that may reduce this "*integration* time".

is necessary a more intensive test in order to understand if we can reach some maximum value (that we will not exceed anytime) or if it still vary depending from the previously exposed dependencies (as it will probably be).

This may not be considered a problem, since also in *SQL* interrogation, response times depend from the quantity of the data that the system has to retrieve; the only thing that we must consider is to keep this time as little as possible (possibly a linear or quadratic dependency from the number of results).

8.4 Summary

In this chapter we have evaluate some results and exposed some consideration about the expected and the real complexity of the new algorithms implemented in the system.

After all this consideration, we have supposed that our system will now react correctly to the interrogation that users may make to it.

Chapter 9

Conclusions and Future Work

9.1 Original Contributions

As it was already described in the previous chapters, the main objective of our work is to produce a *query distribution module* which has to return a set of single-source *complete* set of *sound* queries starting from a set of *sound* and *complete* set of multi-source queries.

During the development of the new modules it become clear that also some optimizations on the number of queries to be sent to the execution's engines must be done, since many useless and, redundant queries could be generated by the rewriting engine.

Also sometimes it was necessary to drop some of the *rewriting engine* outcoming queries since they are incorrect and, once ran on the datasource, they may return wrong results (or they would lead the whole execution module to fail, since the parse could not translate the query), faking the results of our integration process.

After all these considerations were made, the new implemented query dispatching engine has been designed and developed into a prototype, respecting all the previously stated requirements.

Since a "*integration*" module lacks in the system, we have also proposed a solution to this problem, implementing this module in order to complete the *CA* system functionalities.

9.2 Further Remarks

Probably many of the aspects of the work exposed in this thesis may be criticized, starting from the many steps that the proposed algorithm has to make in order to distribute a single query. Unfortunately, as we already said in Chapters 6 and 7, all of these steps are necessary (e.g. it is not possible to evaluate if a query, after the distribution, keeps the original joins until the query has been distributed), because of the structure of a generic *SPARQL* query and of the consequent modifications to be performed on it.

Another choice that might be criticized is the choice of the heuristics we have made. Someone can say that sometimes verify if a query is contained in another one could be more time consuming than sending the query itself to the sources and retrieve the result for the integration step.

This, luckily, is not true, since we have already seen in Section 8.3 that the most critical point of the newly developed algorithms is the data integration step, since the wider it is the result space, the longer will be the integration time.

However, for everyone who prefers not to use the optimization made by containment heuristic, the usage of it has been parameterized, and it could be disabled (obviously it is enabled by default choice), as a user wants it.

A possible objection to our choice of a *mediator* based architecture, is that this type of architecture does not scale very well (since there is the specific need of a central point of integration); However, considering the dynamicity of the whole CONTEXT-ADDICT project, the approach we have followed seems to be the most natural one.

Also a criticism can be made to the choice of the usage of RDFS(DL) as domain ontology specification language (RDFS(DL) is poorer, in terms of expressive power, than OWL(DL), but this makes it possible to use a reasoner on its structure).

However, we note that in this work we have seen ontologies merely as models used to distribute queries using semantic interconnections: we rewrite queries using semantic links and once we have obtained results from the data sources we have to return to the user these results in terms of the Enriched Domain ontology. Using a formalism like RDFS means that basically we do not express constraints over the global schema apart inclusions expressed by

taxonomies: in this sense, considering sound mappings[1] our query rewriting and distribution algorithm returns only certain answers ([27]).

9.3 Future Works

During our thesis work, we have produced a prototype that, after the fast test phase that has been carried out, seems to work well in most of the tested use cases.

Besides, a lot of work has to be done in order to complete the CONTEXT-ADDICT framework and allow it to do its job correctly and further investigation must be done on the product of this thesis work too.

Let us specify that the testing of our work has been actually focused on *relational* data sources, but many other types of sources can be used with our system, although many components (like semantic extractors or query executors) lack in the system, making impossible to perform tests on this types of sources.

First of all, A more detailed index of the sources has to be implemented since the system need to understand on which sources a query has to be split in order to become a single-source query (e.g. relational datasources) or when it can leave a multi-source query (e.g. RDF/ontology files).

More testing has to be done on the system too; Performing more tests can make more data available and maybe, with the new data, it is possible to understand if some strange (but correct) pattern of query maybe discarded during some of our processing step, suggesting any modifications to our algorithms.

Now that the system is, only in case of relational datasources integration, complete it is also possible to start to design a graphical front-end for the framework, since currently it is difficult to use the system for a final user.

Another thing that must be implemented is a query execution and data integration parallelization (at the moment the integration module architecture is serial), thing that probably will reduce the integration step computation time, resulting in a smaller bottleneck for our application.

In addition it is necessary to modify the architecture of our core system modules, since at the moment they work as a stand-alone application; a server

[1] Mappings stating that some terms belonging to data source ontologies (virtually) contain instances that are a subset of those (virtually) contained by terms of the Domain ontology.

or another type of an end-point implementation would be more suitable, since the system has simply to receive a string which represent a SPARQL query and has to return a "*table-like*" (or a *XML* file) structure that contains the result produced by the received query, once ran on the datasources.

Appendix A

Examples of ontologies

In this appendix we are going to report some of the ontologies that we have used in the previous chapters for our examples and testing purposes.

A.1 Vehicle domain ontology

This ontology is the one which represent the domain in which we are interested; the final user must issue queries on this ontology, and the system must rewrite it in order to make possible to be sent on datasource ontologies. Also final result must be inserted into this ontology, for the final integration step.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:p1="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Woman">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Person"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Risk"/>
  <owl:Class rdf:ID="Motorcycle">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Vehicle"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Car">
```

```
    <rdfs:subClassOf rdf:resource="#Vehicle"/>
  </owl:Class>
  <owl:Class rdf:ID="High">
    <rdfs:subClassOf rdf:resource="#Risk"/>
  </owl:Class>
  <owl:Class rdf:ID="Low">
    <rdfs:subClassOf rdf:resource="#Risk"/>
  </owl:Class>
  <owl:Class rdf:ID="Mid">
    <rdfs:subClassOf rdf:resource="#Risk"/>
  </owl:Class>
  <owl:Class rdf:ID="Man">
    <rdfs:subClassOf rdf:resource="#Person"/>
  </owl:Class>
  <owl:Class rdf:ID="Manufacturer"/>
  <owl:ObjectProperty rdf:ID="drives">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Vehicle"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="hasRiskClass">
    <rdfs:range rdf:resource="#Risk"/>
    <rdfs:domain rdf:resource="#Person"/>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:ID="hasMname">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Manufacturer"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasName">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:FunctionalProperty rdf:ID="hasBrand">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    <rdfs:domain rdf:resource="#Vehicle"/>
    <rdfs:range rdf:resource="#Manufacturer"/>
  </owl:FunctionalProperty>
</rdf:RDF>

<!-- Created with Protege (with OWL Plugin 3.3.1, Build 430)  http://protege.stanford.edu -->
```

A.2 Mapping ontologies

In this section it is possible to review how our mappings between domain and datasources ontologies are made, in order to understand how the rewriting engine and the data integration module works.

A.2.1 Mapping beetwen vehicle domain and rosex1 ontologies

```
<?xml version="1.0"?>

<rdf:RDF xml:base="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/mapping_rosex2.owl"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rosex1SemanticONTO="file:///Users/Lele/Documents/workspace/TIS-ROSEX_progetto/result/tests/rosex1SemanticONTO.owl#"
     xmlns:mapping_rosex2="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/mapping_rosex2.owl#"
     xmlns:vehicledomain="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#">
    <owl:Ontology rdf:about="">
        <owl:imports rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl"/>
        <owl:imports rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl"/>
    </owl:Ontology>


  <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#motorcycle">
    <rdfs:subClassOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Motorcycle"/>
  </rdf:Description>

  <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#auto">
    <rdfs:subClassOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Car"/>
  </rdf:Description>

  <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#brand">
    <rdfs:subClassOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Manufacturer"/>
  </rdf:Description>

  <!-- OBJECT PROPERTIES -->

  <owl:ObjectProperty rdf:about="file:/Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasBrand"/>

  <owl:ObjectProperty rdf:about="file:/Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasRiskClass"/>

  <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#auto.manufacturer">
      <rdfs:subPropertyOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasBrand"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#motorcycle.manufacturer">
      <rdfs:subPropertyOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasBrand"/>
  </owl:ObjectProperty>


  <!-- DATA PROPERTIES -->

  <owl:DatatypeProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#hasMname"/>

  <owl:DatatypeProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#hasName"/>

  <owl:DatatypeProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#brand.name">
      <rdfs:subPropertyOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasMname"/>
  </owl:DatatypeProperty>

</rdf:RDF>
```

A.2.2 Mapping beetwen vehicle domain and rosex2 ontologies

```
<?xml version="1.0"?>
```

```
<rdf:RDF xml:base="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/mapping_rosex3.owl"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rosex2SemanticONTO="file:///Users/Lele/Documents/workspace/TIS-ROSEX_progetto/result/tests/rosex2SemanticONTO.owl#"
     xmlns:mapping_rosex3="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/mapping_rosex3.owl#"
     xmlns:vehicledomain="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#">
    <owl:Ontology rdf:about="">
        <owl:imports rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl"/>
        <owl:imports rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl"/>
    </owl:Ontology>

<owl:Class rdf:ID="M1">
    <owl:equivalentClass>
      <owl:Class>
       <owl:intersectionOf rdf:parseType="Collection">
          <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Man"/>
          <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Woman"/>
       </owl:intersectionOf>
      </owl:Class>
    </owl:equivalentClass>
    <rdfs:comment xml:lang="en">LAV</rdfs:comment>
  </owl:Class>
  <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente">
    <rdfs:subClassOf rdf:resource="#M1"/>
  </rdf:Description>

  <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#motorcycle">
    <rdfs:subClassOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Motorcycle"/>
  </rdf:Description>

  <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#auto">
    <rdfs:subClassOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Car"/>
  </rdf:Description>

  <!-- OBJECT PROPERTIES -->

  <owl:ObjectProperty rdf:about="file:/Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#drives"/>

  <owl:ObjectProperty rdf:about="file:/Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasRiskClass"/>

  <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#drives_auto">
      <rdfs:subPropertyOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#drives"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#drives_moto">
      <rdfs:subPropertyOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#drives"/>
  </owl:ObjectProperty>

  <!-- DATA PROPERTIES -->

  <owl:DatatypeProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#hasName"/>

  <owl:DatatypeProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente.name">
      <rdfs:subPropertyOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasName"/>
  </owl:DatatypeProperty>
</rdf:RDF>
```

A.2.3 Mapping beetwen vehicle domain and rosex4 ontologies

```
<?xml version="1.0"?>
```

```
<rdf:RDF xml:base="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/mapping_rosex4.owl"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rosex1SemanticONTO="file:///Users/Lele/Documents/workspace/TIS-ROSEX_progetto/result/tests/rosex4SemanticONTO.owl#"
     xmlns:mapping_rosex2="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/mapping_rosex4.owl#"
     xmlns:vehicledomain="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#">
    <owl:Ontology rdf:about="">
        <owl:imports rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl"/>
        <owl:imports rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl"/>
    </owl:Ontology>


  <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#motorcycle">
    <rdfs:subClassOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Motorcycle"/>
  </rdf:Description>

  <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#auto">
    <rdfs:subClassOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Car"/>
  </rdf:Description>

  <rdf:Description rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand">
    <rdfs:subClassOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#Manufacturer"/>
  </rdf:Description>

  <!-- OBJECT PROPERTIES -->

  <owl:ObjectProperty rdf:about="file:/Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasBrand"/>

  <owl:ObjectProperty rdf:about="file:/Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasRiskClass"/>

  <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#auto.manufacturer">
      <rdfs:subPropertyOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasBrand"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#motorcycle.manufacturer">
      <rdfs:subPropertyOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasBrand"/>
  </owl:ObjectProperty>


  <!-- DATA PROPERTIES -->

  <owl:DatatypeProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#hasMname"/>

  <owl:DatatypeProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#hasName"/>

  <owl:DatatypeProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand.name">
      <rdfs:subPropertyOf rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/vehicledomain.owl#hasMname"/>
  </owl:DatatypeProperty>

</rdf:RDF>
```

A.3 Datasources ontologies

Finally, in the follow up of this section we will report the three *DSO* used in order to perform queries on the original relational datasources.

This ontology are the *semantic ontologies* extracted by the *ROSEX* module once it is executed on the respective relational source.

A.3.1 rosex1 semantic ontology

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:sem="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#">
 <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#motorcycle">
  <owl:disjointWith>
   <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#auto"/>
  </owl:disjointWith>
  <owl:disjointWith>
   <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#brand"/>
  </owl:disjointWith>
 </owl:Class>
 <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#brand">
  <owl:disjointWith rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#auto"/>
 </owl:Class>
 <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#auto.manufacturer">
  <rdfs:range rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#brand"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#auto"/>
 </owl:ObjectProperty>
 <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#motorcycle.manufacturer">
  <rdfs:range rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#brand"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#motorcycle"/>
 </owl:ObjectProperty>
 <owl:FunctionalProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#brand.name">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#brand"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
 <owl:FunctionalProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#motorcycle.moto_plate">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#motorcycle"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
 <owl:FunctionalProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#auto.auto_plate">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex1SemanticONTO.owl#auto"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
</rdf:RDF>
```

A.3.2 rosex2 semantic ontology

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:sem="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#">
 <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#motorcycle">
  <owl:disjointWith>
   <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente"/>
  </owl:disjointWith>
 </owl:Class>
 <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#auto">
  <owl:disjointWith rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente"/>
  <owl:disjointWith rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#motorcycle"/>
 </owl:Class>
 <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#drives_moto">
  <rdfs:range rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#motorcycle"/>
```

```
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente"/>
 </owl:ObjectProperty>
 <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#drives_auto">
  <rdfs:range rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#auto"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente"/>
 </owl:ObjectProperty>
 <owl:FunctionalProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#motorcycle.moto_plate">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#motorcycle"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
 <owl:FunctionalProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#auto.auto_plate">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#auto"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
 <owl:FunctionalProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente.name">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex2SemanticONTO.owl#cliente"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
</rdf:RDF>
```

A.3.3 rosex4 semantic ontology

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:sem="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#">
 <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#motorcycle">
  <owl:disjointWith>
   <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#auto"/>
  </owl:disjointWith>
  <owl:disjointWith>
   <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand"/>
  </owl:disjointWith>
 </owl:Class>
 <owl:Class rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand">
  <owl:disjointWith rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#auto"/>
 </owl:Class>
 <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#auto.manufacturer">
  <rdfs:range rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#auto"/>
 </owl:ObjectProperty>
 <owl:ObjectProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#motorcycle.manufacturer">
  <rdfs:range rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#motorcycle"/>
 </owl:ObjectProperty>
 <owl:FunctionalProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand.name">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#brand"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
 <owl:FunctionalProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#motorcycle.moto_plate">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#motorcycle"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
 <owl:FunctionalProperty rdf:about="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#auto.auto_plate">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="file:///Users/Lele/Documents/workspace/TIS-RewSparQL_progetto/ontologies/rosex4SemanticONTO.owl#auto"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
</rdf:RDF>
```

Bibliography

[1] http://www.w3.org/DesignIssues/Notation3.

[2] http://www.w3.org/RDF/.

[3] http://www.w3.org/TR/rdf-sparql-query/.

[4] http://www.w3.org/XML/.

[5] *OWL Web Ontology Language Guide.*

[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[7] F. Baader, S. Brandt, and C. Lutz. Pushing the *el* envelope. pages 364 – 369, 2005.

[8] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The description logic handbook: theory, implementation and applications.* Cambridge University Press, 2003.

[9] F. Baader and W. Nutt. Basic description logics. *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95, 2003.

[10] C. Bizer and R. Cyganiak. D2r server: Publishing relational databases on the semantic web. 2006.

[11] C. Bolchini, C. Curino, FA Schreiber, and L. Tanca. Context integration for mobile data tailoring. *Proceedings of the 7th International Conference on Mobile Data Management (MDM'06)-Volume 00*, 2006.

[12] Cristiana Bolchini, Carlo Curino, Marco Giorgetta, Alessandro Giusti, Antonio Miele, Fabio A. Schreiber, and Letizia Tanca. *Polidbms: Design and prototype implementation of a dbms for portable devices.* 2004.

[13] Cristiana Bolchini, Carlo Curino, Fabio A. Schreiber, and Letizia Tanca. Context integration for mobile data tailoring. In *Proc. IEEE/ACM of Int. Conf. on Mobile Data Management*. IEEE, ACM, May 2006.

[14] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, and R. Rosati. Mastro-i: Efficient integration of relational data through dl ontologies. 2007.

[15] Huajun Chen, Zhaohui Wu, Heng Wang, and Yuxin Mao. Rdf/rdfs-based relational database integration. pages 94 – 94, 2006.

[16] Carlo Curino, Giorgio Orsi, and Letizia Tanca. X-som: A flexible ontology mapper. In *DEXA Workshops*, pages 424 – 428, 2007.

[17] Carlo A. Curino, Letizia Tanca, and Carlo Zaniolo. Information systems integration and evolution: Ontologies at rescue. In *International Workshop on Semantic Technologies in System Maintenance (STSM)*, 2008.

[18] C. P. de Laborda and S. Conrad. Relational.owl: a data and schema representation format based on owl. In *Proc. of the 2nd Asia-Pacific Conf. on conceptual modelling APCM'05*, volume 43, pages 89–96, 2005.

[19] J. Dolby, A. Fokoue, A. Kalyanpur, L. Ma, E. Schonberg, K. Srinivas, and X. Sun. Scalable grounded conjunctive query evaluation over large and expressive knowledge bases. pages 403 – 418, 2008.

[20] F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. *Principles of Knowledge Representation*, pages 191–236, 1996.

[21] W. Grosso, H. Eriksson, R. Fergerson, J. Gennari, S. Tu, and M. Musen. Knowledge modeling at the millennium the design and evolution of protege. 2000.

[22] P. Haase and Y. Wang. A decentralized infrastructure for query answering over distributed ontologies. pages 1351 – 1356, 2007.

[23] A. Halevy, Z. Ivesa, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer-data management system. 2004.

[24] Alon Y. Halevy. Theory of answering queries using views. *IGMOD Record (ACM Special Interest Group on Management of Data)*, 29(4):40 – 47, 2000.

[25] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal The International Journal on Very Large Data Bases*, 10(4):270 – 294, 2001.

[26] I. Horrocks and P.F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*, pages 17–29, 2003.

[27] G. Inglese. Ontology-based query processing in a dynamic data integration system. Master's thesis, Politecnico di Milano, 2007.

[28] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo Cuenca-Grau, and James Hendler. Swoop - a web ontology editing browser. *Journal of Web Semantics*, 4(2):144 – 153, 2006.

[29] M. Krötzsch, S. Rudolph, and P. Hitzler. Elp: Tractable rules for owl 2. pages 649 – 664, 2008.

[30] T. Lee. Arithmetical definability over finite structures. *Mathematical Logic Quarterly 49*, 4:385 – 393, 2003.

[31] M. Lenzerini. Data integration: a theoretical perspective. *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, 2002.

[32] A. Levy. The information manifold approach to data integration. 1998.

[33] C. Lutz, D. Toman, and F. Wolter. Conjunctive query answering in the description logic *el* using a relational database system. pages 2070 – 2075, 2009.

[34] N. J. Nilsson M. R. Genesereth. *Logical Foundations of Artificial Intelligence*. Morgan Kaufman Publishers, San Mateo, California, 1987.

[35] L. Macagnino. Estrazione di ontologie da basi di dati relzionali basata sulla semantica. Master's thesis, Politecnico di Milano, 2006.

[36] A. D. Maedche. *Ontology Learning for the Semantic Web*. Kluwer Academic Publishers, Norwell, Massachusetts, 2003.

[37] A. Magni. Relazione progetto tecnologie per i sistemi informativi. 2006.

[38] B. Motik, U. Sattler, and R. Studer. Query answering for owl-dl with rules. *Intl Journal of Web Semantics 3*, 1:41 – 60, 2005.

[39] G. Orsi. An ontology-based data integration system: Solving semantic inconsistencies. Master's thesis, Politecnico di Milano, 2006.

[40] A. Poggi. Structured and semi-structured data integration, 2006.

[41] Guus Schreiber and Mike Dean. Owl web ontology language reference. w3c recommendation. 2004.

[42] T. Gruber. A Translation Approach to Portable Ontology Specifications. *International Journal of Human and Computer Studies*, 1993.

[43] C. Curino G. Orsi E. Panigati L. Tanca. Accessing and documenting relational databases through owl ontologies. *FQAS 2009*, pages 431 – 442, 2009.

[44] H. Wache, T. Voegele, T. Visser, H. Stuckenschmidt, H. Schuster, G. Neumann, and S. Huebner. *Ontology-based integration of information - a survey of existing approaches*. 2001.

[45] W. Xing, O. Corcho, C. Goble, and M. D. Dikaiakos. Active ontology: An information integration approach for dynamic information sources. 2007.